IEEE Std 1003.5, 1999 Edition
(Incorporates IEEE Std 1003.5-1992,
IEEE Std 1003.5b-1996, and
IEEE Std 1003.5c-1998)

IEEE Standard for Information Technology—POSIX® Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API)

Includes

Amendment 1: Realtime Extensions

and

Amendment 2: Protocol-Independent Interfaces

Sponsor

Portable Applications Standards Committee
of the
IEEE Computer Society

Approved 8 December 1998

IEEE Standards Board

Abstract: This standard is part of the POSIX® series of standards for applications and user interfaces to open systems. It defines the Ada language bindings as package specifications and accompanying textual descriptions of the application program interface (API). This standard supports application portability at the source code level through the binding between ISO 8652:1995 (Ada) and ISO/IEC 9945-1:1996 (IEEE Std 1003.1-1996) (POSIX) as amended by IEEE P1003.1g/D6.6. Terminology and general requirements, process primitives, the process environment, files and directories, input and output primaries, device- and class-specific functions, language-specific services for Ada, system databases, synchronization, memory management, execution scheduling, clocks and timers, message passing, task management, the XTI and socket detailed network interfaces, event management, network support functions, and protocol-specific mappings are covered. It also specifies behavior to support the binding that must be proviced by the Ada.
Keywords: Ada, API, application portability, computer language bindings, information exchange, interprocess communication, networks, open systems, operating systems, portable application, POSIX, POSIX language bindings, protocol-specific, protocol-independent, real-time, sockets, thread, XTI.

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

Print: ISBN 0-7381-1539-8 SH94710
PDF: ISBN 0-7381-1540-1 SS94710

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.

Contents

TABLES

Introduction

(This introduction is not a part of IEEE Std 1003.5, 1999 Edition, IEEE Standard for Information Technology—POSIX Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API)—Amendment 2: Protocol-Independent Interfaces, but is included for information only.)

This standard is an amended version of IEEE Std 1003.5b-1996. The basic goal of this standard is to provide an Ada application program interface for the language-independent services made accessible to C-language applications programs by the interfaces defined in ISO/IEC 9945-1:1996 (IEEE Std 1003.1-1996) {2} as amended by IEEE P1003.1g {B14}.

The intent is to support portability of Ada applications via a standard binding to the services provided by a POSIX-conforming operating system. POSIX is defined by the standard C-language interfaces cited above. Therefore, much of the work in producing this standard was deciding what features of those C-language interfaces represented POSIX functionality, as opposed to C-language-specific features.

This standard provides package specifications and accompanying textual description for a set of Ada packages that represent the POSIX system. This standard also specifies behavior to support the binding that must be provided by the Ada compilation system, and further defines behavior specified as implementation defined in the Ada language standard (particularly in the area of `Text_IO`) for use in a POSIX environment.

The emphasis in POSIX is on application program portability, so the interfaces in this standard are not intended to be sufficient to implement an Ada compilation system or a POSIX shell as defined in IEEE Standard 1003.2 {B16}. For an application, the intent is that a Strictly Conforming POSIX.5 Application (one that uses only the facilities in this standard and that does not depend on implementation-defined behavior) can be ported to any Conforming Implementation of these interfaces and that the binding makes it easy to identify where a program is not strictly conforming and makes such programs easier to port.

Organization of This Standard

The standard is divided into three parts:

— Statement of scope, list of normative references, and conformance information (Section 1)

— Definitions and global concepts (Section 2)

— The various interface facilities (Sections 3 through 19)

The content of the sections parallels that of the correspondingly numbered sections of ISO/IEC 9945-1:1996 and IEEE P1003.1g/D6.6, with a few changes required to accomodate differences between the Ada and C-language interfaces.This standard

has no Section 10, since there is no Ada binding for that Section 10 (Data Interchange Formats) of ISO/IEC 9945-1:1996.

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

Related Standards Activities

Activities to extend this standard to address additional requirements can be anticipated in the future[1)].

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

Anyone interested in participating in the PASC working groups addressing these issues should send his or her 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, USA, and ask to have this information forwarded to the chair of the appropriate PASC working group. A person who is interested in participating in this work at the international level should contact his or her 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, USA; Telephone: +1 202 371-0101; FAX: +1 202 728-9614.

IEEE Std 1003.5-1992 was prepared by the 1003.5 Working Group, sponsored by the Technical Committee on Operating Systems and Applications Environments of the IEEE Computer Society. At the time IEEE Std 1003.5-1992 was approved the membership of the IEEE P1003.5 working group was as follows:

Technical Committee on Operating Systems and Application Environments (TCOS)

Chair:	Jehan-François Pâris

TCOS Standards Subcommittee

Chair:	Jim Isaak
Vice Chairs:	Ralph Barker
	Hal Jesperson
	Lorraine Kevra
	Pete Meier
	Andrew Twigger
Treasurer:	Peter Smith
Secretary:	Shane McCarron

P1003.5 Working Group Officials

Chair:	James P. Lonjers
	Steven Deller (1989-1991)
	Major Terrence Fong (1988-1989)
Vice Chairs:	James P. Lonjers (1990-1991)
	Major Terrence Fong (1989-1990)
	Stowe Boyd (1988-1989)
Editors:	David Emery
	Hal Jesperson
	Steven Deller (1988-1989)
Rationale Editor:	Mitch Gart
Secretary:	C. Jayne Baker
	David Emery (1988-1989)

Technical Reviewers

Ted Baker	David Emery	Jim Lonjers
Steven Deller	Mitch Gart	Jim Moore
Dennis Doubleday		Stephen Schwarm

Working Group

Ted Baker	Michael Gillam	Sue LeGrand
Stowe Boyd	Al Globus	James Lonjers
Bevin Brett	Mars Gralia	James Moore
Charles Brown	Jayne Guyse	Mark Ruddock
Robert Brown	Ken Harvey	Stephen Schwarm
Bhavesh Damania	Ruth Hirt	Michael Shapiro
Steven Deller	Jeff Hooley	Brian Sullivan
Dennis Doubleday	Michael Kjolsrud	Del Swanson
David Emery	Peter Krupp	Robert Voigt
Terry Fong	James Leathrum	Olle Wikstrom
Mitchell Gart		John Zenor

The following persons where members of the balloting group for IEEE Std 1003.5-1992.

Harold C. Adams
John S. Adams
Omar Ahmed
David Allen
Charles J. Antonelli
B. Ardary
David Athersych
Randall Atkinson
Randal J. August
Kenneth A. Austin
Carolyn J. Baker
Robert L. Baker
Ted Baker
James Baldo
Brad Balfour
Gary E. Barnes
Mitchell C. Barnhart
Randall Barron
Steven Barryte
Barbara K. Beauchamp
Gary Beerman
E. Jerome Bell
Donald Bennett
Peter A. Berggren
Mark Biggar
Robert Bismuth
Alex Blakemore
Stephen Blanchette, Jr.
Pieter Botman
Stowe Boyd
Carl Brandon
Philip Brashear
Joseph P. Brazy
Mark S. Breckenridge
Ronald F. Brender
Jim Briggs
Thomas C. Brooke
Jerry R. Brookshire
Charles O. Brown
Elizabeth B. Brown
Jane C. Bryan
Gary L. Burt
Christopher Byrnes
David Calloway
Nicholas A. Camillone
Kenneth W. Campbell
Rick Carle
David J. Carlson
Dana Carson
Jeffrey R. Carter
Jerry Cashin
H. L. Catala
Larry Chandler
Andy Cheese
James Chelini
Andrew Chung
Brad Clark
Lori A. Clarke
Norman H. Cohen
Edward Colbert
Phillippe Collard
Robert A. Conti
William M. Corwin
Mike Cossey
John Courtney
Donald Cragun
Richard A. Crawford
Jim Creegan
Phyllis Crill
John J. Cupak
Charles Dana
William H. Dashiell
David Davis
Rich DeBernardo
David DeFanti
Mike Dean
Dave Decot
Steven Deller
Jorge Diaz-Herrera
Michael B. Dillencourt
James H. Dobbins
Audrey Dorofee
Terence Dowling
Diptendu Dutta
Eugene Edelstein
Theodore F. Elbert
Richard W. Elwood
David Emery
Arny Engelson
Philip H. Enslow
William Eventoff
Gary Falacara
John H. Fauerby
Charles A. Finnell
Jeffery Fischer
Shayne Flint
Terence Fong
Edward J. Forbes, Jr.
Roy S. Freedman
Randal S. Freier
Dale J. Gaumer
Larry Gearhart
K. M. George
Gregory A. Gicca
Robert T. Goettge
Phillip Goldstein
Roger Golliver
William N. Goolsby
William J. Goulet
Mars J. Gralia
Allen L. Grau
Charles R. Grauling
Daniel Green
Tom Griest
F. Grize
Ernesto Guerrieri
Lawrence M. Gunther
R. N. Hagen
Charles Hammons
Peter A. Hansen
Sam Harbaugh
Samuel Harbison
David S. Hardin
Charles Harkey
Loren L. Hart
Thomas S. Hawker
Clark M. Hay
Ralph Hayward
John Craig Heberle
William Hefley
A. Marlow Henne
Donald C. Hill
Norman Hines
C. Michael Holloway
Jeffrey Hooley
Joseph P. Hoolihan
Tom Housman
Richard Howard
Norman R. Howes
Lynne M. Hubbs
David K. Hughes
Richard G. Hull
Jeremy James
Hal Jespersen
Darryl N. Johnson
Bruce Johnston
Alain Jouchoux
Juern Juergens
Steven Kahn
Fumimiko Kamijo
Alan Kaminsky
Ling Kan
Karl Kelley
Robert H. C. Kemp
Judy S. Kerner
James J. Keys
Paul J. King
Hans R. Klay
Kenneth Kloss
Robert Knighten
Joseph B. Kolb
John C. Krasnowski
Lak Ming Lam
Rudolf C. Landwehr
Charles F. Lanman

Gary Lauther
Patricia K. Lawlis
Scott A. Leschke
M. Levitz
Stephen H. Levy
F. C. Lim
Timothy E. Lindquist
J. J. Logan
James P. Lonjers
Warren E. Loper
Mark Loveland
George A. Ludgate
Sonny Lundahl
Wesley Mackey
Austin J. Maher
James Maloney
Roger Martin
Robert Mathis
Fred Maymir-Ducharme
Catherine McDonald
Robert L. McGarvey
Daniel L. McNamee
Robert McWhirter
Nancy R. Mead
Geoff Mendal
Jay Michael
Gary W. Miller
Robert E. Miller
Judah Mogilensky
Al Mok
Charles S. Mooney
James D. Mooney
Freeman Moore
James W. Moore
Jerry A. Moore
John I. Moore, Jr.
Duncan Morrill
M. W. Morron
Gary Mrenak
David G. Mullens
Richard E. Nesee
Sai Lun Ng
Daniel Nissen
Karl Nyberg
James O'Day
Evelyn Obaid
Patricia Oberndorf
Kurt M. Olender
S. Ron Oliver
James K. Parrish
Thomas Parrish
Offer Pazy
Walt Penney
Guido Persch
Flavio Petersen
Thomas A. Peterson
George W. Petznick
Hane W. Polzer
J. Pottmyer
Charles Pow
Eileen Quann
Paul Rabin
John Reddan
W. Scott Redmon
Gregg Reed
Carl Reinert
Judith Richardson
Richard A. Rink
Clyde Roby
C. Allan Rofer
Hyman Rosen
Jerome D. Rosen
Frederick M. Rysz
Agnes M. Sardi
Robert J. Satnik
Allen Saxton
Lorne H. Schachter
F. P. Schauer
Alfred H. Scholldorf
Ron Schroeder
Mike Schultz
W. L. Schultz
Fritz Schulz
Leonard Seagren
Richard Seibel
Lawrence H. Shafer
Michael D. Shapiro
John G. Shea
Nagy M. Shehad
Dan Shia
Thomas E. Shields
Keith Shillington
David Shochat
Stephen Schwarm
Robert Charles Shock
Jerome L. Sibol, Jr.
Lee Silverthorn
Ronald Skoog
Thomas J. Smith
Charles Snyder
Jon S. Squire
Jeff Stevenson
Brian Sullivan
Del Swanson
S. Tucker Taft
Ravi Tavakley
Donn S. Terry
John A. Thalhamer
William J. Thomas
Peter L. Thompson
James L. Troy
Roger Tubby
Mark-Rene Uchida
L. David Umbaugh
Robert B. Urling
Evelyn M. Uzzle
Frances Van Scoy
Leonard Vanek
Michael W. Vannier
Uwe Wacker
Robert N. Wagoner
Mary Wall
Stephen R. Walli
Neal Walters
Kenneth Wasmundt
William Webster
J. Richard Weger
Brian Weis
Robert Weissensee
Michael K. Welter
Stephen Wersan
Thomas Wheeler
William Whitaker
Bruce Wieand
David Willcox
David C. Willet
David Williamson
Paul A. Willis
David H. Winfield
David C. Wolfe
Paul A. Wolfgang
Michal Young
Oren Yuan
Janusz Zalewski
K. M. Zemrowski
John Zolnowsky

When the IEEE Standards Board approved IEEE Std 1003.5-1992 on 18 June 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 Clyde R. Camp	Ben C. Johnson	John L. Rankins
	Walter J. Karplus	Wallace S. Read
Donald C. Fleckenstein	Ivor N. Knight	Ronald H. Reimer
Jay Forster *	Joseph L. 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
James Beall
Richard B. Engleman
David E. Soffrin
Stanley Warshaw

Mary Lynne Nielsen
IEEE Standards Project Editor

IEEE Std 1003.5b-1996 was prepared by the P1003.5 working group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time IEEE Std 1003.5b-1996 was approved the membership of the P1003.5 working group was as follows:

Portable Applications Standards Committee (PASC)

Chair: Jehan-François Pâris

PASC Standards Subcommittee

Chair:	Lowell Johnson
Vice Chair:	Charles Severance
Functional Chairs:	Barry Needham
	John Spencer
	Jay Ashford
	Andrew Josey
Treasurer:	Peter Smith
Secretary:	Charles Severance

IEEE P1003.5 Working Group Officials

Chair:	James P. Lonjers (1991-1994)
	Stephen Schwarm (1995-1996)
	Ted Baker (1996-1997)
Vice Chairs:	Stephen Schwarm (1991-1995)
	Randy Greene (1995-1996)
	David Emery (1003.5 Interpretations)
Editor:	Ted Baker (P1003.5b)
Rationale Editor:	Lee Lucas
Secretary:	C. Jayne Guyse (1991-1993)
	Peter Obermayer (1994-1996)

Technical Reviewers

Ted Baker	Lee Lucas	Henry H. Robbins
Mark Faulk	Peter Obermayer	Stephen Schwarm
Ted Giering	Offer Pazy	Del Swanson
Randy Greene	Ruth A. Peek	Laurent Visconti
	Ed Posnak	

Working Group

Theodore P. (Ted) Baker	David K. Hughes	Ray Ricco
Bevin Brett	James Lonjers	Henry H. Robbins
Steven Deller	Lee Lucas	Stephen Schwarm
David Emery	Peter Obermayer	Jim Smith
Mark Faulk	James T. Oblinger	Del Swanson
Randy Greene	Offer Pazy	Laurent Visconti
C. Jayne Guyse	Ruth A. Peek	John Zenor

The following persons were members of the balloting group for IEEE Std 1003.5b-1996:

Alejandro A. Alonso	Norman R. Howes	Dave Plauger
Theodore P. Baker	David K. Hughes	Arlan Pool
Robert Barned	Judy Kerner	Henry H. Robbins
Andy Bihain	Philippe Kruchten	Stephen Schwarm
William M. Corwin	Thomas M. Kurihara	Leonard W. Seagren
Steven Deller	Arthur Licht	Robert Alan Siegel
David Emery	C. Douglass Locke	Dennis C. Stewart
Philip H. Enslow	James P. Lonjers	Alfred Strohmeier
Michael Gonzalez	Lee W. Lucas	Del Swanson
C. Jayne Guyse	Roland McGrath	Mark-René Uchida
Joe Gwinn	Paul Murdock	USENIX
Patrick Hebert	James T. Oblinger	Victor Fay-Wolfe
Steven Howell	Offer Pazy	John Zenor

When the IEEE Standards Board approved IEEE Std 1003.5b-1996 on 20 June 1996, it had the following membership:

Donald C. Loughry, *Chair* **Richard J. Holleman,** *Vice Chair*

Andrew G. Salem, *Secretary*

Gilles A. Baril	E. G. "Al" Kiener	Jose R. Ramos
Clyde R. Camp	Joseph L. Koepfinger*	Arthur K. Reilly
Joseph A. Cannatelli	Stephen R. Lambert	Ronald H. Reimer
Stephen L. Diamond	Lawrence V. McCall	Gary S. Robinson
Harold E. Epstein	Bruce McClung	Ingo Rüsch
Donald C. Fleckenstein	Marco W. Migliaro	John S. Ryan
Jay Forster *	Mary Lou Padgett	Chee Kiow Tan
Donald N. Heirman	John W. Pope	Leonard L. Tripp
Ben C. Johnson		Howard L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
Alan H. Cookson
Chester C. Taylor

Lisa S. Young
IEEE Standards Project Editor

IEEE Std 1003.5c-1998 was prepared by the P1003.5 working group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time IEEE Std 1003.5c-1998 was approved the membership of the P1003.5 working group was as follows:

PASC Standards Subcommittee

Chair:	Lowell Johnson
Vice Chair:	Joe Gwinn
Functional Chairs:	Curtis Royster
	Jason Zions
	Jay Ashford
	Andrew Josey
Secretary:	Nick Stoughton

IEEE P1003.5 Working Group Officials

Chair:	Ted Baker
Vice Chair:	Linda Harowicz
Editors:	Craig Meyer (P1003.5c Editor)

Working Group

Ted Baker	Greg Bussiere	Craig Meyer
	Linda Harowicz	

The following persons were voting members of the balloting group for IEEE Std 1003.5c-1998:

Ted Baker	Mars J. Gralia	Craig Meyer
Bob Barned	Linda Harowicz	Stephen Michell
Carl Brandon	Matthew Heaney	Howard E. Neely
Greg Bussiere	Niklas Holsti	Peter E. Obermayer
Jorge L. Diaz-Herrera	David C. Hoos	James T. Oblinger
Victor Giddings	Michael J. Kamrad	Jan Pukite
Michael Gonzalez	Mark Lundquist	Curtis Royster

The following persons were nonvoting members of the balloting group for IEEE Std 1003.5c-1998:

Robert E. Allen	Robert A. Duff	Robert C. Leif
A. Barnes	W. Douglas Findly	B. Craig Meyers
Ronald Bjornseth	Anthony Gargaro	James W. Moore
Stephen E. Blake	David Gross	Tushar Pokle
Chad Bremmon	Maretta Holden	Bill Pritchet
Vincent Celier	Harry Joiner	Michael Rohan
Hans O. Danielsson	Rush Kester	David Shochat
John Davies	Jim Kroening	Lynn Stuckey
Peter Dencker	Mark Larsen	Terry J. Westley
Guido Duerinckx		Stephen Whiting

c

When the IEEE Standards Board approved IEEE Std 1003.5c-1998 on 8 December 1998, it had the following membership:

Richard J. Holleman, *Chair* **Donald N. Heirman,** *Vice Chair*

Judith Gorman, *Secretary*

James H. Gurney	Jim D. Isaak	L. Bruce McClung
Satish K. Aggarwal	Lowell G. Johnson	Louis-François Pau
Clyde R. Camp	Robert Kennelly	Ronald C. Petersen
Gary R. Engman	E. G. "Al" Kiener	Gerald H. Peterson
Harold E. Epstein	Joseph L. Koepfinger*	John B. Posey
Jay Forster*	Stephen R. Lambert	Gary S. Robinson
Thomas F. Garrity	Jim Logothetis	Hans E. Weinrich
Ruben D. Garzon	Donald C. Loughry	Donald W. Zipse

*Member Emeritus

Yvette Ho Sang
IEEE Standards Project Editor

c

IEEE Standard for Information Technology—POSIX® Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API)—Amendment 2: Protocol Independent Interfaces

c

Section 1: General

1.1 Scope

This standard defines a set of system application program interfaces to operating system services. These interfaces provide access via the Ada programming language to the same operating system services for which C-language interfaces are specified in ISO/IEC 9945-1:1996 {2} [1)][2)] and IEEE P1003.1g {B14}.

c

The purpose of this standard is to support application portability at the Ada source code level. This standard is intended to be used by both application developers and system implementors.

This standard is intended to be compatible with implementations of the 1995 revision to the Ada language standard (ISO/IEC 8652:1995 {1}). Fall-back approaches compatible with implementations of the original Ada language standard (ISO/IEC 8652:1987 {B5}) are also provided (see 1.3).

1) Plain numbers in curly braces correspond to those of the normative references in 1.2. Numbers preceded by a "B" in curly braces correspond to those of the bibliography in Annex A. See 2.1 for the description of this and the other typographical conventions followed in this document.

2) A language-independent definitions of this standard was once under development, but work on that project was suspended.

c

This standard is intended to contain no specifications that conflict with “Year 2000” requirements. c

This standard comprises three major components:

— Definitions for terminology and concepts, and definitions and specifications that govern program structures, language-system interaction, and related requirements.

— Definitions of the specific Ada interfaces to the system services defined by the POSIX standards, presented in the form of Ada packages.

— Interpretations of Ada semantics with respect to the POSIX standards.

The following areas are outside the scope of this standard:

(1) User interface (shell) and commands associated with Ada program development.

(2) Ada bindings to the archive/interchange file formats for *tar* and *cpio*.

(3) Network protocols.

(4) Graphics and windowing interfaces.

(5) Database management system interfaces.

(6) Object or binary code portability.

(7) System configuration and resource availability.

(8) Interfaces to the Ada runtime system.

When the XTI Detailed Network Interface option and/or the Sockets Detailed Network Interface option are supported, then a set of DNI’s (see 2.2.3.26) are also within the scope of this standard. A DNI is intended to provide access to protocol-specific features of the underlying network for highly portable applications that need access to sophisticated network features. The DNI’s are based on the SPG4 XTI and 4.4 BSD socket specifications.

The following areas are outside of the scope of the DNI’s:

— Interface to manipulate underlying protocol implementations

— Network management interface

— Interface to manipulate performance-specific features

— Definition for protocol address formats c

This standard describes the external characteristics and facilities that are of importance to applications developers, rather than the implementation approaches that may be employed to achieve them. Special emphasis is placed on those facilities and capabilities needed for the broad spectrum of applications.

This standard has been defined exclusively at the source code level. The objective is that a Strictly Conforming POSIX.5 Application can be compiled to execute on any conforming implementation, within the portability of the application Ada code itself.

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. Members of IEC and ISO maintain registers of currently valid International Standards.

{1} ISO/IEC 8652:1995[3], *Information technology—Programming languages—Ada* [Revision of first edition (ISO/IEC 8652:1987)], 15 February 1995.

{2} ISO/IEC 9945-1:1996 (IEEE Std 1003.1-1996[4]), *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].* This edition incorporates the extensions for realtime applications (POSIX.1b, POSIX.1i) and threads (POSIX.1c).

{3} ISO/IEC 8072:1996 (CCIT X.214:1988[5]), *Information technology—Open systems interconnection—Transport service definition.*

{4} ISO/IEC 8073:1992 (CCITT X.224:1992), *Information technology—Telecommunications and information exchange between systems—Open systems interconnection— Protocol for providing the connection-mode transport service.*

{5} ISO/IEC 8208:1995, *Information technology—Data communications—X.25 Packet layer protocol for data terminal equipment.*

{6} ISO/IEC 8348:1996, *Information technology—Open systems interconnection—Network service definition.*

{7} ISO/IEC 8473-1:1994, *Information technology—Protocol for providing the connectionless-mode network service: Protocol specification.*

{8} ISO/IEC 8473-3:1995, *Information technology—Protocol for providing the connectionless-mode network service: Provision of the underlying service by an X.25 subnetwork.*

{9} ISO/IEC 8602:1995, *Information Technology—Protocol for providing the OSI connectionless-mode transport service.*

{10} ISO/IEC 8878:1992, *Information technology—Telecommunications and information exchange between systems—Use of X.25 to provide the OSI connection-mode network service.*

---

3) ISO/IEC publications can be obtained from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch) or from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). ISO/IEC publications can also be obtained in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA (http://www.ansi.org).

4) IEEE standards publications are available from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://www.standards.ieee.org).

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

{11} ISO/IEC ISP 11188-3:1996, *Information Technology—International standardization profile—Common upper layer requirements—Part 3: Minimal OSI upper layer facilities*.

{12} IETF RFC 768:1980[6], *User Datagram Protocol*.

{13} IETF RFC 791:1981, *Internet Protocol DARPA Internet Program Protocol Specification*.

{14} IETF RFC 793:1981, *Transmission Control Protocol DARPA Internet Program Protocol Specification*.

{15} IETF RFC 919:1984, *Broadcasting Internet Datagrams*.

{16} IETF RFC 922:1984, *Broadcasting Internet Datagrams in the Presence of Subnets*.

{17} IETF RFC 1006:1987, *ISO Transport Service on Top of the TCP, Version: 3*.

{18} IETF RFC 1122:1989, *Requirements for Internet Hosts—Communication Layers*.

NOTE: Abbreviations for the above standards are defined in 2.2.3.

1.3 Conformance

1.3.1 Implementation Conformance

1.3.1.1 Requirements

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

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

(2) For packages and subprograms that are defined to be dependent on implementation options, either the runtime behavior shall be as defined by this standard or references to the name shall be rejected at compile or link time.

(3) The system may provide additional facilities not required by this standard. Nonstandard extensions shall be identified as such in the system documentation. Nonstandard extensions, when used, may change the behavior of functions or facilities defined by this standard, but only if the application *activates an extension*, by reference to an implementation-defined extension package in a **`with`** clause, calling a subprogram in an implementation-defined extension package, or using an implementation-definedconfiguration pragma(see 10.1.5 (8) of {1}).

The conformance document shall define an environment in which an application can be run with the behavior specified by this standard. In no case except package name conflicts shall such an environment require modification of a Strictly Conforming POSIX.5 Application. An implementation shall not change package specifications in this standard except by the following:

6) IETF documents can be obtained in printed form from the Network Information Center, Network Solutions, 14200 Park Meadow Drive, Suite 200, Chantilly, VA 22021, USA, or in electronic form via FTP over the Internet from nic.ddn.mil.

(a) Adding **with** clauses, pragmas, representation specifications, and comments.

(b) Replacing instances of the words *implementation-defined* with appropriate value(s).

(c) Optionally changing the type POSIX_Character (see 2.4.2).

(d) Adding or changing private parts.

(e) Making any other changes that are lexically transparent to Ada compilers.

(f) Adding specific declarations for which permission is explicitly granted in the text of this standard.

(g) For Ada 95 implementations, changing packages with names of the form POSIX_XXX to child packages of the package POSIX with names of the form POSIX.XXX and adding library-level renaming declarations of the form:

```
with POSIX.XXX;
package POSIX_XXX renames POSIX.XXX;
```

*NOTE: The implementor does not have permission to change underscores to dots in the **with** clauses of the package specifications defined by this standard.*

Packages defined for the sockets and XTI interfaces shall be renamed as indicated in Table 1.1.

Table 1.1 – Sockets and XTI Package Renaming

Ada 83 Package Name	Ada 95 Package Name
POSIX_Event_Management	POSIX.Event_Management
POSIX_Sockets	POSIX.Sockets
POSIX_Sockets_Local	POSIX.Sockets.Local
POSIX_Sockets_Internet	POSIX.Sockets.Internet
POSIX_Sockets_ISO	POSIX.Sockets.ISO
POSIX_XTI	POSIX.XTI
POSIX_XTI_Internet	POSIX.XTI.Internet
POSIX_XTI_ISO	POSIX.XTI.ISO
POSIX_XTI_mOSI	POSIX.XTI.mOSI

(h) For implementations that only support the original Ada language standard {B5}, changing Ada 95 general access types to ordinary access types. c

(i) Where a subprogram has a parameter with a default value expression in the final position, omitting the default value expression and providing an equivalent overloaded definition of the same function name, without the last parameter, if this change results in a gain in efficiency. For example,

```
procedure P(X: T:= E);
```

may be replaced by

```
procedure P(X: T);
procedure P;
```

Calling the version of procedure P without the parameter X shall have the same effect as calling P(X=>E).

(4) Only the following types may be implemented using Ada access types:

— POSIX.POSIX_String_List
— POSIX_Asynchronous_IO.AIO_Descriptor
— POSIX_Condition_Variables.Attributes
— POSIX_Condition_Variables.Condition
— POSIX_Condition_Variables.Condition_Descriptor
— POSIX_Event_Management.File_Descriptor_Set c
— POSIX_Message_Queues.Message_Queue_Descriptor
— POSIX_Mutexes.Attributes
— POSIX_Mutexes.Mutex
— POSIX_Mutexes.Mutex_Descriptor
— POSIX_Process_Environment.Environment
— POSIX_Process_Primitives.Process_Template
— POSIX_Semaphores.Semaphore
— POSIX_Semaphores.Semaphore_Descriptor
— POSIX_Sockets.IO_Vector_Array_Pointer
— POSIX_Sockets.Socket_Address_Info_List
— POSIX_Sockets_Internet.Database_Array_Pointer
— POSIX_Sockets_Internet.Internet_Socket_Address_Pointer
— POSIX_Sockets_ISO.ISO_Socket_Address_Pointer
— POSIX_Sockets_Local.Local_Socket_Address_Pointer
— POSIX_XTI.Octet_Buffer_Pointer
— POSIX_XTI.Protocol_Option_List
— POSIX_XTI.Protocol_Option_List_Pointer
— POSIX_XTI_Internet.Internet_XTI_Address_Pointer
— POSIX_XTI_Internet.Database_Array_Pointer
— POSIX_XTI_ISO.ISO_XTI_Address_Pointer
— POSIX_XTI_mOSI.mOSI_XTI_Address_Pointer
— POSIX_XTI_mOSI.Presentation_Context_List
— POSIX_XTI_mOSI.Syntax_Object_List c
— Any other type specifically defined as an Ada access type in this standard

No other types defined in this standard shall be implemented as access types or types with components of an access type.

(5) A POSIX implementation shall not raise Program_Error on elaboration of a POSIX package or on execution of a POSIX subprogram, due to elaboration order dependencies in the POSIX implementation.

(6) Except as explicitly provided for in this standard, POSIX.POSIX_Error and Standard.Storage_Error are the only exceptions that shall be raised by operations declared in this standard.

(7) Wherever this standard states a requirement on an implementation, without any other explicit qualification, the requirement applies to all conforming implementations.

1.3.1.2 Documentation

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

The conformance document shall contain a statement that indicates the full name, number, and date of the standard that applies. The conformance document may also list software standards approved by ISO/IEC or any ISO/IEC member body that are available for use by a ISO/IEC Strictly Conforming POSIX.5 Application (see 1.3.2.2.1). Applicable characteristics whose documentation is required by one of these standards, or by standards of government bodies, may also be included.

The document shall describe the contents of the packages `POSIX`, `POSIX_Limits`, `POSIX_Options`, `POSIX_Configurable_File_Limits`, and `POSIX_Configurable_System_Limits` and shall state values, the conditions under which those values may change, and the limits of such variations, if any.

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

No specifications other than those described in this subclause shall be present in the conformance document.

The phrase “shall be documented” in this standard means that documentation of the feature shall appear in the conformance document, as described previously, unless the system documentation is explicitly mentioned.

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

1.3.1.3 Conforming Implementation Options

The symbolic constants in the packages `POSIX`, `POSIX_Limits`, and `POSIX_Options` provide declarations for a range of variation in the standard that is implementation defined. The packages `POSIX_Configurable_System_Limits` and `POSIX_Configurable_File_Limits` contain subprograms that may further constrain the implementation-defined ranges.

An implementation is required to define all of the subprograms for all of the operations defined in this standard, including those whose implementation is optional. If an unimplemented feature is used, the exception `POSIX.POSIX_Error` shall be raised, with the error code `POSIX.Operation_Not_Implemented`.

1.3.2 Application Conformance

All applications claiming conformance to this standard shall fall within one of the categories defined in the following subclauses.

1.3.2.1 Strictly Conforming POSIX.5 Application

A Strictly Conforming POSIX.5 Application is an application that requires only the facilities described in this standard and {1}. Such an application shall be designed to work for all behaviors and values allowed by this standard, including behaviors that are unspecified or implementation defined. Such applications are permitted to adapt to the availability of facilities whose availability is indicated by the values from the packages `POSIX`, `POSIX_Limits`, `POSIX_Options`, `POSIX_Configurable_File_Limits`, and `POSIX_Configurable_System_Limits`.

1.3.2.2 Conforming POSIX.5 Application

There are three categories of *Conforming POSIX.5 Application*s:

1.3.2.2.1 ISO/IEC Conforming POSIX.5 Application

An ISO/IEC Conforming POSIX.5 Application is an application that uses only the facilities described in this standard and approved conforming language bindings for any ISO or IEC standard. Such an application shall include a statement of conformance that documents all options and limit dependencies and all other ISO or IEC standards used.

1.3.2.2.2 <National Body> Conforming POSIX.5 Application

A <National Body> Conforming POSIX.5 Application differs from an ISO/IEC Conforming POSIX.5 Application in that it also may use specific standards of a single ISO/IEC member body, referred to here as *<National Body>*. Such an application shall include a statement of conformance that documents all options and limit dependencies and all other <National Body> standards used.

1.3.2.2.3 Conforming POSIX.5 Application Using Extensions

A Conforming POSIX.5 Application Using Extensions is an application that differs from other Conforming POSIX.5 Applications only in that it uses nonstandard facilities that are consistent with this standard. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a Conforming POSIX.5 Application. A Conforming POSIX.5 Application Using Extensions shall be either an ISO/IEC Conforming POSIX.5 Application Using Extensions or a <National Body> Conforming POSIX.5 Application Using Extensions. (See 1.3.2.2.1 and 1.3.2.2.2.)

Section 2: Terminology and General Requirements

2.1 Editorial Conventions

Text with no marginal bars is carried over unchanged from POSIX.5b.

Text with grey marginal bars tagged with the letter “c” is new or substantially changed between POSIX.5b and POSIX.5c. c

A summary of typographical conventions followed by this standard is shown in Table 2.1.

Table 2.1 – Typographical Conventions

Reference	Example
Ada Identifier	`POSIX_String_List`
Ada Implementation Defined Element	*implementation-defined*
Ada Reserved Word	**`generic`**
Bibliographic Citation	{B1}
C-Language Data Type	*long*
C-Language Error Number	[EINTR]
C-Language Function	*system()*
C-Language Global External	*errno*
C-Language Header	`<sys/stat.h>`
Citation of Normative Reference	ISO/IEC 8602 {9}
Conceptual Attribute	Message Length
Cross-Reference: Section	Section 2
Cross-Reference: Annex	Annex C
Cross-Reference: Clause	2.1
Cross-Reference: Subclause	2.1.2, 2.1.2.3, *etc.*
Environment Variable	**PATH**
Figure Reference	Figure 7-3
File Name	`/tmp`
Implementation Limit	Realtime Signals Maximum
Implementation Option	Job Control option
Table Reference	Table 6-1

Long Ada identifiers are occasionally hyphenated for typographic purposes only; none of the identifiers in this standard have names that include a hyphen.

Qualified name notation, with the package name, is generally used for names that are not defined within the section where the name is used. The exceptions to this rule are names that are defined by the Ada language and certain names defined in the package `POSIX` that are used so frequently that the fully qualified form would be merely cumbersome (*e.g.*, `POSIX_String` is used instead of `POSIX.POSIX_String`).

Numbers within braces, such as “{1},” represent citations. If the number within braces appears alone, it corresponds to that of one of the normative references (1.2). A number preceded by a B, such as “{B1}”, corresponds to one of bibliographic entries listed in Annex A. Bibliographic entries are for information only.

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

(1) Terms defined in 2.2.1 and 2.2.2 are expressed as subclause titles. Alternative forms of the terms appear in [brackets].

(2) The initial appearances of other terms, applying to a limited portion of the text, are in *italics*.

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

Filenames and pathnames are shown in `Courier`. When a pathname is shown starting with "**$HOME/**", this indicates the remaining components of the pathname are to be related to the directory named by the **HOME** environment variable.

Defined names that are usually in lowercase are never used at the beginning of a sentence or anywhere else that regular English usage would require capitalization.

Parenthetical expressions within normative text also contain normative information. The general typographic hierarchy of parenthetical expressions is:

{ [( )] }

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

Annexes marked as *normative* are parts of the standard that pose requirements, exactly the same as the numbered sections, but they follow the body of the standard for clarity of exposition. *Informative* annexes are for information only and pose no requirements. The index, and all material preceding page 1 of this standard (the front matter) are also only informative.

Notes that appear in a small italic font have two different meanings, depending on their location:

— When they are within the normal text of the document, they are the same as footnotes—informative, posing no requirements on implementations or applications.

— When they are attached to tables or figures, they are normative, posing requirements.

Text marked as examples (including the use of "*e.g.*") is for information only.

The typographical conventions listed here are for ease of reading only. Any inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

2.2 Definitions

2.2.1 Terminology

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

2.2.1.1 conformance document: A document provided by an implementor that contains implementation details as described in 1.3.1.2.(Quoted from POSIX.1 {2}.)

2.2.1.2 implementation defined: An indication that the implementation shall define and document the requirements for correct program constructs and correct data of a value or behavior. (Quoted from POSIX.1 {2}.)

2.2.1.3 may: An indication of an optional feature.

(1) With respect to implementations, the word *may* is to be interpreted as an optional feature that is not required in this standard but can be provided.

(2) With respect to Strictly Conforming POSIX.5 Applications, the word *may* means that the optional feature shall not be used.

(Paraphrased from POSIX.1 {2}.)

2.2.1.4 obsolescent: An indication that a certain feature may be considered for withdrawal in future revisions of this standard. (Quoted from POSIX.1 {2}.) Use of obsolescent features is deprecated.

NOTE: Obsolescent features are retained for upward compatibility from POSIX.5, or to maintain symmetry with certain aspects of the C-language binding of POSIX.1. Their use in Ada applications is discouraged.

2.2.1.5 protocol-specific: An indication that the implementation, or Strictly Conforming POSIX.5 Applications where appropriate, shall behave in a manner that is specified in D.1 and D.2 .

2.2.1.6 shall: An indication of a requirement on the implementation or on Strictly Conforming POSIX.5 Applications, where appropriate. (Paraphrased from POSIX.1 {2}.)

2.2.1.7 should:

(1) With respect to implementations, an indication of an implementation recommendation, but not a requirement.

(2) With respect to applications, an indication of a recommended programming practice for applications and a requirement for Strictly Conforming POSIX.5 Applications.

(Paraphrased from POSIX.1 {2}.)

2.2.1.8 supported: A condition regarding optional functionality.(Quoted from POSIX.1 {2}.)

Certain functionality in this standard is optional, but the interfaces to that functionality are always required. If the functionality is supported, the interfaces work as specified by this standard (except that they do not raise the exception indicating the unsupported case). If the functionality is not supported, the interface shall raise the exception indicating the unsupported case.

(Paraphrased from POSIX.1 {2}.)

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

Electronically distributed documents for an implementation are considered part of the system documentation. (Quoted from POSIX.1 {2}.)

2.2.1.10 undefined: An indication that this standard imposes no portability requirements on the use by an application of an indeterminate value or the behavior of that application with erroneous program constructs or erroneous data.

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application using such behaviors is using extensions, as defined in 1.3.2.2.3.

(Paraphrased from POSIX.1 {2}.)

2.2.1.11 unspecified: An indication that this standard imposes no portability requirements on applications for correct program constructs or correct data regarding a value or behavior.

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application requiring a specific behavior, rather than tolerating any behavior when using that functionality, is using extensions, as defined in 1.3.2.2.3.

(Paraphrased from POSIX.1 {2}.)

2.2.2 General Terms

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

2.2.2.1 abort deferred operation: An operation that always continues to completion without being affected by an abort. (Paraphrased from 9.8 (5) of the Ada RM {1}.)

Certain operations are required by the Ada language to be abort deferred. It is implementation defined whether other operations defined by this standard are abort deferred.

2.2.2.2 abort completion point: A point at which the execution of an aborted construct must complete. (Quoted from 9.8 (15) of the Ada RM {1}.)

2.2.2.3 abortive release: An abrupt termination of a network connection that may result in the loss of data. c

2.2.2.4 absolute pathname: See the explanation of *pathname resolution* (2.3.11).

2.2.2.5 access mode: A form of access to a file. or an attribute of a file indicating the kind of operations that may be performed on the file. For more detail see the explanation of file access permissions in 2.3.7.

2.2.2.6 Ada I/O: The I/O operations defined in Annex A of the Ada RM {1} and further defined in Section 8 of this standard.

2.2.2.7 address space: The memory locations that can be referenced by a process. (Quoted from POSIX.1 {2}.)

2.2.2.8 ancillary data: Optional, protocol-specific or local-system-specific information. The information can be both local or end-to-end significant. It can be header information or part of the data portion. It can be protocol-specific and implementation- or system-specific.

2.2.2.9 application: A computer program that performs some desired function. For the purpose of standard, an application is a program that uses the interfaces defined in this standard.

2.2.2.10 application entity title: In OSI, a title that unambiguously identifies an application entity. An application entity title is composed of an application process title and an application entity qualifier (ISO/IEC 7498 {B3}).

2.2.2.11 application entity qualifier: In OSI, a component of an application entity title that is unambiguous within the scope of the application process (ISO/IEC 7498 {B3}).

2.2.2.12 application process title: In OSI, a title that unambiguously identifies an application process. An application process title is a single name, which, for convenience, may be structured internally (ISO/IEC 7498 {B3}). c

2.2.2.13 appropriate privileges: An implementation-defined means of associating privileges with an implementation-defined process with regard to the subprogram calls and options defined in this standard that need special privileges.

There may be zero or more such means.

(Paraphrased from POSIX.1 {2}.)

2.2.2.14 arm a timer: To start a timer measuring the passage of time, enabling the notification of a process when the specified time or time interval has passed. (Quoted from POSIX.1 {2}.)

2.2.2.15 asynchronous events: Events that occur independently of the execution of the application. c

2.2.2.16 asynchronous I/O [AIO] operation: An I/O operation that does not of itself cause the task requesting the I/O to be blocked. An asynchronous I/O operation and the requesting task may be running concurrently. (Paraphrased from POSIX.1 {2}.)

2.2.2.17 asynchronous I/O [AIO] completion: The state of an asynchronous read or write operation when a corresponding synchronous read or write would have completed and any associated status attributes have been updated. (Paraphrased from POSIX.1 {2}.)

2.2.2.18 asynchronously generated signal: An occurrence of a signal that is generated by some mechanism external to the task receiving the signal; for example, via a call to `POSIX_Signals.Send_Signal` by another process or via the keyboard. Being asynchronous is a property of how an occurrence of the signal was generated and not a property of the signal. All signals may be generated asynchronously. The effect of an asynchronously generated occurrence of a *reserved signal* (3.3.1) on an Ada implementation is undefined.

NOTE: Only asynchronously generated signal occurrences are visible to an Ada application as signals. All signal occurrences that are not generated asynchronously are translated into exceptions.

Whether a signal occurrence is generated asynchronously is not to be confused with whether it is delivered or accepted. The difference between delivery and acceptance of a signal is defined in 3.3.1.

2.2.2.19 background process: A process that is a member of a background process group. (Quoted from POSIX.1 {2}.)

2.2.2.20 background process group: Any process group other than a foreground process group that is a member of a session that has established a connection with a controlling terminal. (Quoted from POSIX.1 {2}.)

2.2.2.21 bind: To assign a network address to an endpoint. c

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.23 blocking: Executing with `POSIX_IO.Non_Blocking` not set. See also *nonblocking* (2.2.2.102). c

2.2.2.24 blocking behavior: The effect on other tasks in the same partition when a task is blocked by a POSIX operation. Certain POSIX operations are required to block the calling task under defined conditions. For implementation-defined reasons a blocked task may prevent other tasks from executing. Names for blocking behaviors are defined in 2.4.1.5.

2.2.2.25 blocked task: An Ada task that is not running or ready to run. A task is either blocked or ready to run. While ready, a task competes for the available execution resources that it requires to run. An operation that causes a task to become blocked is said to *block* the task, and an operation that causes a task to no longer be blocked is said to *unblock* the task. (Paraphrased from Section 9 (10) of the Ada RM {1}.)

2.2.2.26 broadcast: The transfer of data from one endpoint to several endpoints, as it is described in RFC 919 {15} and RFC 922 {16}. c

2.2.2.27 byte: The measurement unit for data used in this standard. By common usage the term *byte* usually refers to eight bits of data, but within the context of this standard the size of a byte is implementation defined subject to the constraints given in 2.4.1.3. The size of a byte is given by the constant `POSIX.Byte_Size`.

NOTE: In the context of serial I/O (e.g., see Section 7), transmitting a byte of data may require the transmission of more bits than the size of a byte in memory, since, for example, stop bits and parity bits might be included.

2.2.2.28 canonical input processing: The processing of terminal input in the form of text lines. For more detail see the explanation of canonical mode input processing in secrefCanonical Mode Input Processing.

2.2.2.29 character: A sequence of one or more values of type `POSIX.POSIX_Character`.

NOTE: This definition of the term character applies when it is used by itself. It does not apply to qualified phrases containing the word character, such as POSIX character (2.2.2.126), graphic character (2.2.2.80), Ada character (2.3.3), and character special file (2.2.2.30).

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

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

(Paraphrased from POSIX.1 {2}.)

2.2.2.31 character special file for use with XTI calls: A file of a particular type that is used for process-to-process communication as described in Section 17 of this standard. A character special file for use with XTI calls corresponds to a communications endpoint that uses a specified family of communications protocols. C

2.2.2.32 child process: See *POSIX process* (2.2.2.128).

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.34 communication provider: A component of the system that provides the communications service through an endpoint.

2.2.2.35 communications endpoint: See *endpoint* (2.2.2.56).

2.2.2.36 communications user: An application that uses process-to-process communication services. C

2.2.2.37 completion of a call: The execution of a construct or entity is complete when the end of that execution has been reached, or when a transfer of control causes it to be abandoned. Completion due to reaching the end of execution, or due to the transfer of control of an **`exit`**, **`return`**, **`goto`**, **`requeue`**, or of the selection of a **`terminate`** alternative is normal completion. Completion is abnormal when control is transferred out of a construct due to abort or the raising of an exception. (Quoted from 7.6.1(2) of the Ada RM {1}.)

2.2.2.38 condition variable: A synchronization object that allows a task to become blocked until it is unblocked by some event. The unblocking may occur spontaneously or as a result of a timeout or another task performing a condition-signalling operation on the condition variable. In use, condition variables are always associated with mutexes. See also 2.2.2.98.

2.2.2.39 connection: An association established between two or more endpoints for the transfer of data.[7][8]

2.2.2.40 connection mode: The transfer of data in the context of a connection.[8] See also *connectionless mode* (2.2.2.41).

2.2.2.41 connectionless mode: The transfer of data other than in the context of a connection.[8] See also *connection mode* (2.2.2.40) and *datagram* (2.2.2.45). c

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

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

(Quoted from POSIX.1 {2}.)

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.44 current working directory: See *working directory* (2.2.2.198).

2.2.2.45 datagram: A unit of data transferred from one endpoint to another in connectionless mode service. c

2.2.2.46 device: A computer peripheral or an object that appears to the application as such. (Quoted from POSIX.1 {2}.)

7) Connection establishment generally involves communication between the endpoints.

8) Definition is based on ISO/IEC 7498-1ISO/IEC 7498 {B3}.

2.2.2.47 directory: A file that contains directory entries.

No two entries in a directory shall have the same filename.

(Quoted from POSIX.1 {2}.)

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

Several directory entries can associate different filenames with the same file.

(Quoted from POSIX.1 {2}.)

2.2.2.49 disarm a timer: To stop a timer from measuring the passage of time, thereby disabling any future process notifications (until the timer is armed again). (Quoted from POSIX.1 {2}.)

2.2.2.50 dot: The filename consisting of a single dot character (.). (Quoted from POSIX.1 {2}.)

For the significance of dot see the explanation of pathname resolution in 2.3.11.

2.2.2.51 dot-dot: The filename consisting of (. .). (Quoted from POSIX.1 {2}.)

For the significance of dot-dot see the explanation of pathname resolution in 2.3.11.

2.2.2.52 effective group ID: An attribute of a process that is used in determining various permissions, including the file access permissions described in 2.3.7.

This value is subject to change during the process lifetime, as described in 4.1 and 3.1). See also *group ID* (2.2.2.81).

(Paraphrased from POSIX.1 {2}.)

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

This value is subject to change during the process lifetime, as described in 4.1 and 3.1). See also *user ID* (2.2.2.196).

(Paraphrased from POSIX.1 {2}.)

2.2.2.54 empty directory: A directory that contains, at most, entries for dot and dot-dot. (Quoted from POSIX.1 {2}.)

2.2.2.55 empty string [null string]: A zero-length array whose components are of some character type.

2.2.2.56 endpoint: An object that is created and maintained by a communications provider and used by applications for sending and receiving data; endpoints are used by the communications providers to identify the sources and destinations of data.

c

2.2.2.57 environment task: The anonymous task whose execution elaborates the library items of the declarative part of an active partition, and then calls the main subprogram, if there is one. (Paraphrased from 10.2 (8) of the Ada RM {1}.)

2.2.2.58 Epoch: A base reference time defined as 0 hours, 0 minutes, 0.0 seconds, 1 January 1970, Universal Coordinated Time (see Blair {B2}). (Paraphrased from POSIX.1 {2}.)

2.2.2.59 erroneous execution: In this standard as defined in 1.1.5 of the Ada RM {1}.

2.2.2.60 error code of a task: An attribute of a task that ordinarily specifies information about the most recent error that caused `POSIX_Error` to be raised. (See 2.4.6.)

2.2.2.61 event management: The mechanism that enables applications to register for and be made aware of external events such as data becoming available for reading. c

2.2.2.62 `Exec` family of operations: The collection of operations that cause a new program to be executed, *i.e.*, the `Exec` and `Exec_Search` procedures in `POSIX_Unsafe_Process_Primitives` and the `Start_Process` and `Start_Process_Search` procedures in `POSIX_Process_Primitives`.

2.2.2.63 FIFO special file [FIFO]: A type of file with the property that data written to such a file is read on a first-in-first-out basis. (Quoted from POSIX.1 {2}.)

Other characteristics of FIFOs are described in 5.2 and 6.1.

2.2.2.64 file: An object that can be written to, or read from, or both. (Quoted from POSIX.1 {2}.)

A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, socket, character special file for use with XTI calls, and directory. Other types of files may be defined by the implementation. (Paraphrased from POSIX.1 {2}.) c

2.2.2.65 file description: See *open file description* (2.2.2.108).

2.2.2.66 file descriptor: A per-process unique nonnegative integer value used to identify an open file for the purpose of file access. (Quoted from POSIX.1 {2}.)

2.2.2.67 file group class: A property of a file indicating access permissions for a process related to the group identification of the process.

A process is in the file group class of a file if the process is not in the file owner class and if the effective group ID or one of the supplementary group IDs of the process matches the group ID associated with the file. Other members of the class may be implementation defined. (Paraphrased from POSIX.1 {2}.)

2.2.2.68 filename: A nonempty string that is used to name a file.

A filename consists of, at most, `POSIX_Limits.Filename_Maxima'Last` components of type `POSIX.POSIX_Character`. The characters composing the name may be selected from the set of all the character values excluding the slash character and the null character. The filenames dot and dot-dot have special meaning, as specified in the explanation of *pathname resolution* (2.3.11). A filename is sometimes referred to as a pathname component.

(Paraphrased from POSIX.1 {2}.)

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.70 file other class: A property of a file indicating access permissions for a process related to the user and group information of the process.

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

(Quoted from POSIX.1 {2}.)

2.2.2.71 file owner class: A property of a file indicating access permissions for a process related to the user identification of the process.

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

(Quoted from POSIX.1 {2}.)

2.2.2.72 file permission: Information about a file that is used, along with other information, to determine whether a process has read, write, or execute/search permission to a file. (Quoted from POSIX.1 {2}.)

The file permission information is divided into three parts: owner, group, and other. Each part is used with the corresponding file class of processes. These permissions are contained in the *file mode*, as described in 5.1. The detailed usage of the file permission information in access decisions is described in *file access permissions* in 2.3.7. (Paraphrased from POSIX.1 {2}.)

2.2.2.73 file serial number: A per-file-system unique value for a file.

File serial numbers are unique throughout a file system.

(Quoted from POSIX.1 {2}.)

2.2.2.74 file system: A collection of files, together with certain of their attributes. (Quoted from POSIX.1 {2}.)

Each file system provides a separate binding of file serial numbers to files. A given file serial number is associated with at most one file in a file system, but it may refer to distinct files in distinct file systems. In other words, each file system defines a new *name space*, giving meaning to the *names* (file serial numbers) that designate files.

2.2.2.75 first open, of a file: The act when a process opens a file, message queue, or shared memory object that is not currently open within any process. (Paraphrased from POSIX.1 {2}.)

2.2.2.76 flow control: The mechanism employed by a communications provider that constrains a sending entity to wait until the receiving entities can safely receive additional data without loss. c

2.2.2.77 foreground process: A process that is a member of a foreground process group. (Quoted from POSIX.1 {2}.)

2.2.2.78 foreground process group: A group of processes that have certain privileges, denied to processes in background process groups, when accessing their controlling terminal.

Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. See 7.1.0.7.

(Quoted from POSIX.1 {2}.)

2.2.2.79 foreground process group ID: The process group ID of the foreground process group. (Quoted from POSIX.1 {2}.)

2.2.2.80 graphic character: A sequence of one or more `POSIX.POSIX_Characters` representing a single graphic symbol.

2.2.2.81 group ID: A value identifying a group of system users.

Each system user is a member of at least one group. A group ID is defined in the package `POSIX_Process_Identification` (4.1). When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the (optional) supplementary group IDs, or an (optional) saved set-group-ID.

(Quoted from POSIX.1 {2}.)

2.2.2.82 host byte order: The native representation of an integer: unsigned integer m is the representation in *host byte order* of bit string $b_n b_{n-1} \ldots b_0$ (where b_n is the *most significant*, or *highest order* bit, and b_0 is the *least significant* or *lowest order* bit) if $m = 2^n * b_n + 2^{n-1} * b_{n-1} + \cdots + 2^0 * b_0$. c

2.2.2.83 job control: A facility that allows users to stop (suspend) selectively the execution of processes and continue (resume) their execution at a later time. (Quoted from POSIX.1 {2}.)

The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. Conforming implementations may optionally support job control facilities. The presence of this option is indicated to the application at compile time by the subtype `Job_Control_Support` in package `POSIX_Options` or at run time by the value returned by the function `Job_Control_Is_Supported` in package `POSIX_Configurable_System_Limits`; see 2.4 and 4.5. (Paraphrased from POSIX.1 {2}.)

2.2.2.84 last close: The act of a process closing a file, message queue, or shared memory object that results in the file, message queue, or shared memory object no longer being open within any process. (Paraphrased from POSIX.1 {2}.)

2.2.2.85 linger: To wait for a period of time before terminating a connection to allow outstanding data to be transferred. c

2.2.2.86 link: See *directory entry* (2.2.2.48).

2.2.2.87 link count: The number of directory entries that refer to a particular file. (Quoted from POSIX.1 {2}.)

2.2.2.88 local interprocess communication [local IPC]: The transfer of data between processes in the same system. c

2.2.2.89 login: The unspecified (by this standard) activity by which a user gains access to the system. (Paraphrased from POSIX.1 {2}.)

Each login shall be associated with exactly one login name.

(Paraphrased from POSIX.1 {2}.)

2.2.2.90 login name: A user name that is associated with a login. (Quoted from POSIX.1 {2}.)

2.2.2.91 map a range of addresses: To create an association process's address space and a range of physical memory or some memory object, such that a reference to an address in that range of the address space results in a reference to the associated physical memory or memory object.

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

(Quoted from POSIX.1 {2}.)

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

When used in conjunction with `Map_Memory`, `Open_And_Map_Shared_Memory`, or `Open_Or_Create_And_Map_Shared_Memory`, a memory object will appear in the address space of the calling process.

(Paraphrased from POSIX.1 {2}.)

2.2.2.93 memory-resident: Managed by the implementation in such a way as to provide an upper bound on memory access times. (Quoted from POSIX.1 {2}.)

2.2.2.94 message: Information that can be transferred among tasks (possibly in different processes) by being added to and removed from a message queue.

A message queue consists of a fixed-size buffer.

(Quoted from POSIX.1 {2}.)

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.96 message queue descriptor: A per-process unique value used to identify an open message queue.

2.2.2.97 mode: A collection of attributes that specifies the type of a file and its access permissions. (Quoted from POSIX.1 {2}.)

For more detail see the explanation of file access permissions in 2.3.7.

2.2.2.98 mutex: A synchronization object used to allow multiple tasks (possibly in different processes) to serialize their access to shared data or other shared resources.

The name derives from the capability it provides, namely, mutual exclusion.

(Paraphrased from POSIX.1 {2}.)

2.2.2.99 mutex owner: The task that last locked a mutex, until that same task unlocks the mutex.

2.2.2.100 network address: A network-visible identifier used to designate specific endpoints in a network. Specific endpoints on host systems shall have addresses, and host systems may also have addresses.

2.2.2.101 network byte order: An implementation-defined way of representing an integer so that, when transmitted over a network via a network endpoint, the integer shall be transmitted as an appropriate number of octets with the most significant octet first. This term has been used in historical systems and base documents to denote this representation. This representation is used in many networking protocols, including Internet and OSI. However, it should not be assumed this representation is useful with all network protocols.

2.2.2.102 nonblocking: Executing with `POSIX_IO.Non_Blocking` set. When executing with `POSIX_IO.Non_Blocking` set, functions do not wait for protocol events (*e.g.*, acknowledgments) to occur before returning control. See also *blocking* (2.2.2.23). c

2.2.2.103 noncanonical input processing: The processing of terminal input as uninterpreted characters. (See Section 7.)

2.2.2.104 null character: The value `POSIX_Character'Value(0)`, if defined. The use of a null character in filenames or environment variable names or values produces undefined results. Implementations shall not return null characters to applications in filenames or environment variable names or values.

2.2.2.105 null string: See *empty string* (2.2.2.55).

2.2.2.106 octet: Unit of data representation that consists of eight contiguous bits. |c

2.2.2.107 open file: A file that is currently associated with a file descriptor. (Quoted from POSIX.1 {2}.)

2.2.2.108 open file description: A record of how a process or group of processes is accessing a file.

Each file descriptor shall refer to exactly one open file description, but an open file description may be referred to by more than one file descriptor. A file offset, file status (see 6.1.7), and file access modes are attributes of an open file description.

(Quoted from POSIX.1 {2}.)

2.2.2.109 orderly release: The graceful termination of a network connection with no loss of data. |c

2.2.2.110 orphaned process group: A process group in which the parent of every member is either itself a member of the group or is not a member of the session of the group. (Paraphrased from POSIX.1 {2}.)

An orphaned process group is no longer a member of the session of the process that created it. A process group can become orphaned when all other members of the session exit.

2.2.2.111 page: The granularity of memory mapping and locking, *i.e.*, a fixed-length contiguous range of the address space of a process. Physical memory and memory objects can be mapped into the address space of a process on page boundaries and in integral multiples of pages. Process address space can be locked into memory (*i.e.*, made memory-resident) on page boundaries and in integral multiples of pages. (Paraphrased from POSIX.1 {2}.)

NOTE: There is no implied requirement that usage of the term page in this interface for memory mapping necessarily be the same as the term might be used in a virtual memory implementation.

2.2.2.112 page size: The number of storage units in a page.

NOTE: Unlike POSIX.1, in this standard the page size is given in storage units rather than bytes. Storage units are used in this standard for consistency with the units used in the address arithmetic provided by `System.Storage_Elements`, in case the system storage unit is not one byte.

2.2.2.113 parent directory:

(1) When discussing a given directory, the directory that contains a directory entry for the given directory. The parent directory is represented by the pathname dot-dot in the given directory.

(2) When discussing other types of files, a directory containing a directory entry for the file under discussion.

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

(Quoted from POSIX.1 {2}.)

2.2.2.114 parent process: See *POSIX process* (2.2.2.128).

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.116 partition: A partition is a program or part of a program that can be invoked from outside the Ada implementation. Each partition may run in a separate address space, possibly on a separate computer. An *active partition* is a partition that contains at least one task. Every active partition has an *environment task*, on which all the other tasks of that partition depend. (Paraphrased from 10.2 (2) of the Ada RM {1}.) An active partition corresponds to a POSIX process.

2.2.2.117 path prefix: A pathname, with an optional ending slash, that refers to a directory. (Quoted from POSIX.1 {2}.)

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

A pathname consists of, at most, `POSIX_Limits.Pathname_Maxima'Last` components of type `POSIX.POSIX_Character`. It has an optional beginning slash followed by zero or more filenames separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered the same as one slash. A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated exactly the same as a single slash. The interpretation of the pathname is explained in 2.3.11.

(Paraphrased from POSIX.1 {2}.)

2.2.2.119 pathname component: See *filename* (2.2.2.68).

2.2.2.120 permission: See *file permission* (2.2.2.72).

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

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

(Paraphrased from POSIX.1 {2}.)

2.2.2.122 pipe: An object accessed by one of the pair of file descriptors created by the `POSIX_IO.Create_Pipe` procedure.

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

(Paraphrased from POSIX.1 {2}.)

2.2.2.123 polling: A scheduling scheme whereby the local process periodically checks until the prespecified events (*e.g.*, read, write) have occurred. c

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

For a filename to be portable across conforming implementations of this standard, it shall consist only of the following characters:

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

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

(Paraphrased from POSIX.1 {2}.)

2.2.2.125 portable pathname character set: The set of characters from which portable pathnames are constructed. The set contains all the characters of the portable filename set, plus the character slash (/).

2.2.2.126 POSIX character: A value of the type `POSIX_Character`. An array of POSIX characters, of type `POSIX_String` is called a *POSIX string*.

2.2.2.127 POSIX I/O: The I/O operations defined by this standard, except for Section 8.

2.2.2.128 POSIX process: A conceptual object, having an associated address space, one or more threads of control executing within that address space, a collection of system resources required for execution, and certain other attributes. A

POSIX process is said to perform an action if any of the conceptual threads of control within it performs the action. See also 2.3.1.

Many of the *system resources* defined by this standard are shared among all of the threads of control within a process. These resources include (but are not limited to) the process ID; the parent process ID; process group ID; session membership; real, effective, and saved set-user-ID; real, effective, and saved set-group-ID; supplementary group IDs; current working directory; root directory; file mode creation mask; file descriptors; timers; file locks; mutexes; environment variables; process argument list; and possibly, a process signal mask. For more detail on the POSIX signal model see 3.3.1.

A process is created by another process with procedures `POSIX_Process_Primitives.Start_Process`, `POSIX_Process_Primitives.Start_Process_Search`, or the function `POSIX_Unsafe_Process_Primitives.Fork`. The process that issues `Start_Process`, `Start_Process_Search`, or `Fork` is known as the parent process. The newly created process is the child process.

2.2.2.129 potentially blocking operation: An operation that is not allowed within a protected action, because it may be required to block the calling task. (Paraphrased from 9.5.1 (8) of the Ada RM {1}.) Certain operations are defined by the Ada language to be potentially blocking. In addition, every operation defined by this standard whose effect is defined as blocking the calling task under any circumstance is also a potentially blocking operation.

2.2.2.130 priority: The general term for an integer-valued attribute of processes, tasks, messages, and asynchronous I/O operations, whose value is used in selecting among entities of the same kind. Numerically higher values represent higher priorities and are given preference for selection over lower priorities.

The *priority of an Ada task* is an integer that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The *base priority* of a task is the priority with which it was created, or to which it was later set by `Dynamic_Priorities.Set_Priority` (defined in D.5 of the Ada RM {1}). At all times, a task also has an *active priority*, which generally reflects its base priority as well as any priority it inherits from other sources. *Priority inheritance* is the process by which the priority of a task or other entity (*e.g.*, a protected object, defined in D.3 of the Ada RM {1}) is used in the evaluation of the active priority of another task. At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. (Paraphrased from D.1 (20) of the Ada RM {1}.)

2.2.2.131 privilege: See *appropriate privileges* (2.2.2.13).

2.2.2.132 process: See *POSIX process* (2.2.2.128).

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.134 process group ID: A unique value identifying a process group during its lifetime.

A process group ID shall not be reused by the system until the process group lifetime ends.

(Quoted from POSIX.1 {2}.)

2.2.2.135 process group leader: The unique process, within a process group, that created the process group. (Quoted from POSIX.1 {2}.)

2.2.2.136 process group lifetime: A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, due either to the end of the process lifetime of the last process or to the last remaining process calling the `Set_Process_Group_ID` procedure. (Paraphrased from POSIX.1 {2}.)

2.2.2.137 process ID: A unique value identifying a process during its lifetime.

The process ID is a value of the type `Process_ID` defined in the package `POSIX_Process_Identification` (4.1). A process ID shall not be reused by the system until the process lifetime ends. In addition, if a process group exists where the process ID of the process group leader is equal to that process ID, that process ID shall not be reused by the system until the process group lifetime ends.

An implementation shall reserve a value of process ID for use by system processes. A process that is not a system process shall not have this process ID.

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

After a process is created, it is considered active. Its threads of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a `Wait_For_Child_Process` procedure for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

(Paraphrased from POSIX.1 {2}.)

2.2.2.139 process-to-process communication: The transfer of data between processes.

2.2.2.140 program: A set of partitions, which can execute in parallel with one another, possibly in a separate address space and possibly on a separate computer. (Paraphrased from 10.2 (1) of the Ada RM {1}.)

2.2.2.141 protocol: A set of semantic and syntactic rules for exchanging information.

2.2.2.142 protocol independent interface: An interface that enables the application to be insulated from the specifics of the underlying protocol stack which provides the communication services. Protocol independent interfaces allow the application to be written so that it can be ported to various protocol stacks.

2.2.2.143 protocol engine: A component of the DNI implementation model that is a conceptual machine that implements a particular communications protocol profile.

2.2.2.144 protocol event: In the DNI implementation model, an event that is generated by a protocol engine and queued for attention by the event handler.

2.2.2.145 protocol profile: A set of one or more protocol definitions and, where applicable, the identification of chosen classes, subsets, option and parameters of those definitions, necessary for accomplishing a particular function. In the context of this standard, a protocol profile can be thought of as a vertical slice through a layered set of communications protocols. c

2.2.2.146 read-only file system: A file system that has implementation-defined characteristics restricting modifications. (Quoted from POSIX.1 {2}.)

2.2.2.147 ready task: A task that is not blocked. The ready tasks include those that are running as well as those that are waiting for a processor. See also *blocked task* (2.2.2.25).

NOTE: For consistency, this standard uses the definition of ready from the Ada RM {1}, which includes running tasks among those that are ready. Thus the meaning of ready is close to the meaning of runnable as used in POSIX.1, but the two terms differ in that ready includes running threads. This difference in terminology requires differences in several descriptions between this Ada binding and the base POSIX standards to preserve compatible semantics.

2.2.2.148 real group ID: The attribute of a process that, at the time of process creation, identifies the group of the user who created the process. (Quoted from POSIX.1 {2}.)

This value is subject to change during the process lifetime, as described in 4.1. See also *group ID* (2.2.2.81).

2.2.2.149 real user ID: The attribute of a process that, at the time of process creation, identifies the user who created the process. (Quoted from POSIX.1 {2}.)

This value is subject to change during the process lifetime, as described in 4.1. See also *user ID* (2.2.2.196).

2.2.2.150 record: A collection of related data units or words that itself is treated as a unit. c

2.2.2.151 referenced shared memory object: A shared memory object that is open or has one or more mappings defined on it. (Quoted from POSIX.1 {2}.)

2.2.2.152 region:

(1) As relates to the address space of a process, sequence of addresses.

(2) As relates to a file, a sequence of offsets.

(Quoted from POSIX.1 {2}.)

2.2.2.153 regular file: A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. (Quoted from POSIX.1 {2}.)

2.2.2.154 relative pathname: See the explanation of *pathname resolution* (2.3.11).

2.2.2.155 reserved signal: Signals that the application cannot accept and for which the application cannot modify the signal action or masking because the signals are reserved for use by the Ada language implementation. The reserved signals defined by this standard are `Signal_Abort`, `Signal_Alarm`, `Signal_Floating_Point_Error`, `Signal_Illegal_Instruction`, `Signal_Segmentation_Violation`, `Signal_Bus_Error`. If the implementation supports any signals besides those defined by this standard, the implementation may also reserve some of those.

2.2.2.156 resolution [time]: The minimum time interval that a clock can measure or whose passage a timer can detect. (Quoted from POSIX.1 {2}.)

2.2.2.157 root directory: A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash. (Quoted from POSIX.1 {2}.)

2.2.2.158 running task: The task currently being executed by a processor.

2.2.2.159 saved set-group-ID: When the Saved IDs option is implemented, an attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described for `Set_Group_ID`. For more detail see 2.3.7 and Section 4. (Paraphrased from POSIX.1 {2}.)

2.2.2.160 saved set-user-ID: When the Saved IDs option is implemented, an attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described for `Set_User_ID`. For more detail see 2.3.7 and Section 4. (Paraphrased from POSIX.1 {2}.)

2.2.2.161 semaphore: A shareable resource that has a nonnegative integer value. When the value is zero, is a (possibly empty) set of tasks is awaiting the availability of the semaphore.

2.2.2.162 semaphore decrement operation: An operation that decrements the value of a semaphore, blocking until this is possible.

If, prior to the operation, the value of the semaphore is zero, the semaphore decrement operation shall cause the calling task to be blocked and added to the set of tasks (possibly in different processes) awaiting the semaphore. Otherwise, the semaphore value is decremented. For more detail see 11.1.7.

2.2.2.163 semaphore increment operation: An operation that increments the value of a semaphore or unblocks a waiting task.

If, prior to the operation, any tasks are awaiting the semaphore, then some task from that set shall be removed from the set and be unblocked. Otherwise, the semaphore value shall be incremented. For more detail see 11.1.8. indexx[blocking]semaphore

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

Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership by the procedure `Create_Session` in the package `POSIX_Process_Identification`. Implementations that support `Set_Process_Group_ID` can have multiple process groups in the same session.

(Paraphrased from POSIX.1 {2}.)

2.2.2.165 session leader: A process that has created a session. (Quoted from POSIX.1 {2}.) For more detail on the role of session leader see 4.1.2.

2.2.2.166 session lifetime: The period between when a session is created and the end of the lifetime of all the process groups that remain as members of the session. (Quoted from POSIX.1 {2}.)

2.2.2.167 shared memory object: An object that represents memory that can be mapped concurrently into the address space of more than one process. (Quoted from POSIX.1 {2}.)

These named regions of storage may be independent of the file system and can be mapped into the address space of one or more processes to allow them to share the associated memory.

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.169 signal-driven mode: A mode of operation in which the signal `POSIX_Signals.Signal_IO` is sent to the owner of a socket whenever an I/O operation becomes possible on that socket. In this mode, `POSIX_Signals.Signal_IO` is sent when additional data could be sent on the socket, when new data arrives to be received on a socket, or a state transition occurs that would allow a send or receive call to return status without blocking. Signal-driven mode is enabled by setting the `POSIX_IO.Signal_When_Socket_Ready` flag on the socket and disabled by resetting the `POSIX_IO.Signal_When_Socket_Ready` flag. The default mode for signal driven mode is disabled. c

2.2.2.170 signal queueing: When queueing is enabled for a signal, occurrences of that signal are queued in FIFO order and information is included if the signal is from a source that supplies information. Otherwise, the signal queue may be only one occurrence deep and it is implementation defined whether the data are included. Support for signal queueing is governed by the Realtime Signals option. Not all signals may support queueing.

2.2.2.171 slash: The literal character `"/"`.

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

(Quoted from POSIX.1 {2}.)

2.2.2.172 socket: A file of a particular type that is used as a communications endpoint for process-to-process communication as described in Section 18.

2.2.2.173 socket address: An address associated with a socket or remote endpoint. The address may include multiple parts, such as a network address associated with a host system and an identifier for a specific endpoint. c

2.2.2.174 storage unit: The length of an addressable element of storage in the machine, measured in bits. (Every storage element has the same size.) This term is used here as defined in 13.7 (31) of the Ada RM {1}, where it is declared as `System.Storage_Unit`.

NOTE: The storage unit is very likely to be one byte, but this is not a requirement. For example, it might be 32 or 64 bits.

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

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

(Quoted from POSIX.1 {2}.)

2.2.2.176 supplementary group ID: An attribute of a process, used in determining file access permissions. (Quoted from POSIX.1 {2}.)

A process has group IDs in addition to the effective group ID. The size of this list of supplementary group IDs is specified at compile time by `Groups_Maxima` in package `POSIX_Limits`, or at run time by the value of the function `Groups_Maximum` in package `POSIX_Configurable_System_Limits`. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created. Whether the effective group ID of a process is included in or omitted from its list of supplementary group IDs is unspecified.

2.2.2.177 synchronized I/O completion: The state of an I/O operation that has either been successfully transferred or diagnosed as unsuccessful. (Quoted from POSIX.1 {2}.)

2.2.2.178 synchronized I/O data integrity completion: A degree of completion for an I/O operation that occurs when:

(1) For read, the operation has been completed or diagnosed as unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting task. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests shall be successfully transferred prior to reading the data.

(2) For write, the operation has been completed or diagnosed as unsuccessful. The write is complete only when the data specified in the write request is successfully transferred, and all file system information required to retrieve the data is successfully transferred.

File attributes that are not necessary for data retrieval (Last Access Time, Last Modification Time, Last Status Change Time) need not be successfully transferred prior to returning to the calling task.

(Paraphrased from POSIX.1 {2}.)

2.2.2.179 synchronized I/O file integrity completion: Identical to a synchronized I/O data integrity completion with the addition that all file attributes relative to the I/O operation (including Last Access Time, Last Modification Time, Last Status Change Time) shall be successfully transferred prior to returning to the calling task. (Paraphrased from POSIX.1 {2}.)

2.2.2.180 synchronized I/O operation: An I/O operation performed on a file that provides the application assurance of the integrity of its data and files. (Quoted from POSIX.1 {2}.) See also *synchronized I/O file integrity completion* (2.2.2.178) and *synchronized I/O data integrity completion* (2.2.2.179).

2.2.2.181 synchronous I/O operation: An I/O operation that causes the task requesting the I/O to be blocked from further use of the processor until that I/O operation completes. (Quoted from POSIX.1 {2}.)

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

2.2.2.182 system: An implementation of this standard. (Paraphrased from POSIX.1 {2}.)

2.2.2.183 system crash: An event initiated by an unspecified circumstance that causes all processes (possibly other than special system processes) to be terminated in an undefined manner, after which any changes to the state and contents of files created or written to by a conforming POSIX.5 application prior to the interval are undefined, except as required elsewhere in this standard. (Paraphrased from POSIX.1 {2}.)

2.2.2.184 system process: An object, other than a process executing an application, that is defined by the system and has a process ID. (Quoted from POSIX.1 {2}.) An implementation shall reserve at least one process ID for system processes.

2.2.2.185 system reboot: An implementation-defined sequence of events that may result in the loss of transitory data, *i.e.*, data that are not saved in permanent storage, including message queues, shared memory, semaphores, and processes. (Paraphrased from POSIX.1 {2}.)

2.2.2.186 task: An Ada object with a thread of control, defined by Section 9 of the Ada RM {1} as follows:

> The execution of an Ada program consists of the execution of one or more tasks. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks.

2.2.2.187 terminal [terminal device]: A character special file that obeys the specifications of 7.1. (Quoted from POSIX.1 {2}.)

2.2.2.188 terminate (a connection): To dissolve an association established between two or more endpoints for the transfer of data. c

2.2.2.189 thread of control: A sequence of instructions executed by a conceptual sequential subprogram, independent of any programming language. More than one thread of control may execute concurrently, interleaved on a single processor, or on separate processors. The conceptual threads of control in an Ada application are Ada tasks. They may, but need not, correspond to the POSIX threads defined in POSIX.1.

2.2.2.190 timeouts: A method of error checking whereby an expected event is tested to occur within a specified period of time. c

2.2.2.191 timer: An object that can notify a process when the time as measured by a particular clock has reached or passed a specified value or when a specified amount of time as measured by a particular clock has passed. (Quoted from POSIX.1 {2}.)

Timers are per process; that is, they cannot be shared between processes.

2.2.2.192 timer overrun: A condition that occurs each time a timer for which there is already an expiration signal queued to the process expires. (Quoted from POSIX.1 {2}.)

2.2.2.193 unbind: To remove the association between a network address and an endpoint.

2.2.2.194 unblocked mode (function): A function that behaves like a blocked function, except that when it returns, the function may be incomplete and the application process may have to invoke the interface again.

2.2.2.195 unit data: See *datagram* (2.2.2.45). c

2.2.2.196 user ID: A value identifying a system user.

A User ID is a value of the type `User_ID` defined in the package `POSIX_Process_Identification`. When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or an (optional) saved set-user-ID.

2.2.2.197 user name: A value of `POSIX_String` that is used to identify a user, as described in 9.1.

2.2.2.198 working directory [current working directory]: A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash. (Quoted from POSIX.1 {2}.)

2.2.3 Abbreviations

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

2.2.3.1 ACSE: Association Control Service Element c

2.2.3.2 Ada 83: The original Ada language standard, approved by ANSI in 1983 and by ISO/IEC in 1987 as ISO/IEC 8652 {B5}.

2.2.3.3 Ada 95: The 1995 Ada language standard, approved as ISO/IEC 8652 {1}, used in contrast to Ada 83.

2.2.3.4 Ada RM: The Ada language standard, ISO/IEC 8652: 1995(E) {1}, also known as the Ada Reference Manual.

2.2.3.5 AE: Application Entity, in the context of protocol mappings (Annex D).

2.2.3.6 AP: Application Process, in the context of protocol mappings (Annex D).

2.2.3.7 API: Application Program Interface

2.2.3.8 AARE: A-ASSOCIATE Response (ISO/IEC ISP 11188-3 {11})

2.2.3.9 AIO: Asynchronous Input and Output

2.2.3.10 AK TPDU: Acknowledge Transport Protocol Data Unit (ISO/IEC 8073 {4}) c

2.2.3.11 APDU: Application Protocol Data Unit (ISO/IEC ISP 11188-3 {11})

2.2.3.12 ASN.1: Abstract Syntax Notation One - ISO/IEC 8824-1: 1995, Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.

2.2.3.13 BER: Basic Encoding Rules for ASN.1 - ISO/IEC 8825:1990, Abstract Syntax Notation One (ASN.1): Basic Encoding Rules.

2.2.3.14 BSD: Berkeley Software Distribution

2.2.3.15 CC TPDU: Connection Confirm Transport Protocol Data Unit (ISO/IEC 8073 {4})

2.2.3.16 CLNP: Connectionless-Mode Network Protocol (see ISO/IEC 8473-1 {7})

2.2.3.17 CLTP: Connectionless-Mode Transport Protocol (see ISO/IEC 8602 {9})

2.2.3.18 CPU: Central Processing Unit

2.2.3.19 CR: Carriage Return (see CRin 7.1.0.12)

2.2.3.20 CR TPDU: Connection Request Transport Protocol Data Unit (ISO/IEC 8073 {4})

2.2.3.21 CULR: Common Upper Layer Requirements (in OSI, ISO/IEC ISP 11188-3 {11})

2.2.3.22 DC TPDU: Disconnection Confirm Transport Protocol Data Unit (ISO/IEC 8073 {4})

2.2.3.23 DCS: Defined Context Set (ISO/IEC 8073 {4})

2.2.3.24 DR TPDU: Disconnection Request Transport Protocol Data Unit (ISO/IEC 8073 {4})

2.2.3.25 DT TPDU: Data Transport Protocol Data Unit (ISO/IEC 8073 {4})

2.2.3.26 DNI: Detailed Network Interface

2.2.3.27 ED TPDU: Expedited Data Transport Protocol Data Unit (ISO/IEC 8073 {4})

2.2.3.28 EM: Event Management

2.2.3.29 EOF: End of File (EOF in 7.1.0.12)

2.2.3.30 EOL: End of Line (EOL in 7.1.0.12)

2.2.3.31 ETSDU: Expedited Transport Service Data Unit

2.2.3.32 FD: File Descriptor (6.1.1)

2.2.3.33 FIFO: First-In-First-Out (5.2.1)

2.2.3.34 FIN: no more data from sender (RFC 793 {14})

NOTE: This value, along with RST, SYN, and URG, represents a control bit in word 4 of the TCP/IP protocol header. This is a case where the API assumes an intimate knowledge of the protocol being programmed. There are no names in the Ada or C-language bindings to sockets for these constructs. This is also the case for much of the terminology used in Annex D for the ISO protocols.

2.2.3.35 FSM: Finite State Machine

2.2.3.36 ICMP: Internet Control Message Protocol – IETF RFC 792: 1981, Internet Control Message Protocol.

2.2.3.37 IETF: Internet Engineering Task Force

2.2.3.38 I/O: Input and Output

2.2.3.39 IP: Internet Protocol (RFC 793 {14})

2.2.3.40 IPC: Interprocess Communication

2.2.3.41 LIS: Language-Independent Specification

2.2.3.42 mOSI: Minimal 7-Layer OSI Stack (ISO/IEC 7498 {B3})

2.2.3.43 NL: New Line (NL in 7.1.0.12)

2.2.3.44 NSAP: Network Service Access Point

2.2.3.45 OOB: Out of Band

2.2.3.46 OSI: Open Systems Interconnection (ISO/IEC 7498 {B3})

2.2.3.47 PASC: Portable Applications Standards Committee

2.2.3.48 PDU: Protocol Data Unit

2.2.3.49 PDV: Presentation Data Value

2.2.3.50 POSIX.1: ISO/IEC 9945-1:1996, the POSIX C-language system API from which this standard is derived {2}.

2.2.3.51 POSIX.1b: IEEE Std 1003.1b-1993, {B12}.

2.2.3.52 POSIX.1c: IEEE Std 1003.1c-1995, {B13}.

2.2.3.53 P1003.1g: IEEE P1003.1g/D6.6, {B14}.

2.2.3.54 POSIX.1i: IEEE Std 1003.1i-1995, {B15}.

2.2.3.55 POSIX.5: IEEE Std 1003.5-1992, {B17}.

2.2.3.56 POSIX.5b: IEEE Std 1003.5b-1996, {B18}.

2.2.3.57 POSIX.13:1998 IEEE Std 1003.13-1998. {B19}.

2.2.3.58 QOS: Quality of Service

2.2.3.59 RFC: Request for Comment

2.2.3.60 RST: Reset The Connection (RFC 793 {14})

NOTE: See the notes for 2.2.3.34, above

.

2.2.3.61 SEDU: Service Expedited Data Unit

2.2.3.62 SDU: Service Data Unit

2.2.3.63 SYN: Synchronize Sequence Numbers RFC 793 {14})

NOTE: See the notes for 2.2.3.34, above

.

2.2.3.64 TCP: Transmission Control Protocol (of the Internet, see RFC 1122 {18})

2.2.3.65 TP: Transport Protocol (ISO/IEC 8073 {4})

2.2.3.66 TPDU: Transport Protocol Data Unit (ISO/IEC 8073 {4})

2.2.3.67 TSAP: Transport Service Access Point

2.2.3.68 TSDU: Transport Service Data Unit

2.2.3.69 UD TPDU: Unit Data Transport Protocol Data Unit (ISO/IEC 8602 {9})

2.2.3.70 UDP: User Datagram Protocol (Internet, RFC 768 {12})

2.2.3.71 URG: Urgent Pointer Field Significant (RFC 793 {14})

NOTE: See the notes for 2.2.3.34, above

.

2.2.3.72 XTI: X/Open Transport Interface

c

2.3 General Concepts

2.3.1 Process/Active Partition Relationship

An Ada active partition in execution shall behave as if it is a single POSIX process, throughout this standard. Thus, all the tasks of the active partition shall execute within the environment of the same POSIX process and share the same set of process attributes. (See 2.2.2.128.)

NOTE: The active partition in Ada 95 generalizes the concept of a main program execution in Ada 83.

NOTE: While every Ada active partition in a POSIX environment must correspond to some process, the two concepts are not exactly equivalent. In particular, a process may execute a program written in other languages. Moreover, during the lifetime of a process it may execute several different programs (or Ada active partitions), possibly written in different languages, sequentially. (See 3.2.)

2.3.2 Task/Thread Relationship

The only threads of control visible to an Ada application via this standard are tasks; requirements that relate to threads of control are generally described in terms of Ada tasks.

An implementation may optimize away the thread of control of certain tasks (known as *passive task*s) if one of the following applies:

— The implementation takes responsibility for the safety of this optimization. In other words, it is able to determine that it will not violate any requirements of the Ada RM {1}, or change the effect of any operations or pragmas defined by this standard that are supported by the implementation and used within the application.

— The application takes responsibility for the safety of this optimization, requesting it explicitly by implementation-defined means, such as a pragma.

A process may contain threads of control that do not correspond to Ada tasks. Such threads of control may be created invisibly by the implementation of the Ada language or of this standard or via operating system interfaces not defined by this standard. Such other threads of control are not required to behave as Ada tasks. The implementation shall not allow the application to refer to such threads as tasks, via task names or values of type `Ada_Task_Identification.Task_ID`.

NOTE: The tasks of an Ada program may also be implemented using more than one process in the underlying system — as long as all the process attributes specified by this standard (e.g., see 3.1.2, 3.2.2, and Section 4) — are shared by all the tasks in the program, and all tasks in the Ada active partition appear to the other POSIX processes in the system to be a single POSIX process when viewed through the interfaces defined in this standard.

2.3.3 Ada Character Differences

In 3.5.2 of the Ada RM {1} the type `Character` is defined as “a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of

the ISO 10646 Basic Multilingual Plane (BMP)." It also defines the type `Wide_Character`, which "is a character type whose values correspond to the 65536 code positions of the ISO 10646 Basic Multilingual Plane (BMP)."

This standard defines its own character set, `POSIX_Character`, which is neither required nor prohibited from having the same set of values and representation as `Standard.Character` or `Standard.Wide_Character`.

2.3.4 Posix Signals Are Not Interrupts

The application program interfaces for signals defined by this standard are synchronous; that is, a task executes an operation that causes it to block until an occurrence of the signal is available. The preferred interface is via the `Await_Signal` and `Await_Signal_Or_Timeout` operations (see 3.3.15 and 3.3.16). For compatibility with POSIX.5, a signal can also be attached to an entry of a task, via the Ada interrupt-entry mechanism, if the Signal Entries option is supported. (See 3.3.17.)

NOTE: The use of task entries to accept signals is obsolescent.

2.3.5 System Call Exception Errors

Ada supports an exception mechanism that is used in this binding to report errors from system calls. The details of the error are reported by an error code that can be accessed via the function `Get_Error_Code` in package `POSIX`.

2.3.6 Extended Security Controls

The access control (see 2.3.7) and privilege mechanisms (see 2.2.2.13) have been defined to allow implementation-defined *extended security controls*. These controls permit an implementation to provide security mechanisms to implement different security policies from the polices described in this standard. These mechanisms shall be implemented and mapped onto the POSIX interfaces so they appear to be extensions to the POSIX security system.

2.3.7 File Access Permissions

The standard file access control mechanism uses the file permission set, as described in this subclause. This permission set is determined at file creation by the procedures `POSIX_IO.Open_Or_Create`, `POSIX_Sockets.Bind` (for a socket whose protocol family is `POSIX_Sockets_Local.Local_Protocol`), `POSIX_Files.Create_FIFO`, or `POSIX_Files.Create_Directory` and are changed by `POSIX_Files.Change_Permissions`. The permission set can be read by the function `POSIX_File_Status.Permission_Set_Of`. C

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

(1) Specify file permission set for the owner class, group class, and other class of the file, corresponding to the access permissions.

(2) Be enabled only by explicit user action, on a per-file basis by the file owner or a user with the appropriate privilege.

(3) Be disabled for a file after the file permission set is changed for that file. The disabling of the alternate mechanism need not disable any additional mechanisms defined by an implementation.

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

(1) If a process has the appropriate privilege:
   (a) If read, write, or directory search permission is requested, access is granted.
   (b) If execute permission is requested, access is granted if execute permission is granted to at least one user by the file permission set or by an alternate access control mechanism; otherwise, access is denied.

(2) Otherwise:
   (a) The file permission set of a file contains read, write, and execute/search permissions for the file owner class, file group class, and file other class.
   (b) Access is granted if an alternate access control mechanism is not enabled and the requested access permission is set for the class to which the process belongs or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access is denied.

Unless otherwise stated, the user ID is the effective user ID of the process and the group ID is the effective group ID of the process. The effective user ID and the effective group ID can be saved in the saved set-user-ID or saved set-group-ID if the Saved IDs option is supported. (See 4.1.3.)

2.3.8 File Hierarchy

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

2.3.9 Filename Portability

Filenames should be constructed from the portable filename character set because the use of other characters can be confusing or ambiguous in certain contexts. The function `Is_Portable_Filename` (2.4) is provided for checking a `POSIX_String` to verify if it is a portable filename. The function `Is_Portable_Pathname` (2.4) is provided to check for portable pathnames.

2.3.10 File Times Update

Each file has three associated time value attributes. These are the Last Access Time, the Last Modification Time, and the Last Status Change Time. They are updated when file data have been accessed, file data have been modified, or file status has been changed, respectively. They are accessed via functions defined in the package `POSIX_File_Status`.

For each function or procedure in this standard that reads or writes file data or changes the file status, the appropriate attributes are noted as *marked for update*. If an implementation of such a function marks for update a time-related attribute not specified by this standard, what such time-related attributes it updates shall be documented, except that any changes caused by pathname resolution need not be documented. For the other functions in this standard (those that are not explicitly required to read or write file data or change file status, but that in some implementations happen to do so), the effect is unspecified.

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

2.3.11 Pathname Resolution

Pathname resolution is performed for a process to resolve a pathname to a particular file in a file hierarchy. Multiple pathnames may resolve to the same file. (See 5.2.1 and 5.2.3.)

Each filename in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment `a/b`, file `b` is located in directory `a`). Pathname resolution fails if a filename in the pathname cannot be located in the directory specified by its predecessor. If the pathname begins with a slash, the predecessor of the first filename in the pathname is taken to be the root directory of the process (such pathnames are referred to as absolute pathnames). If the pathname does not begin with a slash, the predecessor of the first filename of the pathname is taken to be the current working directory of the process (such pathnames are referred to as relative pathnames). If the pathname begins with two leading slashes, the predecessor of the first filename in the pathname may be interpreted in an implementation-defined manner.

The interpretation of a pathname component is dependent on Filename Limit and whether the Filename Truncation option is supported for the path prefix of the component. If the length in POSIX characters of a pathname component exceeds `Filename_Limit` and the Filename Truncation option is not supported for the pathname prefix, the implementation shall raise the exception `POSIX_Error` and set the error code `Filename_Too_Long`. Otherwise, the implementation shall ignore all the characters after the initial substring whose length in POSIX characters is the value returned by `Filename_Limit` for the pathname component. (See Section 5.)

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

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

2.3.12 Allocated Storage

In some places the POSIX packages may have to allocate storage. One example is string lists. In any place where an access type is declared in the interface or where heap storage is likely to be dynamically allocated, storage retrieval procedures are declared. The storage retrieval procedures allow an application to free any dynamic storage that is likely to be allocated by a POSIX package.

2.4 Package POSIX

This package defines types, constants, and operations that have general applicability across this standard, including the following:

— Specifying a count of I/O units.

— Obtaining information about whether a given operation blocks the entire process or just the calling task.

— Specifying characters, character strings, and lists of character strings.

— Specifying option sets, which are used in other interfaces defined by this standard.

— Obtaining error information when an operation defined by this standard fails.

— Representing time values with nanosecond precision.

— Obtaining information about the specific system on which a process is executing.

— Specifying the class of signals to mask during an interruptible operation.

```
with Ada_Streams,
    Interfaces;                                                          c
package POSIX is
   -- 2.4.1 Constants and Static Subtypes
   -- 2.4.1.1 Version Identification
   POSIX_Version : constant := 1997_XX;
   POSIX_Ada_Version : constant := 1998_YY;                              c
   -- 2.4.1.2 Optional Facilities
   subtype Job_Control_Support is Boolean                    -- obsolescent
      range implementation-defined;                          -- obsolescent
   subtype Saved_IDs_Support is Boolean                      -- obsolescent
      range implementation-defined;                          -- obsolescent
   subtype Change_Owner_Restriction is Boolean               -- obsolescent
      range implementation-defined;                          -- obsolescent
   subtype Filename_Truncation is Boolean                    -- obsolescent
      range implementation-defined;                          -- obsolescent
   -- 2.4.1.3 Bytes and I/O Counts
   Byte_Size : constant := implementation-defined-integer;
   type IO_Count is optional parent type
      range 0 .. implementation-defined;
   subtype IO_Count_Maxima is IO_Count range 32767 .. IO_Count'Last;
   type Octet is mod 2**8;
   type Octet_Array is array (Positive range <>) of Octet;
   function Host_To_Network_Byte_Order (Host_32 : Interfaces.Unsigned_32)
     return Interfaces.Unsigned_32;
   function Host_To_Network_Byte_Order (Host_16 : Interfaces.Unsigned_16)
     return Interfaces.Unsigned_16;                                      c
```

```
function Network_To_Host_Byte_Order (Net_32 : Interfaces.Unsigned_32)
  return Interfaces.Unsigned_32;
function Network_To_Host_Byte_Order (Net_16 : Interfaces.Unsigned_16)
  return Interfaces.Unsigned_16;
-- 2.4.1.4 System Limits
Portable_Groups_Maximum : constant Natural := 0;              -- obsolescent
subtype Groups_Maxima is Natural                              -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Argument_List_Maximum : constant Natural := 4096;    -- obsolescent
subtype Argument_List_Maxima is Natural                       -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Child_Processes_Maximum : constant Natural := 6;     -- obsolescent
subtype Child_Processes_Maxima is Natural                     -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Open_Files_Maximum : constant Natural := 16;         -- obsolescent
subtype Open_Files_Maxima is Natural                          -- obsolescent
   range implementation-defined;                        -- obsolescent
Portable_Stream_Maximum : constant Natural := 8;              -- obsolescent
subtype Stream_Maxima is Natural                              -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Time_Zone_String_Maximum : constant Natural := 3;    -- obsolescent
subtype Time_Zone_String_Maxima is Natural                    -- obsolescent
   range implementation-defined;                              -- obsolescent
-- Pathname Variable Values
Portable_Link_Limit_Maximum : constant Natural := 8;          -- obsolescent
subtype Link_Limit_Maxima is Natural                          -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Input_Line_Limit_Maximum :
   constant IO_Count := 255;                                  -- obsolescent
subtype Input_Line_Limit_Maxima is IO_Count                   -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Input_Queue_Limit_Maximum :
   constant IO_Count := 255;                                  -- obsolescent
subtype Input_Queue_Limit_Maxima is IO_Count                  -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Filename_Limit_Maximum : constant Natural := 14;     -- obsolescent
subtype Filename_Limit_Maxima is Natural                      -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Pathname_Limit_Maximum : constant Natural := 255;    -- obsolescent
subtype Pathname_Limit_Maxima is Natural                      -- obsolescent
   range implementation-defined;                              -- obsolescent
Portable_Pipe_Limit_Maximum : constant IO_Count := 512;       -- obsolescent
subtype Pipe_Limit_Maxima is IO_Count                         -- obsolescent
   range implementation-defined;                              -- obsolescent
-- 2.4.1.5 Blocking Behavior Values
type Blocking_Behavior is (Tasks, Program, Special);
subtype Text_IO_Blocking_Behavior is Blocking_Behavior
   range implementation-defined;
IO_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
File_Lock_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
Wait_For_Child_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
XTI_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
Sockets_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
```

```
subtype Realtime_Blocking_Behavior is Blocking_Behavior
   range implementation-defined;
-- 2.4.1.6 Signal Masking for Interruptible Operations
type Signal_Masking is (No_Signals, RTS_Signals, All_Signals);
-- 2.4.2 POSIX Characters
type POSIX_Character is
   (
   --  ' ','0','1','2','3','4','5','6','7','8','9',
   --  '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',
   --  '.','_','-','/','"','#', '&', ''', '(', ')',
   --  '*', '+', ',', ':', ';', '<', '=', '>', '|'
    other characters are implementation defined);
-- 2.4.3 POSIX Strings
type POSIX_String is array (Positive range <>) of POSIX_Character;
function To_POSIX_String (Str : String) return POSIX_String;
function To_POSIX_String (Str : Wide_String)return POSIX_String;
function To_String (Str : POSIX_String) return String;
function To_Wide_String (Str : POSIX_String) return Wide_String;
function To_Stream_Element_Array (Buffer : POSIX_String)
  return Ada_Streams.Stream_Element_Array;
function To_POSIX_String (Buffer : Ada_Streams.Stream_Element_Array)
  return POSIX_String;
subtype Filename is POSIX_String;
subtype Pathname is POSIX_String;
function Is_Filename (Str : POSIX_String) return Boolean;
function Is_Pathname (Str : POSIX_String) return Boolean;
function Is_Portable_Filename (Str : POSIX_String) return Boolean;
function Is_Portable_Pathname (Str : POSIX_String) return Boolean;
-- 2.4.4 String Lists
type POSIX_String_List is limited private;
Empty_String_List : constant POSIX_String_List;
procedure Make_Empty (List : in out POSIX_String_List);
procedure Append
   (List : in out POSIX_String_List;
    Str :  in POSIX_String);
generic
   with procedure Action
      (Item : in POSIX_String;
       Quit : in out Boolean);
procedure For_Every_Item (List : in POSIX_String_List);
function Length (List : POSIX_String_List) return Natural;
function Value
   (List :  POSIX_String_List;
    Index : Positive)
  return POSIX_String;
-- 2.4.5 Option Sets
type Option_Set is private;
function Empty_Set return Option_Set;
function "+" (L, R : Option_Set) return Option_Set;
function "-" (L, R : Option_Set) return Option_Set;
function "<" (Left, Right : Option_Set) return Boolean;
function "<="(Left, Right : Option_Set) return Boolean;
function ">" (Left, Right : Option_Set) return Boolean;
function ">="(Left, Right : Option_Set) return Boolean;
Option_1 :  constant Option_Set;
```

```
Option_2 :  constant Option_Set;
Option_3 :  constant Option_Set;
Option_4 :  constant Option_Set;
Option_5 :  constant Option_Set;
Option_6 :  constant Option_Set;
Option_7 :  constant Option_Set;
Option_8 :  constant Option_Set;
Option_9 :  constant Option_Set;
Option_10 : constant Option_Set;
Option_11 : constant Option_Set;
Option_12 : constant Option_Set;
Option_13 : constant Option_Set;
Option_14 : constant Option_Set;
Option_15 : constant Option_Set;
Option_16 : constant Option_Set;
Option_17 : constant Option_Set;
Option_18 : constant Option_Set;
Option_19 : constant Option_Set;
Option_20 : constant Option_Set;
Option_21 : constant Option_Set;
Option_22 : constant Option_Set;
Option_23 : constant Option_Set;
Option_24 : constant Option_Set;
Option_25 : constant Option_Set;
Option_26 : constant Option_Set;
Option_27 : constant Option_Set;
Option_28 : constant Option_Set;
Option_29 : constant Option_Set;
Option_30 : constant Option_Set;
Option_31 : constant Option_Set;
-- 2.4.6 Error Codes and Exceptions
POSIX_Error : exception;
type Error_Code is range implementation-defined;
function Get_Error_Code return Error_Code;
procedure Set_Error_Code (Error : in Error_Code);
function Is_POSIX_Error (Error : Error_Code) return Boolean;
function Image (Error : Error_Code) return String;
No_Error : constant Error_Code := 0;
E2BIG,
Argument_List_Too_Long :     constant Error_Code := impl-def-static-expression;
EFAULT,
Bad_Address :                constant Error_Code := impl-def-static-expression;
EBADF,
Bad_File_Descriptor :        constant Error_Code := impl-def-static-expression;
EBADMSG,
Bad_Message :                constant Error_Code := impl-def-static-expression;
EPIPE,
Broken_Pipe :                constant Error_Code := impl-def-static-expression;
ENOTEMPTY,
Directory_Not_Empty :        constant Error_Code := impl-def-static-expression;
ENOEXEC,
Exec_Format_Error :          constant Error_Code := impl-def-static-expression;
EEXIST,
File_Exists :                constant Error_Code := impl-def-static-expression;
EFBIG,
File_Too_Large :             constant Error_Code := impl-def-static-expression;
ENAMETOOLONG,
Filename_Too_Long :          constant Error_Code := impl-def-static-expression;
```

```
EXDEV,
Improper_Link :              constant Error_Code := impl-def-static-expression;
ENOTTY,
Inappropriate_IO_Control_Operation :
                             constant Error_Code := impl-def-static-expression;
EIO,
Input_Output_Error :         constant Error_Code := impl-def-static-expression;
EINTR,
Interrupted_Operation :      constant Error_Code := impl-def-static-expression;
EINVAL,
Invalid_Argument :           constant Error_Code := impl-def-static-expression;
ESPIPE,
Invalid_Seek :               constant Error_Code := impl-def-static-expression;
EISDIR,
Is_A_Directory :             constant Error_Code := impl-def-static-expression;
EMSGSIZE,
Message_Too_Long :           constant Error_Code := impl-def-static-expression;
ECHILD,
No_Child_Process :           constant Error_Code := impl-def-static-expression;
ENOLCK,
No_Locks_Available :         constant Error_Code := impl-def-static-expression;
ENOSPC,
No_Space_Left_On_Device :    constant Error_Code := impl-def-static-expression;
ENODEV,
No_Such_Operation_On_Device :
                             constant Error_Code := impl-def-static-expression;
ENXIO,
No_Such_Device_Or_Address :
                             constant Error_Code := impl-def-static-expression;
ENOENT,
No_Such_File_Or_Directory :
                             constant Error_Code := impl-def-static-expression;
ESRCH,
No_Such_Process :            constant Error_Code := impl-def-static-expression;
ENOTDIR,
Not_A_Directory :            constant Error_Code := impl-def-static-expression;
ENOMEM,
Not_Enough_Space :           constant Error_Code := impl-def-static-expression;
ECANCELED,
Operation_Canceled :         constant Error_Code := impl-def-static-expression;
EINPROGRESS,
Operation_In_Progress :      constant Error_Code := impl-def-static-expression;
ENOSYS,
Operation_Not_Implemented :  constant Error_Code := impl-def-static-expression;
EPERM,
Operation_Not_Permitted :    constant Error_Code := impl-def-static-expression;
ENOTSUP,
Operation_Not_Supported :    constant Error_Code := impl-def-static-expression;
EACCES,
Permission_Denied :          constant Error_Code := impl-def-static-expression;
EROFS,
Read_Only_File_System :      constant Error_Code := impl-def-static-expression;
EBUSY,
Resource_Busy :              constant Error_Code := impl-def-static-expression;
EDEADLK,
Resource_Deadlock_Avoided :  constant Error_Code := impl-def-static-expression;
```

```
EAGAIN,
Resource_Temporarily_Unavailable :
                              constant Error_Code := impl-def-static-expression;
ETIMEDOUT,
Timed_Out :                   constant Error_Code := impl-def-static-expression;
EMLINK,
Too_Many_Links :              constant Error_Code := impl-def-static-expression;
EMFILE,
Too_Many_Open_Files :         constant Error_Code := impl-def-static-expression;
ENFILE,
Too_Many_Open_Files_In_System :
                              constant Error_Code := impl-def-static-expression;
-- Socket Error Codes
EADDRINUSE,
Address_In_Use :              constant Error_Code := impl-def-static-expression;
EADDRNOTAVAIL,
Address_Not_Available :       constant Error_Code := impl-def-static-expression;
EALREADY,
Already_Awaiting_Connection :
                              constant Error_Code := impl-def-static-expression;
ECONNABORTED,
Connection_Aborted :          constant Error_Code := impl-def-static-expression;
ECONNREFUSED,
Connection_Refused :          constant Error_Code := impl-def-static-expression;
ECONNRESET,
Connection_Reset :            constant Error_Code := impl-def-static-expression;
EDOM,
Domain_Error :                constant Error_Code := impl-def-static-expression;
EHOSTDOWN,
Host_Down :                   constant Error_Code := impl-def-static-expression;
EHOSTUNREACH,
Host_Unreachable :            constant Error_Code := impl-def-static-expression;
EAFNOSUPPORT,
Incorrect_Address_Type :      constant Error_Code := impl-def-static-expression;
EISCONN,
Is_Already_Connected :        constant Error_Code := impl-def-static-expression;
ENETDOWN,
Network_Down :                constant Error_Code := impl-def-static-expression;
ENETRESET,
Network_Reset :               constant Error_Code := impl-def-static-expression;
ENETUNREACH,
Network_Unreachable :         constant Error_Code := impl-def-static-expression;
ENOBUFS,
No_Buffer_Space :             constant Error_Code := impl-def-static-expression;
ENOTSOCK,
Not_A_Socket :                constant Error_Code := impl-def-static-expression;
ENOTCONN,
Not_Connected :               constant Error_Code := impl-def-static-expression;
EOPNOTSUPP,
Option_Not_Supported :        constant Error_Code := impl-def-static-expression;
EPROTONOSUPPORT,
Protocol_Not_Supported :      constant Error_Code := impl-def-static-expression;
ESOCKTNOSUPPORT,
Socket_Type_Not_Supported :   constant Error_Code := impl-def-static-expression;
EWOULDBLOCK,
Would_Block :                 constant Error_Code := impl-def-static-expression;
EPROTOTYPE,
Wrong_Protocol_Type :         constant Error_Code := impl-def-static-expression;
```

C

```
-- XTI Error Codes
subtype XTI_Error_Code is Error_Code
   range implementation-defined .. implementation-defined;
TBUFOVFLW,
Buffer_Not_Large_Enough :   constant XTI_Error_Code := impl-def-static-expression;
TPROVMISMATCH,
Communications_Provider_Mismatch :
                            constant XTI_Error_Code := impl-def-static-expression;
TNOADDR,
Could_Not_Allocate_Address :
                            constant XTI_Error_Code := impl-def-static-expression;
TQFULL,
Endpoint_Queue_Full :       constant XTI_Error_Code := impl-def-static-expression;
TBADQLEN,
Endpoint_Queue_Length_Is_Zero :
                            constant XTI_Error_Code := impl-def-static-expression;
TLOOK,
Event_Requires_Attention :  constant XTI_Error_Code := impl-def-static-expression;
TFLOW,
Flow_Control_Error :        constant XTI_Error_Code := impl-def-static-expression;
TBADDATA,
Illegal_Data_Range :        constant XTI_Error_Code := impl-def-static-expression;
TBADADDR,
Incorrect_Address_Format :  constant XTI_Error_Code := impl-def-static-expression;
TBADOPT,
Incorrect_Or_Illegal_Option :
                            constant XTI_Error_Code := impl-def-static-expression;
TRESQLEN,
Incorrect_Surrogate_Queue_Length :
                            constant XTI_Error_Code := impl-def-static-expression;
TACCES,
Insufficient_Permission :   constant XTI_Error_Code := impl-def-static-expression;
TBADNAME,
Invalid_Communications_Provider :
                            constant XTI_Error_Code := impl-def-static-expression;
TBADF,
Invalid_File_Descriptor :   constant XTI_Error_Code := impl-def-static-expression;
TBADFLAG,
Invalid_Flag :              constant XTI_Error_Code := impl-def-static-expression;
TBADSEQ,
Invalid_Sequence_Number :   constant XTI_Error_Code := impl-def-static-expression;
TNODATA,
No_Data_Available :         constant XTI_Error_Code := impl-def-static-expression;
TNODIS,
No_Disconnect_Indication_On_Endpoint :
                            constant XTI_Error_Code := impl-def-static-expression;
TNOREL,
No_Orderly_Release_Indication_On_Endpoint :
                            constant XTI_Error_Code := impl-def-static-expression;
TNOUDERR,
No_Unit_Data_Error_On_Endpoint :
                            constant XTI_Error_Code := impl-def-static-expression;
TOUTSTATE,
Operation_Not_Valid_For_State :
                            constant XTI_Error_Code := impl-def-static-expression;
TINDOUT,
Outstanding_Connection_Indications :
                            constant XTI_Error_Code := impl-def-static-expression;
```

```
TPROTO,
Protocol_Error :            constant XTI_Error_Code := impl-def-static-expression;
TSTATECHNG,
State_Change_In_Progress :  constant XTI_Error_Code := impl-def-static-expression;
TRESADDR,
Surrogate_File_Descriptor_Mismatch :
                            constant XTI_Error_Code := impl-def-static-expression;
TNOSTRUCTYPE,
Unsupported_Object_Type_Requested :
                            constant XTI_Error_Code := impl-def-static-expression;
TADDRBUSY,
XTI_Address_In_Use :        constant XTI_Error_Code := impl-def-static-expression;
TNOTSUPPORT,
XTI_Operation_Not_Supported :
                            constant XTI_Error_Code := impl-def-static-expression;
-- Get Socket Address Information Error Codes
subtype Addrinfo_Error_Code is Error_Code
   range implementation-defined .. implementation-defined;
EAI_BADFLAGS,
Invalid_Flags :        constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_MEMORY,
Memory_Allocation_Failed :
                       constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_FAIL,
Name_Failed :          constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_NONAME,
Name_Not_Known :       constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_NODATA,
No_Address_For_Name :  constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_SERVICE,
Service_Not_Supported :
                       constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_AGAIN,
Try_Again :            constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_ADDRFAMILY,
Unknown_Address_Type : constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_FAMILY,
Unknown_Protocol_Family :
                       constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_SOCKTYPE,
Unknown_Socket_Type :  constant Addrinfo_Error_Code := impl-def-static-expression;
-- 2.4.7 System Identification
function System_Name return POSIX_String;
function Node_Name return POSIX_String;
function Release return POSIX_String;
function Version return POSIX_String;
function Machine return POSIX_String;
-- 2.4.8 Time Types
type Seconds is range implementation-defined;
--  must include at least -(2**31 - 1) .. (2**31 - 1)
type Minutes is range implementation-defined;
--  must include at least -(2**31 - 1) .. (2**31 - 1)
type Nanoseconds_Base is range implementation-defined;
--  must include at least -(2**31 - 1) .. (2**31 - 1)
subtype Nanoseconds is Nanoseconds_Base range 0 .. (10**9)-1;
type Timespec is private;
function Get_Seconds (Time : Timespec) return Seconds;
```

```
   procedure Set_Seconds
      (Time : in out Timespec;
       S :    in Seconds);
   function Get_Nanoseconds (Time : Timespec) return Nanoseconds;
   procedure Set_Nanoseconds
      (Time : in out Timespec;
       NS :   in Nanoseconds);
   procedure Split
      (Time : in  Timespec;
       S :    out Seconds;
       NS :   out Nanoseconds);
   function To_Timespec
      (S :     Seconds;
       NS :    Nanoseconds)
     return Timespec;
   function "+" (Left, Right : Timespec) return Timespec;
   function "+" (Left : Timespec; Right : Nanoseconds) return Timespec;
   function "-" (Right : Timespec) return Timespec;
   function "-" (Left, Right : Timespec) return Timespec;
   function "-" (Left : Timespec; Right : Nanoseconds) return Timespec;
   function "*" (Left : Timespec; Right : Integer) return Timespec;
   function "*" (Left : Integer; Right : Timespec) return Timespec;
   function "/" (Left : Timespec; Right : Integer) return Timespec;
   function "/" (Left, Right : Timespec) return Integer;
   function "<" (Left, Right : Timespec) return Boolean;
   function "<="(Left, Right : Timespec) return Boolean;
   function ">" (Left, Right : Timespec) return Boolean;
   function ">="(Left, Right : Timespec) return Boolean;
   function To_Duration (Time : Timespec) return Duration;
   function To_Timespec (D : Duration) return Timespec;
private
   implementation-defined
end POSIX;
```

If the Ada language implementation does not support `Standard.Wide_String`, the operations on type `Wide_String` may be omitted from the package.

2.4.1 Constants and Static Subtypes

2.4.1.1 Version Identification

2.4.1.1.1 Synopsis

```
   POSIX_Version : constant := 1997_XX;
   POSIX_Ada_Version : constant := 1998_YY;
```

2.4.1.1.2 Description

The constant `POSIX_Version` shall provide the year and month of the POSIX standard on which this standard is based. The constant `POSIX_Ada_Version` shall provide the year and month of this standard.

2.4.1.2 Optional Facilities

The option subtypes declared in `POSIX` are obsolescent. They are only provided for compatibility with POSIX.5. The description of implementation options is in 2.5. For

each of the option subtypes declared in the package POSIX, the requirements are the same as on the declaration with the corresponding name in POSIX_Options.

2.4.1.3 Bytes and I/O Counts

2.4.1.3.1 Synopsis

```
Byte_Size : constant := implementation-defined-integer;
type IO_Count is optional parent type
   range 0 .. implementation-defined;
subtype IO_Count_Maxima is IO_Count range 32767 .. IO_Count'Last;
type Octet is mod 2**8;
type Octet_Array is array (Positive range <>) of Octet;
function Host_To_Network_Byte_Order (Host_32 : Interfaces.Unsigned_32)
  return Interfaces.Unsigned_32;
function Host_To_Network_Byte_Order (Host_16 : Interfaces.Unsigned_16)
  return Interfaces.Unsigned_16;
function Network_To_Host_Byte_Order (Net_32 : Interfaces.Unsigned_32)
  return Interfaces.Unsigned_32;
function Network_To_Host_Byte_Order (Net_16 : Interfaces.Unsigned_16)
  return Interfaces.Unsigned_16;
```

c

2.4.1.3.2 Description

The constant Byte_Size is the size, in bits, of one byte of data. (See 2.2.2.27.) A file is viewed as a conceptual sequence of bytes, and a position in a file is specified by an offset given as a count of bytes. In this respect the word *byte* is used here in the same sense as in POSIX.1.

NOTE: A byte is likely to be eight bits, but this size is not a requirement.

The type Ada_Streams.Stream_Element_Array is used to represent a buffer that may contain a block of data for an I/O operation.

NOTE: Because the type Ada_Streams.Stream_Element_Array is used for the low-level I/O and message-passing operations, an application may derive types of the class Stream and use the operations defined by this standard to implement the Read and Write operations on the new stream types.

Instantiation of Unchecked_Conversion shall be supported for conversions, in both directions, between the type Ada_Streams.Stream_Element_Array and all other types.

The type IO_Count is used to specify a position in a regular file or a count of units transferred by an I/O operation. An implementation may specify an optional parent type for IO_Count. The maximum file length allowed on a file system shall be no less than IO_Count_Maxima'First and can be as large as IO_Count_Maxima'Last. The file system may limit the length of a file due to capacity limits.

The type Octet is used to specify a data item for use in network I/O operations that require fixed 8-bit values. The type Octet_Array is used to specify a buffer that may contain a block of these data objects to be used for a network I/O operation.

The functions Host_To_Network_Byte_Order and Network_To_Host_Byte_Order shall convert 16-bit and 32-bit quantities between host byte order and network byte order.

c

NOTE: These functions may be required on some hosts to prepare buffers containing these data types for network transmission.
Unlike common practice in C-language network interfaces, these functions are not *required to prepare Internet addresses and port values for network I/O operations. Any byte swapping required for these types will be handled by the implementation of this binding. (See D.1.3.1.)*

2.4.1.4 System Limits

The implementation limit constants and subtypes declared in POSIX are obsolescent. They are only provided for compatibility with POSIX.5. The description of implementation limits is in 2.6. The requirements on the implementation limit constants and subtypes declared in POSIX are the same as on the corresponding declarations in POSIX_Limits, according to the correspondence given in Table 2.2.

Table 2.2 – Constant and Subtype Correspondences

In Package POSIX	In Package POSIX_Limits
Portable_Argument_List_Maximum	Portable_Argument_List_Maximum
Portable_Child_Processes_Maximum	Portable_Child_Processes_Maximum
Portable_Filename_Limit_Maximum	Portable_Filename_Maximum
Portable_Groups_Maximum	Portable_Groups_Maximum
Portable_Input_Line_Limit_Maximum	Portable_Input_Line_Maximum
Portable_Input_Queue_Limit_Maximum	Portable_Input_Queue_Maximum
Portable_Link_Limit_Maximum	Portable_Links_Maximum
Portable_Open_Files_Maximum	Portable_Open_Files_Maximum
Portable_Pathname_Limit_Maximum	Portable_Pathname_Maximum
Portable_Pipe_Limit_Maximum	Portable_Pipe_Length_Maximum
Portable_Stream_Maximum	Portable_Streams_Maximum
Portable_Time_Zone_String_Maximum	Portable_Time_Zone_String_Maximum
Argument_List_Maxima	Argument_List_Maxima
Child_Processes_Maxima	Child_Processes_Maxima
Filename_Limit_Maxima	Filename_Maxima
Groups_Maxima	Groups_Maxima
Input_Line_Limit_Maxima	Input_Line_Maxima
Input_Queue_Limit_Maxima	Input_Queue_Maxima
Link_Limit_Maxima	Links_Maxima
Open_Files_Maxima	Open_Files_Maxima
Pathname_Limit_Maxima	Pathname_Maxima
Pipe_Limit_Maxima	Pipe_Length_Maxima
Stream_Maxima	Streams_Maxima
Time_Zone_String_Maxima	Time_Zone_String_Maxima

2.4.1.5 Blocking Behavior Values

2.4.1.5.1 Synopsis

```
type Blocking_Behavior is (Tasks, Program, Special);
subtype Text_IO_Blocking_Behavior is Blocking_Behavior
   range implementation-defined;
IO_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
```

```
File_Lock_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
Wait_For_Child_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
XTI_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
Sockets_Blocking_Behavior :
   constant Blocking_Behavior := implementation-defined;
subtype Realtime_Blocking_Behavior is Blocking_Behavior
   range implementation-defined;
```

c

2.4.1.5.2 Description

The `Blocking_Behavior` type is used to specify the blocking behavior of POSIX operations. The values of this type are as follows:

`Tasks`

The blocking behavior is per task. If a task executes a blocking operation and the operation blocks, the task does not hold any execution resources, and therefore cannot prevent any other task from executing.

`Program`

The blocking behavior is per process. If a task executes a blocking operation and the operation blocks, the task continues to hold all the execution resources and therefore prevents all the tasks in the POSIX process from executing.

`Special`

The blocking behavior is neither per task nor per process. If a task executes a blocking operation and the operation blocks, the task may continue to hold some execution resource that may prevent some, but not necessarily all, of the other tasks in the process from executing. The specific conditions under which tasks other than the calling task are prevented from executing are implementation defined.

NOTE: With any form of blocking behavior other than per task, there is danger of accidental deadlock in which a blocked task is waiting for an action by another task, but is implicitly preventing the other task from executing by holding needed execution resources.

If the underlying system supports multithreaded processes as defined in POSIX.1, the blocking behavior of all operations defined by this standard shall be `Tasks`.

Otherwise, different classes of POSIX operations may have different blocking behaviors, subject to the following restrictions:

— Per-task blocking behavior is always required for
  — All potentially blocking operations defined in the package `POSIX_Mutexes`.
  — All potentially blocking operations defined in the package `POSIX_Condition_Variables`.
— For all other operations defined by this standard, blocking behavior shall also be per task, except where this behavior is not feasible using the facilities of the underlying system. The conformance document for each implementation shall list any exceptions to this rule and the justification for the exception.

The following constants and subtypes shall indicate the actual blocking behavior of the particular implementation for operations where exceptions to the per task blocking requirement are permitted. They can be used by an application to tailor its execution based on the blocking behavior of specific operations.

`Text_IO_Blocking_Behavior`

Specifies that if the range of the subtype is `Tasks .. Tasks`, the blocking behavior of Ada I/O operations shall be per task. If the range of the subtype is `Program .. Program`, the blocking behavior of Ada I/O operations shall be per process. If the range of the subtype is `Tasks .. Program`, the blocking behavior of Ada I/O operations shall be selectable via the `Form` parameter. (See 8.1.) If the range of the subtype includes `Special`, the blocking behavior of Ada I/O operations is implementation defined.

`IO_Blocking_Behavior`

Specifies the blocking behavior of POSIX I/O operations that may block. (See 8.1, 6.1 and 6.3.)

`File_Lock_Blocking_Behavior`

Specifies the blocking behavior of the file locking operations. (See 6.2.)

`Wait_For_Child_Blocking_Behavior`

Specifies the blocking behavior of the wait for child termination operations. (See 3.1.)

`Realtime_Blocking_Behavior`

Specifies the blocking behavior of the following realtime operations defined by this standard:

— `POSIX_Semaphores.Wait`
— `POSIX_Signals.Await_Signal`
— `POSIX_Signals.Await_Signal_Or_Timeout`
— `POSIX_Shared_Memory_Objects.Open_Shared_Memory`
— `POSIX_Shared_Memory_Objects.Open_Or_Create_Shared_Memory`
— `POSIX_Generic_Shared_Memory.Open_And_Map_Shared_Memory`
— `POSIX_Generic_Shared_Memory.Open_Or_Create_And_Map_Shared_Memory`
— `POSIX_Message_Queues.Open`
— `POSIX_Message_Queues.Open_Or_Create`
— `POSIX_Message_Queues.Send`
— `POSIX_Message_Queues.Receive`
— `Generic_Message_Queues.Send`
— `Generic_Message_Queues.Receive`
— `POSIX_Asynchronous_IO.List_IO_Wait`
— `POSIX_Asynchronous_IO.Await_IO_Or_Timeout`
— `POSIX_Asynchronous_IO.Await_IO`

`XTI_Blocking_Behavior`

Specifies the blocking behavior of the XTI DNI operations (see Section 17). |C

`Sockets_Blocking_Behavior`

Specifies the blocking behavior of the Sockets DNI operations (see Section 18). c

NOTE: `Realtime_Blocking_Behavior` is a range, rather than a simple constant. Using a range allows for the possibility that an implementation may have different blocking behavior for different operations in the list above. However, the recommended implementation is that the blocking behavior of all these operations be per to permit compile-time optimization.

2.4.1.6 Signal Masking for Interruptible Operations

2.4.1.6.1 Synopsis

```
type Signal_Masking is (No_Signals, RTS_Signals, All_Signals);
```

During some POSIX operations, it is desirable to block the receipt of certain classes of signals. The type `Signal_Masking` specifies what class of signal is masked.

`No_Signals`

No additional signals are masked during the POSIX operation.

`RTS_Signals`

Only the signals reserved for the Ada language implementation are added to the signal mask during the POSIX operation. (See 3.3.1.)

`All_Signals`

All signals that can be masked are masked during the POSIX operation.

2.4.2 POSIX Characters

2.4.2.1 Synopsis

```
type POSIX_Character is
   (
   --  ' ','0','1','2','3','4','5','6','7','8','9',
   --  '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',
   --  '.','_','-','/','"','#', '&', ''', '(', ')',
   --  '*', '+', ',', ':', ';', '<', '=', '>', '|'
    other characters are implementation defined);
```

2.4.2.2 Description

The type `POSIX_Character` shall include an element for each possible bit representation of size `POSIX_Character'Size`.

It shall be possible to represent all data read from and written to files in `POSIX.POSIX_Character`s or concatenations of `POSIX.POSIX_Character`s.

The implementation shall provide correspondences between the standard Ada type `Character` and and the underlying character set of the implementation, type `POSIX_Character`. This correspondence shall have the following characteristics:

(1) The standard Ada string types, `Standard.Character`, and `Wide_Character` shall be as defined in the Ada RM {1}.

(2) A conforming application shall make no assumption about the ordering of the enumeration and the size of the representation. Therefore, a conforming application shall make no assumption that the results of executing `"<"`, `"<="`, `">"`, `">="`, `'Pred`, and `Succ` are preserved across conversions between the standard Ada string types and `POSIX_String`.

(3) A core subset of `Standard.Character` shall be placed into an invertible character mapping with the corresponding subset of `POSIX_Character`, so that characters with the same enumeration literal are mapped to each other. The core subset shall include the following characters:

— Digits:

```
0 1 2 3 4 5 6 7 8 9
```

— Uppercase letters:

```
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
```

— Lowercase letters:

```
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
```

— Special characters: space, ampersand (&), apostrophe (’), left parenthesis ((), right parenthesis ()), star (*), plus (+), comma (,), hyphen (-), period (.), slash (/), colon (:), semicolon (;), less than (<), equals (=), greater than (>), underscore (_).

— Format effectors: horizontal tabulation, vertical tabulation, carriage return, line feed, form feed.

(4) All remaining characters of the type `Standard.Character` shall be mapped to some subset of the remaining characters within `POSIX_Character`. This standard does not require invertibility of this mapping.

(5) Any remaining characters of the type `POSIX_Character` would have no counterpart in `Standard.Character`.

2.4.3 POSIX Strings

2.4.3.1 Synopsis

```
type POSIX_String is array (Positive range <>) of POSIX_Character;
function To_POSIX_String (Str : String) return POSIX_String;
function To_POSIX_String (Str : Wide_String)return POSIX_String;
function To_String (Str : POSIX_String) return String;
function To_Wide_String (Str : POSIX_String) return Wide_String;
function To_Stream_Element_Array (Buffer : POSIX_String)
  return Ada_Streams.Stream_Element_Array;
function To_POSIX_String (Buffer : Ada_Streams.Stream_Element_Array)
  return POSIX_String;
subtype Filename is POSIX_String;
subtype Pathname is POSIX_String;
function Is_Filename (Str : POSIX_String) return Boolean;
function Is_Pathname (Str : POSIX_String) return Boolean;
function Is_Portable_Filename (Str : POSIX_String) return Boolean;
function Is_Portable_Pathname (Str : POSIX_String) return Boolean;
```

2.4.3.2 Description

The type `POSIX_String` is an array of `POSIX_Character`. This type is used as the argument to all POSIX procedures or functions that require a string.' All values of type `POSIX_String` returned by functions defined in this standard shall have the attribute `'First` = 1 unless otherwise noted.

The function `To_POSIX_String` taking arguments of type `Stream_Element_Array` shall convert an `Ada_Streams.Stream_Element_Array` value to a `POSIX_String` value. The function `To_Stream_Element_Array` shall convert a `POSIX_String` value to a `Ada_Streams.Stream_Element_Array` value. These operations shall obey the semantics of instantiations of `Unchecked_Conversion`, as defined in 13.9 J.1 of the Ada RM {1}.

The functions `To_POSIX_String` taking arguments of types `String` and `Wide_String` and the functions `To_String`, and `To_Wide_String` are used to convert between the standard Ada string types and the `POSIX_String` type. All characters in the core subset defined in 2.4.2 shall be *convertible* and their conversion shall be *invertible*. The results of converting the other characters in the standard Ada character set is implementation defined. A character in one character set may translate into a sequence of characters in the other character set. An implementation may convert the string sequences back into the single character when doing the inverse translation.

The significance of the constants `Portable_Pathname_Limit_Maximum` and `Portable_Filename_Limit_Maximum` and the subtypes `Pathname_Limit_Maxima` and `Filename_Limit_Maxima` is explained for the corresponding declarations in the package `POSIX_Limits` (2.6).

The function `Is_Filename` shall return the value `True` if the parameter has the correct syntax (or form) to be a filename for this implementation. The function `Is_Pathname` shall return the value `True` if the parameter has the correct syntax (or form) to be a pathname for this implementation. These functions shall check all constraints set on filename and pathname by the implementation that can be checked without accessing the file system directly. None of the pathname components need exist, and access rights are not checked. These functions shall not raise any exceptions.

The function `Is_Portable_Filename` shall return the value `True` if the parameter has the correct syntax (or form) to be a portable filename. The function `Is_Portable_Pathname` shall return the value `True` if the parameter has the correct syntax (or form) to be a portable pathname. These functions shall not raise any exceptions, and they shall check all constraints on portable filenames and pathnames. The filename need not exist, and access rights are not checked.

2.4.3.3 Error Handling

`Constraint_Error` shall be raised by the string conversion functions `To_POSIX_String`, `To_String`, and `To_Wide_String` if the conversion is not possible.

No exceptions shall be raised by `Is_Filename`, `Is_Pathname`, `Is_Portable_Filename`, and `Is_Portable_Pathname`.

2.4.4 String Lists

2.4.4.1 Synopsis

```
type POSIX_String_List is limited private;
Empty_String_List : constant POSIX_String_List;
procedure Make_Empty (List : in out POSIX_String_List);
procedure Append
   (List : in out POSIX_String_List;
    Str :  in POSIX_String);
generic
   with procedure Action
      (Item : in POSIX_String;
       Quit : in out Boolean);
procedure For_Every_Item (List : in POSIX_String_List);
function Length (List : POSIX_String_List) return Natural;
function Value
   (List :  POSIX_String_List;
    Index : Positive)
  return POSIX_String;
```

2.4.4.2 Description

The type POSIX_String_List is used to process arbitrary lists of POSIX_Strings. The value Empty_String_List represents a POSIX_String_List that contains no POSIX_Strings and is used as a parameter to procedures where an empty POSIX_-String_List is needed. All objects of type POSIX_String_List shall have the initial value Empty_String_List.

The procedure Make_Empty shall remove any POSIX_String values on the POSIX_-String_List value supplied via the parameter, free any allocated storage associated with the supplied POSIX_String_List, and then set the supplied POSIX_String_-List to the value Empty_String_List.

The procedure Append shall append the POSIX_String value supplied via Str to the end of the POSIX_String_List supplied via List. The POSIX_String value supplied via Str shall be copied by the implementation so that the supplied POSIX_String value can be modified without changing the value associated with the POSIX_String_List. The original bounds of the POSIX_String value supplied via Str shall be preserved.

The application program instantiates the generic procedure For_Every_Item, providing an actual procedure for the generic formal procedure Action. When invoked, the newly created instance of For_Every_Item shall invoke the actual procedure supplied for Action once for every POSIX_String value (as the parameter Str) in the POSIX_String_List value supplied via List in the invocation of For_Every_Item. The original bounds of the POSIX_String values shall be preserved. The actual procedure supplied for Action shall be able to force termination of the instance of For_Every_Item by setting the value of Quit to True. Prior to calling Action, For_Every_Item shall set Quit to False. If Action modifies List (as a side effect) the effect of calling the instantiation of For_Every_Item is undefined. Any exceptions raised by the actual procedure supplied for Action shall terminate the iteration of the instantiation of For_Every_Item and shall propagate to the caller of the instantiation.

The function Length shall return the number of (null and nonnull) POSIX_String values in the POSIX_String_List.

NOTE: *Length(S) = 0 if and only if S = Empty_String_List.*

The function Value shall return the value of type POSIX_String in the Index position in the list. The original bounds of the POSIX_String shall be preserved. Constraint_Error shall be raised if Index is greater than the length of the list.

2.4.4.3 Error Handling

Constraint_Error shall be raised by an invocation of the Value function if the parameter Index has a value greater than the list length.

No exceptions are specified by this standard for Make_Empty, Append, For_Every_Item, or Length.

2.4.5 Option Sets

2.4.5.1 Synopsis

```
type Option_Set is private;
function Empty_Set return Option_Set;
function "+" (L, R : Option_Set) return Option_Set;
function "-" (L, R : Option_Set) return Option_Set;
function "<" (Left, Right : Option_Set) return Boolean;
function "<="(Left, Right : Option_Set) return Boolean;
function ">" (Left, Right : Option_Set) return Boolean;
function ">="(Left, Right : Option_Set) return Boolean;
Option_1 :  constant Option_Set;
Option_2 :  constant Option_Set;
Option_3 :  constant Option_Set;
Option_4 :  constant Option_Set;
Option_5 :  constant Option_Set;
Option_6 :  constant Option_Set;
Option_7 :  constant Option_Set;
Option_8 :  constant Option_Set;
Option_9 :  constant Option_Set;
Option_10 : constant Option_Set;
Option_11 : constant Option_Set;
Option_12 : constant Option_Set;
Option_13 : constant Option_Set;
Option_14 : constant Option_Set;
Option_15 : constant Option_Set;
Option_16 : constant Option_Set;
Option_17 : constant Option_Set;
Option_18 : constant Option_Set;
Option_19 : constant Option_Set;
Option_20 : constant Option_Set;
Option_21 : constant Option_Set;
Option_22 : constant Option_Set;
Option_23 : constant Option_Set;
Option_24 : constant Option_Set;
Option_25 : constant Option_Set;
Option_26 : constant Option_Set;
Option_27 : constant Option_Set;
Option_28 : constant Option_Set;
Option_29 : constant Option_Set;
Option_30 : constant Option_Set;
Option_31 : constant Option_Set;
```

2.4.5.2 Description

The type `Option_Set` shall be used to represent sets of options. All objects of this type shall have as their initial value the value returned by `Empty_Set`. The function `Empty_Set` shall return a value of type `Option_Set` with no options in it. The operation `"+"` shall return an option set containing exactly the set of options that are members of one or both of the two operand sets, *i.e.*, the union operation on option sets. The binary operation `"-"` shall return the option set whose members are the options that are members of the left-hand operand set and are not members of the right-hand operand set.

An implementation is permitted to add declarations to the `POSIX` package for additional deferred constants of the `Option_Set` type, provided the names are of the form `Option_N` where `N` is an unsigned integer literal, and the range of `N` is contiguous and includes `1..31`, and all the values of the constants are distinct.

Deferred constants of the `Option_Set` type are also defined. These constants shall each denote a unique set containing a single option. They are provided for use by implementations in defining constants for singleton sets of options, of types derived from `Option_Set`.

Operations on the `Option_Set` type are specified to allow comparison of two option sets. If `Left` and `Right` are option sets, the operations have the meanings shown in Table 2.3.

Table 2.3 – Option Set Comparisons

Relation	Meaning
`Left <= Right`	`Left` is a subset of `Right`
`Left < Right`	`Left` is a proper subset of `Right`
`Left >= Right`	`Right` is a subset of `Left`
`Left > Right`	`Right` is a proper subset of `Left`

2.4.5.3 Error Handling

No exceptions shall be raised by these operations.

2.4.6 Error Codes and Exceptions

2.4.6.1 Synopsis

```
POSIX_Error : exception;
type Error_Code is range implementation-defined;
function Get_Error_Code return Error_Code;
procedure Set_Error_Code (Error : in Error_Code);
function Is_POSIX_Error (Error : Error_Code) return Boolean;
function Image (Error : Error_Code) return String;
No_Error : constant Error_Code := 0;
E2BIG,
Argument_List_Too_Long :     constant Error_Code := impl-def-static-expression;
EFAULT,
Bad_Address :                constant Error_Code := impl-def-static-expression;
EBADF,
Bad_File_Descriptor :        constant Error_Code := impl-def-static-expression;
```

```
EBADMSG,
Bad_Message :                 constant Error_Code := impl-def-static-expression;
EPIPE,
Broken_Pipe :                 constant Error_Code := impl-def-static-expression;
ENOTEMPTY,
Directory_Not_Empty :         constant Error_Code := impl-def-static-expression;
ENOEXEC,
Exec_Format_Error :           constant Error_Code := impl-def-static-expression;
EEXIST,
File_Exists :                 constant Error_Code := impl-def-static-expression;
EFBIG,
File_Too_Large :              constant Error_Code := impl-def-static-expression;
ENAMETOOLONG,
Filename_Too_Long :           constant Error_Code := impl-def-static-expression;
EXDEV,
Improper_Link :               constant Error_Code := impl-def-static-expression;
ENOTTY,
Inappropriate_IO_Control_Operation :
                              constant Error_Code := impl-def-static-expression;
EIO,
Input_Output_Error :          constant Error_Code := impl-def-static-expression;
EINTR,
Interrupted_Operation :       constant Error_Code := impl-def-static-expression;
EINVAL,
Invalid_Argument :            constant Error_Code := impl-def-static-expression;
ESPIPE,
Invalid_Seek :                constant Error_Code := impl-def-static-expression;
EISDIR,
Is_A_Directory :              constant Error_Code := impl-def-static-expression;
EMSGSIZE,
Message_Too_Long :            constant Error_Code := impl-def-static-expression;
ECHILD,
No_Child_Process :            constant Error_Code := impl-def-static-expression;
ENOLCK,
No_Locks_Available :          constant Error_Code := impl-def-static-expression;
ENOSPC,
No_Space_Left_On_Device :     constant Error_Code := impl-def-static-expression;
ENODEV,
No_Such_Operation_On_Device :
                              constant Error_Code := impl-def-static-expression;
ENXIO,
No_Such_Device_Or_Address :
                              constant Error_Code := impl-def-static-expression;
ENOENT,
No_Such_File_Or_Directory :
                              constant Error_Code := impl-def-static-expression;
ESRCH,
No_Such_Process :             constant Error_Code := impl-def-static-expression;
ENOTDIR,
Not_A_Directory :             constant Error_Code := impl-def-static-expression;
ENOMEM,
Not_Enough_Space :            constant Error_Code := impl-def-static-expression;
ECANCELED,
Operation_Canceled :          constant Error_Code := impl-def-static-expression;
EINPROGRESS,
Operation_In_Progress :       constant Error_Code := impl-def-static-expression;
ENOSYS,
Operation_Not_Implemented :   constant Error_Code := impl-def-static-expression;
EPERM,
Operation_Not_Permitted :     constant Error_Code := impl-def-static-expression;
```

```
ENOTSUP,
Operation_Not_Supported :    constant Error_Code := impl-def-static-expression;
EACCES,
Permission_Denied :          constant Error_Code := impl-def-static-expression;
EROFS,
Read_Only_File_System :      constant Error_Code := impl-def-static-expression;
EBUSY,
Resource_Busy :              constant Error_Code := impl-def-static-expression;
EDEADLK,
Resource_Deadlock_Avoided :  constant Error_Code := impl-def-static-expression;
EAGAIN,
Resource_Temporarily_Unavailable :
                             constant Error_Code := impl-def-static-expression;
ETIMEDOUT,
Timed_Out :                  constant Error_Code := impl-def-static-expression;
EMLINK,
Too_Many_Links :             constant Error_Code := impl-def-static-expression;
EMFILE,
Too_Many_Open_Files :        constant Error_Code := impl-def-static-expression;
ENFILE,
Too_Many_Open_Files_In_System :
                             constant Error_Code := impl-def-static-expression;
-- Socket Error Codes
EADDRINUSE,
Address_In_Use :             constant Error_Code := impl-def-static-expression;
EADDRNOTAVAIL,
Address_Not_Available :      constant Error_Code := impl-def-static-expression;
EALREADY,
Already_Awaiting_Connection :
                             constant Error_Code := impl-def-static-expression;
ECONNABORTED,
Connection_Aborted :         constant Error_Code := impl-def-static-expression;
ECONNREFUSED,
Connection_Refused :         constant Error_Code := impl-def-static-expression;
ECONNRESET,
Connection_Reset :           constant Error_Code := impl-def-static-expression;
EDOM,
Domain_Error :               constant Error_Code := impl-def-static-expression;
EHOSTDOWN,
Host_Down :                  constant Error_Code := impl-def-static-expression;
EHOSTUNREACH,
Host_Unreachable :           constant Error_Code := impl-def-static-expression;
EAFNOSUPPORT,
Incorrect_Address_Type :     constant Error_Code := impl-def-static-expression;
EISCONN,
Is_Already_Connected :       constant Error_Code := impl-def-static-expression;
ENETDOWN,
Network_Down :               constant Error_Code := impl-def-static-expression;
ENETRESET,
Network_Reset :              constant Error_Code := impl-def-static-expression;
ENETUNREACH,
Network_Unreachable :        constant Error_Code := impl-def-static-expression;
ENOBUFS,
No_Buffer_Space :            constant Error_Code := impl-def-static-expression;
ENOTSOCK,
Not_A_Socket :               constant Error_Code := impl-def-static-expression;
ENOTCONN,
Not_Connected :              constant Error_Code := impl-def-static-expression;
EOPNOTSUPP,
Option_Not_Supported :       constant Error_Code := impl-def-static-expression;
```

C

```
EPROTONOSUPPORT,
Protocol_Not_Supported :     constant Error_Code := impl-def-static-expression;
ESOCKTNOSUPPORT,
Socket_Type_Not_Supported :  constant Error_Code := impl-def-static-expression;
EWOULDBLOCK,
Would_Block :                constant Error_Code := impl-def-static-expression;
EPROTOTYPE,
Wrong_Protocol_Type :        constant Error_Code := impl-def-static-expression;
-- XTI Error Codes
subtype XTI_Error_Code is Error_Code
   range implementation-defined .. implementation-defined;
TBUFOVFLW,
Buffer_Not_Large_Enough :   constant XTI_Error_Code := impl-def-static-expression;
TPROVMISMATCH,
Communications_Provider_Mismatch :
                            constant XTI_Error_Code := impl-def-static-expression;
TNOADDR,
Could_Not_Allocate_Address :
                            constant XTI_Error_Code := impl-def-static-expression;
TQFULL,
Endpoint_Queue_Full :       constant XTI_Error_Code := impl-def-static-expression;
TBADQLEN,
Endpoint_Queue_Length_Is_Zero :
                            constant XTI_Error_Code := impl-def-static-expression;
TLOOK,
Event_Requires_Attention :  constant XTI_Error_Code := impl-def-static-expression;
TFLOW,
Flow_Control_Error :        constant XTI_Error_Code := impl-def-static-expression;
TBADDATA,
Illegal_Data_Range :        constant XTI_Error_Code := impl-def-static-expression;
TBADADDR,
Incorrect_Address_Format :  constant XTI_Error_Code := impl-def-static-expression;
TBADOPT,
Incorrect_Or_Illegal_Option :
                            constant XTI_Error_Code := impl-def-static-expression;
TRESQLEN,
Incorrect_Surrogate_Queue_Length :
                            constant XTI_Error_Code := impl-def-static-expression;
TACCES,
Insufficient_Permission :   constant XTI_Error_Code := impl-def-static-expression;
TBADNAME,
Invalid_Communications_Provider :
                            constant XTI_Error_Code := impl-def-static-expression;
TBADF,
Invalid_File_Descriptor :   constant XTI_Error_Code := impl-def-static-expression;
TBADFLAG,
Invalid_Flag :              constant XTI_Error_Code := impl-def-static-expression;
TBADSEQ,
Invalid_Sequence_Number :   constant XTI_Error_Code := impl-def-static-expression;
TNODATA,
No_Data_Available :         constant XTI_Error_Code := impl-def-static-expression;
TNODIS,
No_Disconnect_Indication_On_Endpoint :
                            constant XTI_Error_Code := impl-def-static-expression;
TNOREL,
No_Orderly_Release_Indication_On_Endpoint :
                            constant XTI_Error_Code := impl-def-static-expression;
TNOUDERR,
No_Unit_Data_Error_On_Endpoint :
                            constant XTI_Error_Code := impl-def-static-expression;
```

```
TOUTSTATE,
Operation_Not_Valid_For_State :
                          constant XTI_Error_Code := impl-def-static-expression;
TINDOUT,
Outstanding_Connection_Indications :
                          constant XTI_Error_Code := impl-def-static-expression;
TPROTO,
Protocol_Error :          constant XTI_Error_Code := impl-def-static-expression;
TSTATECHNG,
State_Change_In_Progress : constant XTI_Error_Code := impl-def-static-expression;
TRESADDR,
Surrogate_File_Descriptor_Mismatch :
                          constant XTI_Error_Code := impl-def-static-expression;
TNOSTRUCTYPE,
Unsupported_Object_Type_Requested :
                          constant XTI_Error_Code := impl-def-static-expression;
TADDRBUSY,
XTI_Address_In_Use :      constant XTI_Error_Code := impl-def-static-expression;
TNOTSUPPORT,
XTI_Operation_Not_Supported :
                          constant XTI_Error_Code := impl-def-static-expression;
-- Get Socket Address Information Error Codes
subtype Addrinfo_Error_Code is Error_Code
   range implementation-defined .. implementation-defined;
EAI_BADFLAGS,
Invalid_Flags :       constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_MEMORY,
Memory_Allocation_Failed :
                      constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_FAIL,
Name_Failed :         constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_NONAME,
Name_Not_Known :      constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_NODATA,
No_Address_For_Name : constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_SERVICE,
Service_Not_Supported :
                      constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_AGAIN,
Try_Again :           constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_ADDRFAMILY,
Unknown_Address_Type : constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_FAMILY,
Unknown_Protocol_Family :
                      constant Addrinfo_Error_Code := impl-def-static-expression;
EAI_SOCKTYPE,
Unknown_Socket_Type : constant Addrinfo_Error_Code := impl-def-static-expression;
```

c

2.4.6.2 Description

Most functions and procedures raise the exception POSIX_Error when an error or exceptional condition occurs during the processing of the call. The Error Code of the affected task shall be set when this exception is raised. If more than one error occurs in the processing, this standard does not define in what order the errors are detected; therefore, any one of the possible error codes may be set. If the exception is not raised, the current Error Code shall be unchanged for the current task.

Implementations may support additional error codes not named in this subclause, may set the Error Code to values named in this subclause under circumstances other

than those described in this subclause, or may contain extensions or limitations that prevent some errors from occurring. The error handling subclause in each package description specifies which error conditions shall be detected by all implementations and which may be optionally detected by an implementation. Each implementation shall document, in the conformance document, situations in which each of the optional conditions is detected. If no error condition is detected, the associated action requested shall be successful.

Implementations may generate the error codes named in this subclause under circumstances other than those described if and only if all those error conditions can always be treated identically to the error conditions as described in this subclause. Implementations may support additional error codes not listed in this subclause, but they shall not generate a different error code than one required by this subclause for an error condition described in this subclause.

The function `Get_Error_Code` shall return the current Error Code associated with the most recent POSIX operation executed in the context of the calling task. If the Error Code for the calling task has not been set, the value returned is undefined.

The procedure `Set_Error_Code` shall set the Error Code attribute of the calling task to the value specified.

The function `Is_POSIX_Error` shall return `True` if the parameter `Error` specifies is an error code defined by this standard. For all other values, `Is_POSIX_Error` shall return `False`.

The function `Image` shall return a string identifying the error code specified by the parameter `Error`. If `Error` is one of the error codes defined by this standard, the value returned by `Image` shall be the second of the two constant identifiers specified in 2.4.1 for the error code, in uppercase. Otherwise, the value returned by `Image` is implementation defined, subject to the requirement that the value returned shall be distinct for each error code used by the implementation.

If the implementation supports package `Ada.Exceptions`, as defined in 11.4.1 of the Ada RM {1} and if the function `Ada.Exceptions.Exception_Message` is called for an occurrence of `POSIX_Error` raised with a given error code value by the implementation of this standard, the string returned by `Exception_Message` shall be the same as the value returned by `Image` for that error code value.

The subtype `XTI_Error_Code` shall be defined to provide discrete `Error_Code` ranges for the XTI Detailed Network Interface optional services. The subtype `Addrinfo_Error_Code` shall be defined to provide discrete `Error_Code` ranges for the Sockets Detailed Network Interface with the Network Management optional services (*i.e.*, `Get_Socket_Address_Info`). The implementation shall ensure that the ranges of these subtypes of `Error_Code` do not overlap. c

The following descriptions identify the possible *errors* that can cause the exception `POSIX_Error` to be raised and the Error Code attribute of a task to be set by operations defined in this standard. These general descriptions are more precisely defined in the error handling subclauses of the packages that raise them.

`No_Error`

This code is not set by any POSIX operation. It is returned by some POSIX

operations when status is requested and no error has occurred. It shall have the value zero.

`Address_In_Use`

The requested communications address is in use.

`Address_Not_Available`

The specified socket address is not available from the local machine.

`Already_Awaiting_Connection`

The socket is nonblocking and a previous connection attempt has not yet been completed. c

`Argument_List_Too_Long`

The sum of the number of `POSIX_Character`s used by the argument list and environment list of the new process image was greater than Argument List Maximum.

`Bad_File_Descriptor`

A file descriptor argument was out of range or referred to no open file, or a read (write) request was made to a file that was only open for writing (reading).

`Bad_Address`

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

NOTE: This error is not specified for any of the operations defined by this standard. If the implementation uses this error code, the likely cause is an uninitialized parameter.

`Bad_Message`

The implementation has detected a corrupted message.

`Broken_Pipe`

A write was attempted on a pipe or FIFO for which there was no process to read the data.

`Buffer_Not_Large_Enough`

For XTI, the amount of data to be returned in one of the buffers is greater than the maximum length specified. If `Endpoint` is a passive endpoint with data count larger than one, it remains in the `Incoming_Connect` state; otherwise, the state of the endpoint is set to `Idle`.

`Communications_Provider_Mismatch`

For the XTI `Accept_Connection` procedure, indicates that `Current_Endpoint` and `Responding_Endpoint` do not specify the same transport provider.

`Connection_Aborted`

The socket connection was aborted locally.

`Connection_Refused`

The attempt to connect a socket was forcefully refused. c

`Connection_Reset`

The peer has aborted the socket connection.

`Could_Not_Allocate_Address`

An XTI address could not be allocated by the transport provider. c

`Directory_Not_Empty`

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

`Domain_Error`

A time specification is too large for the socket. This error code is returned only on calls to `Set_Socket_Receive_Timeout` and `Set_Socket_Send_Timeout`.

`Endpoint_Queue_Full`

The maximum number of outstanding XTI connection indications has been reached.

`Endpoint_Queue_Length_Is_Zero`

The XTI `Endpoint_Queue_Length` was zero when it was expected to be greater than zero.

`Event_Requires_Attention`

An indication that an asynchronous event is outstanding on the XTI transport endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.) c

`Exec_Format_Error`

A request was made to execute a file that, although it had the appropriate permissions, was not in the format required by the implementation for executable files.

`File_Exists`

An existing file was specified in an inappropriate context.

`File_Too_Large`

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

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`Flow_Control_Error`

`POSIX_IO.Non_Blocking` was set, but the flow control mechanism prevented the transport provider from accepting the XTI function at this time.

`Host_Down`

The destination host for the socket has been determined to be down or disconnected.

`Host_Unreachable`

The host specified in the socket address is not reachable. c

`Illegal_Data_Range`

The application attempted to send an illegal amount of XTI data. c

`Improper_Link`

A link to a file on another file system was attempted.

`Inappropriate_IO_Control_Operation`

A control function was attempted for a file or special file for which the operation was inappropriate.

`Incorrect_Address_Type`

The type of the designated address object is incorrect for this socket.

`Incorrect_Address_Format`

The XTI address specified by the application contained an incorrect protocol address, or the protocol address was in an incorrect format.

`Incorrect_Or_Illegal_Option`

The XTI option specified by the application contained incorrect information, or the information was in an incorrect format.

`Incorrect_Surrogate_Queue_Length`

An attempt was made to accept an XTI connection on `Responding_Endpoint` (where `Responding_Endpoint` does not equal `Current_Endpoint`) with an `Endpoint_Queue_Length` greater than zero. c

`Input_Output_Error`

Some physical input or output error occurred. This error may be reported on a subsequent operation on the same file descriptor. Any other error-causing operation on the same file descriptor may cause the `Input_Output_Error` to be lost.

`Insufficient_Permission`

Indicates that either the application does not have the permission to accept the XTI connection on the responding transport endpoint, to use a specified option, or to use the specified address. c

`Interrupted_Operation`

An asynchronous signal was caught by the task during the execution of an interruptible function. If the signal handler returns normally, then the interrupted function call may raise the `POSIX_Error` exception with this error code.

`Invalid_Argument`

Some invalid argument was supplied.

`Invalid_Communications_Provider`

The application specified a bad name for the XTI transport provider.

`Invalid_File_Descriptor`

For XTI, a bad file descriptor was specified in either `Current_Endpoint` or `Responding_Endpoint`, or the application is illegally accepting a connection on the same transport endpoint on which the connection indication arrived. This error may be returned when the `Endpoint` has been previously closed or an erroneous number may have been passed to the call. c

`Invalid_Flag`

The application specified an invalid XTI flag.

`Invalid_Flags`

The application attempted to set an invalid value for the Flags attribute of an `Socket_Address_Info` object (see 18.4.7.2). c

`Invalid_Seek`

A `Seek` operation was issued on a pipe or FIFO.

`Invalid_Sequence_Number`

The XTI application specified an incorrect sequence number or in the case of a connect request being rejected, an invalid `Call`. parameter. Some outbound data queued for this endpoint may be lost. c

`Is_A_Directory`

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

`Is_Already_Connected`

The socket is already connected.

`Memory_Allocation_Failed`

There was a memory allocation failure when trying to allocate storage for the return value of a function (see 18.4.7.2).

`Message_Too_Long`

The message buffer length is inappropriate, or a socket requires that the message be sent atomically and the size of the message makes this impossible, or the number of segments in a multisegment message exceeds the system limit.

`Name_Failed`

A nonrecoverable error occurred when attempting to resolve the name (see 18.4.7.2).

`Name_Not_Known`

The name is not known. Neither name nor service were passed, at least one which must be passed (see 18.4.7.2).

`Network_Down`

The local network connection is not operational.

`Network_Reset`

The connection was aborted by the network.

`Network_Unreachable`

The network is not reachable from this host.

`No_Address_For_Name`

A valid name was passed, but no address was associated with the name (see 18.4.7.2).

`No_Buffer_Space`

Insufficient resources were available in the system to perform the operation. c

`No_Child_Process`

A call to a `Wait_For_Child_Process` procedure was executed by a process that had no existing child processes or whose status had not yet been reported.

`No_Data_Available`

In nonblocking mode, no XTI connection indications are present; or data are not available; or no connection confirmations have arrived yet; or the procedure has successfully issued a connect, but did not wait for a response from the application.

`No_Disconnect_Indication_On_Endpoint`

No disconnect indications are found on the XTI transport endpoint specified by `Endpoint`. c

`No_Locks_Available`

A system-imposed limit on the number of simultaneous file and record locks was reached, and no more were available at that time.

`No_Orderly_Release_Indication_On_Endpoint`

No orderly release indication is outstanding on the XTI transport endpoint specified by `Endpoint`. c

`No_Space_Left_On_Device`

During a write on a regular file or when extending a directory, no free space was left on the device.

`No_Such_Operation_On_Device`

An attempt was made to apply an inappropriate operation on a device, for example, trying to read a write-only device such as a printer.

`No_Such_Device_Or_Address`

Input or output on a special file referred to a device that did not exist or made a request beyond the limits of the device. This error may also occur when, for example, a tape drive is not online or a removable disk is not loaded on a drive.

`No_Such_File_Or_Directory`

A component of a specified pathname did not exist, or the pathname was an empty string.

`No_Such_Process`

No process could be found corresponding to that specified by the given process ID.

`No_Unit_Data_Error_On_Endpoint`

No unit data error indication currently exists on the specified XTI communications endpoint. c

`Not_A_Directory`

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

`Not_A_Socket`

The file descriptor does not refer to a socket. c

`Not_Connected`

The socket is not connected or otherwise has not had the peer prespecified. c

`Not_Enough_Space`

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

`Operation_Canceled`

The associated asynchronous operation was canceled before completion.

`Operation_In_Progress`

An asynchronous operation has not yet completed; or a socket is nonblocking and the connection cannot be completed immediately. c

`Operation_Not_Implemented`

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

`Operation_Not_Permitted`

An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resource.

`Operation_Not_Supported`

The implementation does not support this feature of the standard.

`Operation_Not_Valid_For_State`

The procedure was called with the transport provider in the wrong XTI state (bad sequence).

`Option_Not_Supported`

The socket type specified does not support one or more of the options selected.

`Outstanding_Connection_Indications`

An indication that there are outstanding XTI connection indications on the endpoint when the application called `Accept_Connection` with `Current_Endpoint` equal to `Responding_Endpoint`. (The application must either first accept the other connections on a different endpoint by using `Accept_Connection` or reject them with `Send_Disconnect_Request`.) c

`Permission_Denied`

An attempt was made to access a file in a way forbidden by its file access permissions.

`Protocol_Error`

An XTI communication problem has occurred and there is no other appropriate error number.

`Protocol_Not_Supported`

The protocol family is not supported, or the type or specified protocol is not supported within the protocol family. c

`Read_Only_File_System`

An attempt was made to modify a file or directory on a file system that was read-only at that time.

`Resource_Busy`

An attempt was made to use a system resource that was not available at the time because it was being used by a process in a manner that would have conflicted with the request being made by this process.

`Resource_Deadlock_Avoided`

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

`Resource_Temporarily_Unavailable`

A temporary condition caused failure of the call, and later calls to the same routine may complete normally.

`Service_Not_Supported`

The service passed was not recognized for the specified socket type (see 18.4.7.2).

`Socket_Type_Not_Supported`

Socket type not supported.

`State_Change_In_Progress`

The XTI transport provider is undergoing a state change.

`Surrogate_File_Descriptor_Mismatch`

The XTI transport provider only allows `Current_Endpoint` and `Responding_Endpoint` to be bound to the same address. c

`Timed_Out`

The operation timed out without completing normally.

NOTE: Most operations with timeouts return error code `Resource_Temporarily_Unavailable`.

`Too_Many_Links`

An attempt was made to have the link count of a single file exceed the value returned by function `POSIX_Configurable_File_Limits.Links_Maximum` for that file.

`Too_Many_Open_Files`

An attempt was made to open more file descriptors than the limit returned by `POSIX_Configurable_System_Limits.Open_Files_Maximum`.

`Too_Many_Open_Files_In_System`

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

`Try_Again`

A temporary and possibly transient error occurred, such as a failure of a server to respond (see 18.4.7.2).

`Unknown_Address_Type`

The address found for the name was of an unsupported type.

`Unknown_Protocol_Family`

The protocol family was not recognized. c

`Unknown_Socket_Type`

The intended socket type was not recognized by `Get_Socket_Address_Info`.

`Unsupported_Object_Type_Requested`

An unsupported XTI object was requested, possibly a request for an object which is inconsistent with the transport provider type specified, *i.e.*, connection-oriented or connectionless.

`Would_Block`

The socket is marked as nonblocking and no connections are present to be accepted; or `Process_OOB_Data` was selected and the implementation does not support blocking to await out-of-band-data.

`Wrong_Protocol_Type`

Protocol has wrong type for socket. For example, a socket of one type attempted to connect to a socket of a different type.

`XTI_Address_In_Use`

The requested XTI communications address is in use.

`XTI_Operation_Not_Supported`

The transport provider does not support this XTI function. c

2.4.6.3 Error Handling

No exceptions are specified by this standard for `Get_Error_Code`, `Set_Error_Code`, `Is_POSIX_Error`, and `Image`.

2.4.7 System Identification

2.4.7.1 Synopsis

```
function System_Name return POSIX_String;
function Node_Name return POSIX_String;
function Release return POSIX_String;
function Version return POSIX_String;
function Machine return POSIX_String;
```

2.4.7.2 Description

The function `System_Name` shall return the name of this implementation of the operating system. The function `Node_Name` shall return the name of this node within an implementation-specified communications network. The function `Release` shall return the current release level of this implementation. The function `Version` shall return the current version level of this release. The function `Machine` shall return the name of the hardware type on which the system is running.

2.4.7.3 Error Handling

No exceptions are specified by this standard for `System_Name`, `Node_Name`, `Release`, `Version`, and `Machine`.

2.4.8 Time Types

2.4.8.1 Synopsis

```
type Seconds is range implementation-defined;
-- must include at least -(2**31 - 1) .. (2**31 - 1)
type Minutes is range implementation-defined;
-- must include at least -(2**31 - 1) .. (2**31 - 1)
type Nanoseconds_Base is range implementation-defined;
-- must include at least -(2**31 - 1) .. (2**31 - 1)
subtype Nanoseconds is Nanoseconds_Base range 0 .. (10**9)-1;
type Timespec is private;
function Get_Seconds (Time : Timespec) return Seconds;
procedure Set_Seconds
   (Time : in out Timespec;
    S :    in Seconds);
function Get_Nanoseconds (Time : Timespec) return Nanoseconds;
procedure Set_Nanoseconds
   (Time : in out Timespec;
    NS :   in Nanoseconds);
procedure Split
   (Time : in  Timespec;
    S :    out Seconds;
    NS :   out Nanoseconds);
function To_Timespec
   (S :     Seconds;
    NS :    Nanoseconds)
  return Timespec;
function "+" (Left, Right : Timespec) return Timespec;
function "+" (Left : Timespec; Right : Nanoseconds) return Timespec;
function "-" (Right : Timespec) return Timespec;
function "-" (Left, Right : Timespec) return Timespec;
function "-" (Left : Timespec; Right : Nanoseconds) return Timespec;
function "*" (Left : Timespec; Right : Integer) return Timespec;
function "*" (Left : Integer; Right : Timespec) return Timespec;
function "/" (Left : Timespec; Right : Integer) return Timespec;
function "/" (Left, Right : Timespec) return Integer;
function "<" (Left, Right : Timespec) return Boolean;
function "<="(Left, Right : Timespec) return Boolean;
function ">" (Left, Right : Timespec) return Boolean;
function ">="(Left, Right : Timespec) return Boolean;
function To_Duration (Time : Timespec) return Duration;
function To_Timespec (D : Duration) return Timespec;
```

2.4.8.2 Description

The time types are used together to represent potentially large amounts of time with nanosecond precision.

Nanoseconds is a subtype that is interpreted as a nonnegative count of nanoseconds, amounting to less than one second. Nanoseconds_Base is the base type of this subtype, which includes negative values and nanosecond counts amounting to more than one second. The Nanoseconds_Base type is provided for situations where the application needs to perform arithmetic on nanosecond counts and the intermediate results may need to be outside the range of Nanoseconds.

Parameters of the type Timespec are used for some operations to represent relative times, that is, the lengths of time intervals. Timespec is also used to specify absolute

times, as an offset value from a time of day (usually the Epoch, *e.g.*, see `Clock_Realtime` in 14.1.4). The range of time offsets expressible by this type shall include at least `-(`2^{31}`-1) .. (`2^{31}`-1)` seconds.

The resolution of `Timespec` is implementation-defined, subject to the constraint that it be no coarser than `POSIX_Limits.Portable_Clock_Resolution_Minimum`.

NOTE: `Timespec` may be represented so that an uninitialized object of this type has a value that cannot be interpreted as a valid time.

`Split` converts a value of the type `Timespec` into two components. The component `S` is a (signed) count of whole seconds, and the component `NS` is a count of nanoseconds in the range 0 .. $10^9 - 1$. If `Time` is a valid `Timespec` value representing an actual time $time$, the values obtained from `Split(Time, S, NS)` shall satisfy the relationship $time = s + ns \cdot 10^{-9}$, where s is the value returned in `S` and ns is the value returned in `NS`.

The functions `Get_Seconds` and `Get_Nanoseconds` shall return the same values as are returned by `Split` in the `S` and `NS` parameters, respectively.

NOTE: The canonical representation of negative values of type `Timespec` may be surprising. To represent -0.3 *seconds,* $s = -1$ *and* $ns = 700\_000\_000$.

`To_Timespec` with parameters `S` and `NS` converts a count of whole seconds and a count of nanoseconds to a corresponding value of the type `Timespec`. The effect shall be the inverse of `Split`, when the source value can be represented exactly in the target type. Moreover, within the resolution of `Timespec` there shall be no cumulative error from repeated conversions. In other words,

— If `T = To_Timespec(S_1, NS_1)`, then after `Split(T, S_2, NS_2)`, `To_Timespec(S_2, NS_2) = T`.

— After `Split(T, S_1, NS_1)`, and `Split(To_Timespec(S_1, NS_1),S_2, NS_2)`, `S_1=S_2` and `NS_1=NS_2`, to the accuracy of the underlying representation.

`To_Timespec` with parameter `D` converts a value of the type `Duration` to a value of the type `Timespec` if the value belongs to the range of time offsets that can be expressed in that type. Otherwise, it raises `Constraint_Error`.

`To_Duration` converts a value of the type `Timespec` to a value of the type `Duration`, if the value belongs to the range of time offsets that can be expressed in that type. Otherwise, it raises `Constraint_Error`.

All conversions to and from `Timespec` shall be accurate to the limit of precision of the target type.

2.4.8.3 Error Handling

`Constraint_Error` shall be raised by the conversion functions if the value of the argument is outside the range that can be converted to the return type.

These operations shall not raise an exception under any other conditions.

2.5 Package POSIX_Options

This package provides types, constants, and operations that can be used to discover which options are supported by an implementation of this standard.

```
with POSIX;
package POSIX_Options is

   subtype Asynchronous_IO_Support       is Boolean range implementation-defined;
   subtype Change_Owner_Restriction      is POSIX.Change_Owner_Restriction;
   subtype Filename_Truncation           is POSIX.Filename_Truncation;
   subtype File_Synchronization_Support  is Boolean range implementation-defined;
   subtype Internet_Datagram_Support     is Boolean range implementation-defined;
   subtype Internet_Protocol_Support     is Boolean range implementation-defined;
   subtype Internet_Stream_Support       is Boolean range implementation-defined;
   subtype ISO_OSI_Protocol_Support      is Boolean range implementation-defined;
   subtype Job_Control_Support           is POSIX.Job_Control_Support;
   subtype Memory_Mapped_Files_Support   is Boolean range implementation-defined;
   subtype Memory_Locking_Support        is Boolean range implementation-defined;
   subtype Memory_Range_Locking_Support  is Boolean range implementation-defined;
   subtype Memory_Protection_Support     is Boolean range implementation-defined;
   subtype Message_Queues_Support        is Boolean range implementation-defined;
   subtype Mutex_Priority_Ceiling_Support
                                         is Boolean range implementation-defined;
   subtype Mutex_Priority_Inheritance_Support
                                         is Boolean range implementation-defined;
   subtype Mutexes_Support               is Boolean range implementation-defined;
   subtype Network_Management_Support    is Boolean range implementation-defined;
   subtype OSI_Connectionless_Support    is Boolean range implementation-defined;
   subtype OSI_Connection_Support        is Boolean range implementation-defined;
   subtype OSI_Minimal_Support           is Boolean range implementation-defined;
   subtype Poll_Support                  is Boolean range implementation-defined;
   subtype Prioritized_IO_Support        is Boolean range implementation-defined;
   subtype Priority_Process_Scheduling_Support
                                         is Boolean range implementation-defined;
   subtype Priority_Task_Scheduling_Support
                                         is Boolean range implementation-defined;
   subtype Process_Shared_Support        is Boolean range implementation-defined;
   subtype Realtime_Signals_Support      is Boolean range implementation-defined;
   subtype Saved_IDs_Support             is POSIX.Saved_IDs_Support;
   subtype Select_Support                is Boolean range implementation-defined;
   subtype Semaphores_Support            is Boolean range implementation-defined;
   subtype Shared_Memory_Objects_Support is Boolean range implementation-defined;
   subtype Signal_Entries_Support        is Boolean range implementation-defined;
   subtype Sockets_DNI_Support           is Boolean range implementation-defined;
   subtype Synchronized_IO_Support       is Boolean range implementation-defined;
   subtype Timers_Support                is Boolean range implementation-defined;
   subtype XTI_DNI_Support               is Boolean range implementation-defined;

end POSIX_Options;
```

2.5.1 Implementation Options

Support for some of the facilities defined by this standard is optional. A Strictly Conforming POSIX.5 Application may adjust its execution, based on information provided by interfaces defined in this standard, to take advantage of optional features that are supported by a particular implementation.

The names and general descriptions of the optional facilities are listed in this subclause. These names are used in the rest of this standard to specify the functionality that depends on each option. Generally, an option is either supported or not supported on a system-wide basis. Support for some options, related to operations on files, is specified as *pathname-specific*. This means that the option may be supported for some files and not for other files, depending on the pathname of the file and the time that the file is accessed.

Asynchronous I/O

The implementation supports asynchronous input and output operations. This option is pathname-specific.

Change Owner Restriction

The procedure `Change_Owner_And_Group` is restricted to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs. Changing the owner ID is restricted to processes with the use privilege. If this option is not supported, no such restriction applies. This option is pathname-specific.

Filename Truncation

Filenames are truncated to the maximum allowed length in POSIX characters, Filename Maximum. If this option is not supported, and a filename is longer than allowed, an error shall result, as defined by the subprograms that use filenames and pathnames. This option is pathname-specific.

File Synchronization

The implementation supports file synchronization operations.

Internet Datagram

The implementation supports an interface to the connectionless-mode Internet transport protocol (*i.e.*, UDP). If the implementation supports this option, it shall also support the Internet Protocol option.

Internet Protocol

The implementation supports an interface to the Internet family of protocols. If the implementation supports this option, it shall also support the XTI Detailed Network Interface option and/or the Sockets Detailed Network Interface option.

Internet Stream

The implementation supports an interface to the connection-mode Internet transport protocol (*i.e.*, TCP). If the implementation supports this option, it shall also support the Internet Protocol option.

ISO/OSI Protocol

The implementation supports an interface to the ISO/OSI family of protocols. If the implementation supports this option, it shall also support the XTI Detailed Network Interface option and/or the Sockets Detailed Network Interface option.

Job Control

The implementation supports job control signals.

Memory Mapped Files

The implementation supports the mapping of files to the address space of a process.

Memory Locking

The implementation supports locking all or part the address space of a process to be continually resident in memory.

Memory Range Locking

The implementation supports locking regions of the address space of a process so that they are continually resident in memory. If this option is supported, the Memory Locking option shall be supported also.

Memory Protection

The implementation supports protecting regions of the address space of a process from read, write, or execute access. If this option is supported, then at least one of the Memory Mapped Files option or the Shared Memory Objects option shall be supported also.

Message Queues

The implementation supports message queues.

Mutex Priority Ceiling

The implementation temporarily raises the priority of a task that holds a mutex to the specified ceiling priority of the mutex.

NOTE: The utility of this option depends on support for the Mutexes option.

Mutex Priority Inheritance

The implementation temporarily raises the priority of a task that holds a mutex to the highest priority of the tasks that are blocked waiting for the mutex.

NOTE: The utility of this option depends on support for the Mutexes option.

Mutexes

The implementation supports mutex and condition variable synchronization objects.

Network Management

The implementation supports network information management services.

OSI Connectionless

The implementation supports an interface to the connectionless-mode of transport service provided by the ISO/OSI family of protocols (*i.e.*, ISO/IEC 8602ISO/IEC 8602 {9}). If the implementation supports this option, it shall also support the ISO/OSI Protocol option.

OSI Connection

The implementation supports an interface to the connection mode of transport service provided by the ISO/OSI family of protocols (*i.e.*, ISO/IEC 8073ISO/IEC 8073 {4}). If the implementation supports this option, it shall also support the ISO/OSI Protocol option. c

OSI Minimal

The implementation supports an interface to the ISO/OSI minimal 7-layer OSI stack (*i.e.,* CULR Part 3). If the implementation supports this option, it shall also support the ISO/OSI Protocol option and either the OSI Connection option or the OSI Connectionless option.

Poll

The implementation supports poll for event management. C

Prioritized I/O

The implementation supports prioritized input and output. This option is pathname-specific.

NOTE: The utility of this option depends on support for the Asynchronous I/O option and for the Priority Process Scheduling option.

Priority Process Scheduling

The implementation supports the process scheduling interfaces defined by this standard.

Priority Task Scheduling

The implementation supports the extended task scheduling interfaces defined by this standard.

NOTE: A prerequisite for support of this option is support for the priority model defined in D.1 of the Ada RM {1} and the pragmas and package interfaces defined in D.2-D.5 of the Ada RM {1}. (See 13.3.)

Process Shared

The implementation supports sharing of mutexes and condition variables between processes, in a shared memory object or memory-mapped file.

NOTE: The utility of this option depends on support for the Mutexes option.

Realtime Signals

The implementation supports the realtime signals, for which queueing, signal information, and notification in priority order are defined.

Saved IDs

The implementation supports a *saved set-user-ID* and a *saved set-group-ID* for each process. If this option is not supported, there is neither a saved set-user-ID nor a saved set-group-ID.

Select

The implementation supports select for event management. C

Semaphores

The implementation supports counting semaphores.

Shared Memory Objects

The implementation supports shared memory objects.

Signal Entries

The implementation supports binding task entries to signals. (See 3.3.17.) The range of the static subtype for this option shall be either `True..True` or `False..False`.

Sockets Detailed Network Interface

The implementation supports the sockets set of DNI for protocol-independent, process-to-process network communications. c

Synchronized I/O

The implementation supports I/O synchronization, which permits a process to force I/O operations to synchronized I/O data integrity completion or synchronized I/O file integrity completion. If this option is supported, then the File Synchronization option shall be supported also. This option is pathname-specific.

Timers

The implementation supports clocks and timers.

XTI Detailed Network Interface

The implementation supports the XTI set of DNI for protocol-independent, process-to-process network communications. c

Information about support for an option is provided to the application in the following forms:

Static subtype

Subtypes, in most cases with names ending with `_Support`, declared in the package `POSIX_Options`, document the implementation-defined possible range of support for the optional feature. The correspondence of options to subtypes shall be as shown in Table 2.4.

These subtypes shall all be static.

Configurable system option

The functions with the same names without the `_Support` suffix declared in the package `POSIX_Configurable_System_Limits` and described in 4.5 define the current system-wide support for the option.

NOTE: The value returned by the function at run time is constrained to be within the static range imposed by the corresponding static subtype.

Configurable pathname variable options

The functions with the same names without the `_Support` suffix declared in the package `POSIX_Configurable_File_Limits` and described in 5.4 define the pathname-dependent support for the option.

NOTE: The value returned by the function at run time is constrained to be within the static range imposed by the corresponding subtype.

The subtypes in package `POSIX_Options` and the correspondingly named functions in the packages `POSIX_Configurable_System_Limits` and `POSIX_Configurable_File_Limits` define one of the following four possible conditions:

(1) If the range is `False..False`, the option is not supported. The corresponding function shall always return `False`.

(2) If the range is `False..True` and the corresponding function returns `False`, the option is not supported.

(3) If the range is `False..True` and the corresponding function returns `True`, the option is supported.

Table 2.4 – Static Subtypes and Options

Subtype	Option	
`Asynchronous_IO_Support`	Asynchronous I/O	
`Change_Owner_Restriction`	Change Owner Restriction	
`File_Synchronization_Support`	File Synchronization	
`Filename_Truncation`	Filename Truncation	
`Job_Control_Support`	Job Control	
`Internet_Datagram_Support`	Internet Datagram	c
`Internet_Protocol_Support`	Internet Protocol	
`Internet_Stream_Support`	Internet Stream	
`ISO_OSI_Protocol_Support`	ISO/OSI Protocol	
`Memory_Mapped_Files_Support`	Memory Mapped Files	
`Memory_Locking_Support`	Memory Locking	
`Memory_Range_Locking_Support`	Memory Range Locking	
`Memory_Protection_Support`	Memory Protection	
`Message_Queues_Support`	Message Queues	
`Mutex_Priority_Ceiling_Support`	Mutex Priority Ceiling	
`Mutex_Priority_Inheritance_Support`	Mutex Priority Inheritance	
`Mutexes_Support`	Mutexes	
`Network_Management_Support`	Network Management	c
`OSI_Connectionless_Support`	OSI Connectionless	
`OSI_Connection_Support`	OSI Connection	
`OSI_Minimal_Support`	OSI Minimal	
`Poll_Support`	Poll	
`Prioritized_IO_Support`	Prioritized I/O	
`Priority_Process_Scheduling_Support`	Priority Process Scheduling	
`Priority_Task_Scheduling_Support`	Priority Task Scheduling	
`Process_Shared_Support`	Process Shared	
`Realtime_Signals_Support`	Realtime Signals	
`Saved_IDs_Support`	Saved IDs	
`Select_Support`	Select	c
`Semaphores_Support`	Semaphores	
`Shared_Memory_Objects_Support`	Shared Memory Objects	
`Signal_Entries_Support`	Signal Entries	
`Sockets_DNI_Support`	Sockets Detailed Network Interface	c
`Synchronized_IO_Support`	Synchronized I/O	
`Timers_Support`	Timers	
`XTI_DNI_Support`	XTI Detailed Network Interface	c

(4) If the range is `True..True`, the option is always supported. The corresponding function shall always return `True`.

NOTE: One kind of implementation option is not represented in this package. A POSIX operation that blocks the calling task may also block other tasks, depending on the implementation. There are several options, corresponding to the different blocking operations. The type, subtype, and constant declarations related to blocking behavior are in the package `POSIX` *(2.4.1.5). The main reason they are not with the other implementation options in package* `POSIX_Options` *is that the subtype* `Text_IO_Blocking_Behavior` *is also used as the type of a* user-*selectable parameter of some operations defined in the package* `POSIX_Supplement_to_Ada_IO` *(see Section 8).*

2.6 Package POSIX_Limits

This package provides types and constants that can be used to discover the capacity limits that are imposed by an implementation of this standard.

```
with POSIX;
package POSIX_Limits is
   -- Portable System Limits
   Portable_Argument_List_Maximum : Natural
      renames POSIX.Portable_Argument_List_Maximum;
   Portable_Asynchronous_IO_Maximum : constant Natural := 1;
   Portable_Child_Processes_Maximum : Natural
      renames POSIX.Portable_Child_Processes_Maximum;
   Portable_Clock_Resolution_Minimum : constant := 20_000_000; -- nanoseconds
   Portable_Filename_Maximum : Natural
      renames POSIX.Portable_Filename_Limit_Maximum;
   Portable_Groups_Maximum : Natural
      renames POSIX.Portable_Groups_Maximum;
   Portable_Input_Line_Maximum : POSIX.IO_Count
      renames POSIX.Portable_Input_Line_Limit_Maximum;
   Portable_Input_Queue_Maximum : POSIX.IO_Count
      renames POSIX.Portable_Input_Queue_Limit_Maximum;
   Portable_Links_Maximum : Natural
      renames POSIX.Portable_Link_Limit_Maximum;
   Portable_List_IO_Maximum : constant Natural := 2;
   Portable_Message_Priority_Maximum : constant Natural := 32;
   Portable_Open_Files_Maximum : Natural
      renames POSIX.Portable_Open_Files_Maximum;
   Portable_FD_Set_Maximum :
      constant Natural := Portable_Open_Files_Maximum;                    c
   Portable_Open_Message_Queues_Maximum : constant Natural := 8;
   Portable_Pathname_Maximum : Natural
      renames POSIX.Portable_Pathname_Limit_Maximum;
   Portable_Pipe_Length_Maximum : POSIX.IO_Count
      renames POSIX.Portable_Pipe_Limit_Maximum;
   Portable_Queued_Signals_Maximum : constant Natural := 32;
   Portable_Realtime_Signals_Maximum : constant Natural := 8;
   Portable_Semaphores_Maximum : constant Natural := 256;
   Portable_Semaphores_Value_Maximum : constant Natural := 32_767;
   Portable_Socket_Buffer_Maximum :
      constant POSIX.IO_Count := Portable_Pipe_Length_Maximum;
   Portable_Socket_IO_Vector_Maximum : constant Natural := 16;
   Portable_Socket_Connection_Maximum : constant Natural := 1;             c
   Portable_Streams_Maximum : Natural
      renames POSIX.Portable_Stream_Maximum;
   Portable_Timer_Overruns_Maximum : constant Natural := 32;
   Portable_Timers_Maximum : constant Natural := 32;
   Portable_Time_Zone_String_Maximum : Natural
      renames POSIX.Portable_Time_Zone_String_Maximum;
   Portable_XTI_IO_Vector_Maximum : constant Natural := 16;                c
   -- Configurable Limits
   subtype Argument_List_Maxima is POSIX.Argument_List_Maxima;
   subtype Asynchronous_IO_Maxima is Natural range implementation-defined;
   subtype Asynchronous_IO_Priority_Delta_Maxima is Natural
      range implementation-defined;
   subtype Child_Processes_Maxima is POSIX.Child_Processes_Maxima;
   subtype FD_Set_Maxima is Natural range
      Portable_FD_Set_Maximum .. implementation-defined;                   c
```

```
   subtype Filename_Maxima is POSIX.Filename_Limit_Maxima;
   subtype Groups_Maxima is POSIX.Groups_Maxima;
   subtype Input_Line_Maxima is POSIX.Input_Line_Limit_Maxima;
   subtype Input_Queue_Maxima is POSIX.Input_Queue_Limit_Maxima;
   subtype Links_Maxima is POSIX.Link_Limit_Maxima;
   subtype List_IO_Maxima is Natural range implementation-defined;
   subtype Open_Message_Queues_Maxima is Natural range implementation-defined;
   subtype Message_Priority_Maxima is Natural range implementation-defined;
   subtype Open_Files_Maxima is POSIX.Open_Files_Maxima;
   subtype Page_Size_Range is Natural range 1 .. implementation-defined;
   subtype Pathname_Maxima is POSIX.Pathname_Limit_Maxima;
   subtype Pipe_Length_Maxima is POSIX.Pipe_Limit_Maxima;
   subtype Queued_Signals_Maxima is Natural range implementation-defined;
   subtype Realtime_Signals_Maxima is Natural range implementation-defined;
   subtype Semaphores_Maxima is Natural range implementation-defined;
   subtype Semaphores_Value_Maxima is Natural range implementation-defined;
   subtype Socket_Buffer_Maxima is POSIX.IO_Count range
      Portable_Socket_Buffer_Maximum .. implementation-defined;
   subtype Socket_IO_Vector_Maxima is Natural range
      Portable_Socket_IO_Vector_Maximum .. implementation-defined;
   subtype Socket_Connection_Maxima is Natural range
      Portable_Socket_Connection_Maximum .. implementation-defined;
   subtype Streams_Maxima is POSIX.Stream_Maxima;
   subtype Timer_Overruns_Maxima is Natural range implementation-defined;
   subtype Timers_Maxima is Natural
      range implementation-defined;
   subtype Time_Zone_String_Maxima is POSIX.Time_Zone_String_Maxima;
   subtype XTI_IO_Vector_Maxima is Natural range
      Portable_XTI_IO_Vector_Maximum .. implementation-defined;
end POSIX_Limits;
```

2.6.1 Implementation Limits

This standard defines certain capacity limits that may be imposed by an implementation. A Strictly Conforming POSIX.5 Application may adjust its execution, based on information provided by interfaces defined in this standard, to operate within the limitations of a particular implementation.

The names and general descriptions of the limits are listed in this subclause. These names are used in the rest of this standard to specify the functionality that depends on each limit. Generally, the value of a limit is system wide. For some limits related to files, the value is *pathname-specific*. In other words, the value of the limit may be different for some file than for others depending on the pathname of the file and the time that the file is accessed.

Argument List Maximum

The maximum total length of the argument list, environment data, and any associated overhead (in bytes) that can be passed to a child when it is started by the `POSIX_Process_Primitives` procedures `Start_Process` and `Start_Process_Search`, and the `POSIX_Unsafe_Process_Primitives` procedures `Exec` and `Exec_Search`.

Asynchronous I/O Maximum

The maximum number of outstanding AIO operations.

Asynchronous I/O Priority Delta Maximum

The maximum amount by which a task can decrease the priority of an AIO operation below the process priority. (See 6.3.1).

Child Processes Maximum

The maximum number of simultaneous processes per real user ID.

Clock Resolution Minimum

The minimum resolution of the system realtime clock. (See `Clock_Realtime` in 14.1.2.)

File Descriptor Set Maximum

The maximum number of file descriptors that may be examined with `Select_File` operations.

Filename Maximum

The maximum length of a filename, measured in POSIX characters. This limit is pathname-specific.

Groups Maximum

The maximum number of simultaneous supplementary group IDs per process. Zero means that this feature is not supported.

Input Line Maximum

The maximum length of an input line, measured in bytes. This limit is pathname-specific.

Input Queue Maximum

The maximum length of a terminal input queue, measured in bytes. This limit is pathname-specific.

Links Maximum

The maximum value of the link count of a file. This limit is pathname-specific.

List I/O Maximum

The maximum number of I/O operations that can be specified in a list I/O call.

Message Priority Maximum

The maximum number of message priority levels supported.

Open Files Maximum

The maximum number of files a process can have open at one time.

Open Message Queues Maximum

The maximum number of message queues that a process can have open at one time.

Page Size

Granularity, measured in storage units, of memory mapping and process memory locking.

NOTE: The page size is measured in storage units, rather than bytes, for consistency with the address arithmetic defined in `System.Storage_Elements` *in case the system storage unit is not one byte.*

Pathname Maximum

The maximum length of a pathname, in POSIX characters. This limit is pathname-specific.

Pipe Length Maximum

The number of bytes that can be written atomically when writing to a pipe or FIFO. This limit is pathname-specific.

Queued Signals Maximum

The maximum number of queued signals a single process can send and have pending at the receiver(s) at any one time.

Realtime Signals Maximum

The maximum number of realtime signal numbers reserved for application use.

Semaphores Maximum

The maximum number of semaphores that a process can have.

Semaphores Value Maximum

The maximum value a semaphore can have.

Socket Buffer Maximum

The maximum number of bytes that can be buffered on a socket for send or receive.

Socket IO Vector Maximum

The maximum number of `IO_Vector` elements in a `IO_Vector_Array` object.

Socket Queued Connection Maximum

The maximum number of connections that can be queued on a socket. c

Streams Maximum

For an implementation that also supports the C-language binding, the maximum number of C-language streams a process can have open at one time.

Timer Overruns Maximum

The maximum number of timer expiration overruns.

Timers Maximum

The maximum number of timers a process can have.

Time Zone String Maximum

The maximum length of the **TZ** environment variable, measured in POSIX characters.

XTI IO Vector Maximum

The maximum number of `IO_Vector` elements in an XTI `IO_Vector_Array` object. c

Information about limits is provided to an application in the following forms:

Portable constant

These numeric constants, whose names begin with "Portable_", define worst-case bounds of values for a conforming implementation. If the limit is an upper bound (*e.g.*, Argument List Maximum), the portable constant for the limit is the smallest value allowed by this standard for the limit. If the limit is a lower bound (*e.g.*, Clock Resolution Minimum), the portable constant is the largest value allowed by this standard for the limit.

These portable constants are declared in the package POSIX_Limits. The correspondence between portable constants and system limits is given in Table 2.5.

For compatibility with POSIX.5, redundant declarations of some of the constants are retained in package POSIX. However, those declarations in package POSIX are obsolescent.

Table 2.5 – Portable Constants and Limits

Constant	Limit	
Portable_Argument_List_Maximum	Argument List Maximum	
none	Asynchronous I/O Priority Delta Maximum	
none	Page Size	
Portable_Asynchronous_IO_Maximum	Asynchronous I/O Maximum	
Portable_Child_Processes_Maximum	Child Processes Maximum	
Portable_Clock_Resolution_Minimum	Clock Resolution Minimum	
Portable_FD_Set_Maximum	File Descriptor Set Maximum	c
Portable_Filename_Maximum	Filename Maximum	
Portable_Groups_Maximum	Groups Maximum	
Portable_Input_Line_Maximum	Input Line Maximum	
Portable_Input_Queue_Maximum	Input Queue Maximum	
Portable_Links_Maximum	Links Maximum	
Portable_List_IO_Maximum	List I/O Maximum	
Portable_Message_Priority_Maximum	Message Priority Maximum	
Portable_Open_Files_Maximum	Open Files Maximum	
Portable_Open_Message_Queues_Maximum	Open Message Queues Maximum	
Portable_Pathname_Maximum	Pathname Maximum	
Portable_Pipe_Length_Maximum	Pipe Length Maximum	
Portable_Queued_Signals_Maximum	Queued Signals Maximum	
Portable_Realtime_Signals_Maximum	Realtime Signals Maximum	
Portable_Semaphores_Maximum	Semaphores Maximum	
Portable_Semaphores_Value_Maximum	Semaphores Value Maximum	
Portable_Socket_Buffer_Maximum	Socket Buffer Maximum	
Portable_Socket_IO_Vector_Maximum	Socket IO Vector Maximum	
Portable_Socket_Connection_Maximum	Socket Queued Connect Maximum	c
Portable_Streams_Maximum	Streams Maximum	
Portable_Timer_Overruns_Maximum	Timer Overruns Maximum	
Portable_Timers_Maximum	Timers Maximum	
Portable_Time_Zone_String_Maximum	Time Zone String Maximum	
Portable_XTI_IO_Vector_Maximum	XTI IO Vector Maximum	c

Static subtype

These subtypes, ending with ..._Maxima, document the implementation-

defined range of possible values for the corresponding limit to the extent the value can be bounded at compile time. The 'First and 'Last values of the subtype shall be static. For maxima, the 'First value of the subtype shall be greater than or equal to the portable constant. If the 'First value of the subtype is equal to its 'Last, the value of the corresponding limit shall always be equal to that constant.

These static subtypes are declared in the package POSIX_Limits. The correspondence between static subtypes and system limits is given in Table 2.6.

For compatibility with POSIX.5, redundant definitions of some of the subtypes are retained in the package POSIX. However, those declarations in othe package POSIX are obsolescent.

These subtypes shall all be static.

Configurable system variable

The functions declared in the package POSIX_Configurable_System_Limits and described in 4.5 shall return the actual system-wide limit imposed by the implementation.

Configurable pathname variable

The functions declared in the package POSIX_Configurable_File_Limits and described in 5.4 shall return the pathname-dependent limit imposed by the implementation.

2.7 Package Ada_Streams

```
with Ada.Streams;
package Ada_Streams renames Ada.Streams;
```

During transition from Ada 83 to Ada 95, the implementation is permitted to replace this renaming declaration with the following package specification.

```
package Ada_Streams is
   -- pragma Pure(Streams);
   -- type Root_Stream_Type is abstract tagged limited private;
   type Stream_Element is -- mod
      impl-def-unsigned-integer-type;
   type Stream_Element_Offset is range implementation-defined;
   subtype Stream_Element_Count is
      Stream_Element_Offset range 0 .. Stream_Element_Offset'Last;
   type Stream_Element_Array is
      array (Stream_Element_Offset range <>) of Stream_Element;
   -- procedure Read (
   --    Stream : in out Root_Stream_Type;
   --    Item :   out Stream_Element_Array;
   --    Last :   out Stream_Element_Offset) is abstract;
   -- procedure Write (
   --    Stream : in out Root_Stream_Type;
   --    Item :   out Stream_Element_Array) is abstract;
private
   implementation-defined
end Ada_Streams;
```

Table 2.6 – Static Subtypes and Limits

Subtype	Limit	
`Argument_List_Maxima`	Argument List Maximum	
`Asynchronous_IO_Maxima`	Asynchronous I/O Maximum	
`Asynchronous_IO_Priority_Delta_Maxima`	Asynchronous I/O Priority Delta Maximum	
`Child_Processes_Maxima`	Child Processes Maximum	
none	Clock Resolution Minimum	
`FD_Set_Maxima`	File Descriptor Set Maximum	c
`Filename_Maxima`	Filename Maximum	
`Groups_Maxima`	Groups Maximum	
`Input_Line_Maxima`	Input Line Maximum	
`Input_Queue_Maxima`	Input Queue Maximum	
`Links_Maxima`	Links Maximum	
`List_IO_Maxima`	List I/O Maximum	
`Message_Priority_Maxima`	Message Priority Maximum	
`Open_Files_Maxima`	Open Files Maximum	
`Open_Message_Queues_Maxima`	Open Message Queues Maximum	
`Page_Size_Range`	Page Size	
`Pathname_Maxima`	Pathname Maximum	
`Pipe_Length_Maxima`	Pipe Length Maximum	
`Queued_Signals_Maxima`	Queued Signals Maximum	
`Realtime_Signals_Maxima`	Realtime Signals Maximum	
`Semaphores_Maxima`	Semaphores Maximum	
`Semaphores_Value_Maxima`	Semaphores Value Maximum	
`Socket_Buffer_Maxima`	Socket Buffer Maximum	
`Socket_IO_Vector_Maxima`	Socket IO Vector Maximum	
`Socket_Connection_Maxima`	Socket Queued Connection Maximum	c
`Streams_Maxima`	Streams Maximum	
`Timer_Overruns_Maxima`	Timer Overruns Maximum	
`Timers_Maxima`	Timers Maximum	
`Time_Zone_String_Maxima`	Time Zone String Maximum	
`XTI_IO_Vector_Maxima`	XTI IO Vector Maximum	c

NOTE: Places where details of the Ada 95 package specification have been omitted, because they use features not supported by Ada 83, are indicated by comments in the package specification above.

The semantics of the types and operations defined in this package shall satisfy the requirements of 13.13.1 of the Ada RM {1}.

2.8 Package System

This standard requires the Ada implementation to include the following declarations in the package `System`:

```
Null_Address :  constant Address := implementation-defined;
function "<"  (Left, Right : Address) return Boolean;
function "<=" (Left, Right : Address) return Boolean;
function ">"  (Left, Right : Address) return Boolean;
function ">=" (Left, Right : Address) return Boolean;
```

```
  function Image (Addr : Address) return String;
  function Value (Str : String) return Address;
  Word_Size : constant := implementation-defined;
```

NOTE: All these declarations are required by Ada 95, and are permitted by Ada 83. They are included here to permit this standard to be upward compatible. The detailed semantics are intentionally left unspecified, to avoid conflicts with Ada 95.

The type `System.Address` shall be nonlimited. Thus, the operations `"="` and `"/="` are defined for it.

The value of `System.Null_Address` shall correspond to the value returned by `System_Storage_Elements.To_Address(0)`.

2.9 Package System_Storage_Elements

```
with System.Storage_Elements;
package System_Storage_Elements renames System.Storage_Elements;
```

During transition from Ada 83 to Ada 95 the implementation is permitted to replace this renaming declaration with the following package specification:

```
with System;
package System_Storage_Elements is
  -- pragma Preelaborate (System.Storage_Units);
   -- Storage Elements and Address Arithmetic
  type Storage_Offset is range implementation-defined;
  subtype Storage_Count is Storage_Offset range 0 .. Storage_Offset'Last;
  type Storage_Element is implementation-defined;
  -- required to be an unsigned type whose size is one storage unit
  type Storage_Array is array
     (Storage_Offset range <>) of Storage_Element;
  -- required to have a component size of one storage unit
  -- required to have components aligned on addressable boundaries
  -- components required to be aliased
  function "+"
     (Left :  System.Address;
      Right : Storage_Offset)
    return System.Address;
  function "+"
     (Left :  Storage_Offset;
      Right : System.Address)
    return System.Address;
  function "-"
     (Left :  System.Address;
      Right : Storage_Offset)
    return System.Address;
  function "-"
     (Left, Right: System.Address)
    return Storage_Offset;
  function "mod"
     (Left :  System.Address;
      Right : Storage_Offset)
    return Storage_Offset;
```

```
   -- Conversions to/from Integers
   type Integer_Address is range implementation-defined;
   function To_Address (Value : Integer_Address) return System.Address;
   function To_Integer (Value : System.Address) return Integer_Address;

end System_Storage_Elements;
```

NOTE: Places where details of the Ada 95 package specification have been omitted, because they use features not supported by Ada 83, are indicated by comments.

The semantics of the types and operations defined in this package shall satisfy the requirements of 13.7.1 of the Ada RM {1}.

2.10 Package POSIX_Page_Alignment

This package provides operations that can be used to compute the address of the first page containing a given object, and to compute the length of a page-aligned region containing the object. These operations are provided to simplify the computation of page-aligned specifications for memory range locking (see 12.2) and memory mapping (see 12.3), where the implementation is permitted to require the specified region to be aligned on a page boundary.

```
with System,
     POSIX,
     System_Storage_Elements;
package POSIX_Page_Alignment is
   function Truncate_To_Page (Addr :   System.Address) return System.Address;
   function Truncate_To_Page (Offset : POSIX.IO_Count) return POSIX.IO_Count;
   function Adjust_Length
      (Addr :   System.Address;
       Length : System_Storage_Elements.Storage_Offset)
     return System_Storage_Elements.Storage_Offset;
   function Adjust_Length
      (Offset : POSIX.IO_Count;
       Length : System_Storage_Elements.Storage_Offset)
     return System_Storage_Elements.Storage_Offset;
   function Length
      (Size :  in Natural)
     return System_Storage_Elements.Storage_Offset;

end POSIX_Page_Alignment;
```

2.10.1 Description

The values computed by these functions are specified as follows:

— `Truncate_To_Page(A)` is the greatest integral multiple of Page Size that is less than or equal to A. That is, if A is an address (file offset) `Truncate_To_Page(A)` shall be the address (file offset) of the first page that includes the storage cell with address (offset) A.

`Truncate_To_Page` can be used to compute the address or offset of the first page containing an object, from the exact address or offset of the object itself.

— `Adjust_Length(A, L)` is the least integral multiple of Page Size greater than or equal to `L + (A-Truncate_To_Page(A))`. In other words, the range of cells with addresses `Truncate_To_Page (A) .. Truncate_To_Page (A) + Adjust_-Length (A, L) -1` shall be the smallest set of complete pages that includes the range of cells with addresses (offsets) `A..A+L`.

`Adjust_Length` can be used to compute the length of a region starting on a page boundary that contains a given object, given the exact (not necessarily page0aligned) address or offset of the object and the exact length of the object itself.

— `Length(B)` is the least integer greater than or equal to `B/System.Storage_-Unit`.

2.10.2 Error Handling

Any of these operations returning an address is permitted to raise `Program_Error` if no reasonable interpretation of the requested operation exists on the machine.

2.11 Environment Description

A list of (name, value) pairs called the *environment*, where the components are both of type `POSIX_String`, is made available when a process begins. This list is processed using the operations of the package `POSIX_Process_Environment`. (See 4.3.2.) The name of an environment variable shall not contain the character `'='`. No meaning is associated with the order of the strings in the environment. If more than one string in the environment of a process has the same *name*, which occurrence is used is unspecified. The following names may be defined and have the indicated meaning if they are defined:

HOME The name of the initial working directory of the user, from the user database.

LOGNAME The login name associated with the current process. The value shall be composed of characters from the portable filename character set.

NOTE: An application that requires, or an installation that actually uses, characters outside the portable filename character set would not strictly conform to this standard. However, it is reasonable to expect that such characters would be used in many countries (recognizing the reduced level of interchange implied by using characters outside the portable filename character set), and applications or installations should permit such usage where possible. No error is defined by this standard for violation of this condition.

PATH The sequence of path prefixes that certain functions apply in searching for an executable file known only by a filename (a pathname that does not contain a slash). The prefixes are separated by a colon (**:**). When a nonzero-length prefix is applied to this filename, a slash is inserted between the prefix and the filename. A zero-length prefix is a special prefix that indicates the current working directory. It appears as two adjacent colons (“**::**”), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. The list is searched from beginning to end until an executable program by the specified name is found. If the pathname being sought contains a slash, the search through the path prefixes is not performed.

TERM The terminal type for which output is to be prepared. This information is used by commands and application programs wishing to exploit special capabilities specific to a terminal.

TZ Time-zone information. The format for the value of the environment variable is defined in 2.11.1.

LANG, **LC_ALL**,
LC_COLLATE,
LC_CTYPE,
LC_MONETARY,
LC_NUMERIC, **LC_TIME**

These environment variable names are defined by POSIX.1. They have no predefined meaning in this standard.

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

The *values* that the environment variables may be assigned are not restricted except that they shall not contain the null character. The total space used by the environment data, the argument list, and any associated overhead is subject to implementation limits when given to a process. (See 2.6.1.)

2.11.1 Time-Zone Information

The value of the environment variable **TZ** shall be used by the package `POSIX_Calendar` to override the default time zone. (See 4.4.) The value of **TZ** shall be of one of the two following forms (spaces inserted for clarity):

(1) :characters

(2) *std offset dst offset, rule*

If **TZ** is of the first format (*i.e.*, if the first character is a colon), the characters following the colon are handled in an implementation-defined manner.

The expanded format (for all **TZ** values whose value does not have a colon as the first character) is as follows:

stdoffset[*dst*[*offset*][,*start*[/*time*],*end*[/*time*]]]

where:

std and *dst* Indicate no less than three, nor more than Time Zone String Maximum, `POSIX_Character`s that are the designation for the standard *(std)* or summer *(dst)* time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:) or digits, the comma (,), the minus (-), the plus (+), and the null character are permitted to appear in these fields, but their meaning is unspecified.

offset Indicates the value one must add to the local time to arrive at Universal Coordinated Time (see Blair {B2}). The *offset* has the form

hh[:*mm*[:*ss*]]

The minutes *(mm)* and seconds *(ss)* are optional. The hour *(hh)* shall be required and may be a single digit. The *offset* following *std* shall be required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour shall be in the range from 0 to 24, and the minutes (and seconds)—if present—shall be in the range from 0 to 59. Use of values outside these ranges causes undefined behavior. If preceded by a minus(-), the time zone shall be east of the Prime Meridian; otherwise, it shall be west (which may be indicated by an optional preceding plus(+).

rule Indicates when to change to and back from summer time. The *rule* has the form

date/*time*,*date*/*time*

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* shall be one of the following:

`J`n The Julian day $n (1 <= n <= 365)$. Leap days shall not be counted. That is, in all years—including leap years—February 28 is day 59 and March 1 is day 60. It is impossible to refer to the occasional February 29 explicitly.

n The zero-based Julian day $(0 <= n <= 365)$. Leap days shall be counted, and it is possible to refer to February 29.

`M`$m.n.d$ The dth day $(0 <= d <= 6)$ of week n of month m of the year $(1 <= n <= 5, 1 <= m <= 12$, where week 5 means "the last d day in month m," which may occur in either the fourth or the fifth week). Week 1 is the first week in which the dth day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign ("-" or "+") shall be allowed. The default, if *time* is not given, shall be `02:00:00`.

Section 3: Process Primitives

This section specifies services dealing with processes and interprocess signals, as defined in the three packages POSIX_Process_Primitives, POSIX_Unsafe_Process_Primitives, and POSIX_Signals. An execution of an Ada active partition corresponds to a POSIX process. The programs executed by POSIX processes may be written in Ada and in other programming languages supported by the underlying operating system. An Ada program can use the types and operations provided in this section to create, communicate with, and destroy other POSIX processes, including executions of programs written in other languages.

3.1 Package POSIX_Process_Primitives

This package provides the types and operations that allow an application to start a new process and to terminate a process, including operations used by a parent process to find out about the termination of a child process.

A single Ada active partition shall appear to the POSIX/Ada language interface as a single process. However, an implementation of the POSIX/Ada language interface may implement a single Ada active partition execution by allocating it across multiple processes in the underlying operating system; in this case, such an allocation shall be hidden by the interface.

The only form of main subprogram that is required to be supported by all Ada language implementations is a public parameterless library procedure (see 10.2 (29) of the Ada RM {1}). Therefore, a Strictly Conforming POSIX.5 Application shall have a public parameterless library procedure as the main subprogram.

```
with POSIX,
     POSIX_IO,
     POSIX_Permissions,
     POSIX_Process_Environment,
     POSIX_Process_Identification,
     POSIX_Signals;
package POSIX_Process_Primitives is
    -- 3.1.1 Process Template
   type Process_Template is limited private;
   procedure Open_Template  (Template : in out Process_Template);
   procedure Close_Template (Template : in out Process_Template);
   procedure Set_Keep_Effective_IDs (Template : in out Process_Template);
   procedure Set_Signal_Mask
      (Template : in out Process_Template;
       Mask :     in     POSIX_Signals.Signal_Set);
   procedure Set_Creation_Signal_Masking
      (Template :       in out Process_Template;
       Masked_Signals : in     POSIX.Signal_Masking := POSIX.RTS_Signals);
   procedure Set_File_Action_To_Open
      (Template : in out Process_Template;
       File :     in     POSIX_IO.File_Descriptor;
       Name :     in     POSIX.Pathname;
       Mode :     in     POSIX_IO.File_Mode := POSIX_IO.Read_Only;
       Options :  in     POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set);
   procedure Set_File_Action_To_Close
      (Template : in out Process_Template;
       File :     in     POSIX_IO.File_Descriptor);
```

```
procedure Set_File_Action_To_Duplicate
   (Template :  in out Process_Template;
    File :      in     POSIX_IO.File_Descriptor;
    From_File : in     POSIX_IO.File_Descriptor);
-- 3.1.2 Process Creation
procedure Start_Process
   (Child :    out POSIX_Process_Identification.Process_ID;
    Pathname : in  POSIX.Pathname;
    Template : in  Process_Template;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
procedure Start_Process
   (Child :    out POSIX_Process_Identification.Process_ID;
    Pathname : in  POSIX.Pathname;
    Template : in  Process_Template;
    Env_List : in  POSIX_Process_Environment.Environment;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
procedure Start_Process_Search
   (Child :    out POSIX_Process_Identification.Process_ID;
    Filename : in  POSIX.Filename;
    Template : in  Process_Template;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
procedure Start_Process_Search
   (Child :    out POSIX_Process_Identification.Process_ID;
    Filename : in  POSIX.Filename;
    Template : in  Process_Template;
    Env_List : in  POSIX_Process_Environment.Environment;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
-- 3.1.3 Process Exit
type Exit_Status is range 0 .. 2**8-1;
Normal_Exit :              constant Exit_Status := 0;
Failed_Creation_Exit :     constant Exit_Status := 41;
Unhandled_Exception_Exit : constant Exit_Status := 42;
procedure Exit_Process (Status : in Exit_Status := Normal_Exit);
-- 3.1.4 Termination Status
type Termination_Status is private;
type Termination_Cause is
   (Exited, Terminated_By_Signal, Stopped_By_Signal);
function Status_Available (Status : Termination_Status)
   return Boolean;
function Process_ID_Of (Status : Termination_Status)
   return POSIX_Process_Identification.Process_ID;
function Termination_Cause_Of (Status : Termination_Status)
   return Termination_Cause;
function Exit_Status_Of (Status : Termination_Status)
   return Exit_Status;
function Termination_Signal_Of (Status : Termination_Status)
   return POSIX_Signals.Signal;
function Stopping_Signal_Of (Status : Termination_Status)
   return POSIX_Signals.Signal;
-- 3.1.5 Wait for Process Termination
procedure Wait_For_Child_Process
   (Status : out Termination_Status;
    Block :  in  Boolean := True;
    Trace_Stopped :  in Boolean := True;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

```
   procedure Wait_For_Child_Process
      (Status : out Termination_Status;
       Child :  in  POSIX_Process_Identification.Process_ID;
       Block :  in  Boolean := True;
       Trace_Stopped :  in Boolean := True;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
   procedure Wait_For_Child_Process
      (Status : out Termination_Status;
       Group :  in  POSIX_Process_Identification.Process_Group_ID;
       Block :  in  Boolean := True;
       Trace_Stopped :  in Boolean := True;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
private
   implementation-defined
end POSIX_Process_Primitives;
```

3.1.1 Process Template

3.1.1.1 Synopsis

```
   type Process_Template is limited private;
   procedure Open_Template  (Template : in out Process_Template);
   procedure Close_Template (Template : in out Process_Template);
   procedure Set_Keep_Effective_IDs (Template : in out Process_Template);
   procedure Set_Signal_Mask
      (Template : in out Process_Template;
       Mask :     in     POSIX_Signals.Signal_Set);
   procedure Set_Creation_Signal_Masking
      (Template :       in out Process_Template;
       Masked_Signals : in     POSIX.Signal_Masking := POSIX.RTS_Signals);
   procedure Set_File_Action_To_Open
      (Template : in out Process_Template;
       File :     in     POSIX_IO.File_Descriptor;
       Name :     in     POSIX.Pathname;
       Mode :     in     POSIX_IO.File_Mode := POSIX_IO.Read_Only;
       Options :  in     POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set);
   procedure Set_File_Action_To_Close
      (Template : in out Process_Template;
       File :     in     POSIX_IO.File_Descriptor);
   procedure Set_File_Action_To_Duplicate
      (Template :  in out Process_Template;
       File :      in     POSIX_IO.File_Descriptor;
       From_File : in     POSIX_IO.File_Descriptor);
```

3.1.1.2 Description

An object of type `Process_Template` is used to represent the actions that are to take place when creating another process with `Start_Process` and `Start_Process_Search`. The user initializes a process template with `Open_Template`, modifies it using the various procedures with names of the form `Set_...`, and then passes it as a parameter to `Start_Process` or `Start_Process_Search`. When the user is done with the template, the user closes the template, and causes any storage allocated for it to be reclaimed, by calling `Close_Template`.

`Open_Template` shall open the process template specified by the parameter `Template`, if it is not already open, and initialize it to the following content:

— The effective user and group IDs shall be obtained from the real user and group IDs of the calling process, respectively.

— The Signal Mask attribute shall contain no signals.

— The set of file descriptor actions shall be empty; that is, no file descriptors will be changed when starting a process, except that file descriptors with the mode `Close_On_Exec` will be closed. (See type `POSIX_IO.File_Mode` in 6.1.1.)

`Close_Template` shall release any dynamically allocated storage that may be associated with the template and shall render the template unusable. Any subsequent calls using the template as a parameter will fail. This call is typically made by the application just before exiting the scope of the template. Failure by an application to call `Close_Template` to release templates may eventually result in storage exhaustion.

`Set_Keep_Effective_IDs` shall cause the effective user and group IDs of the new process to be the same as those of the calling process.

`Set_Signal_Mask` shall set the Signal Mask attribute of the specified template to the value of the parameter `Mask`.

`Set_Creation_Signal_Masking` shall set the `Masked_Signals` parameter that will be used for all the interruptible operations involved in creation and initialization of the new process to the value given by `Masked_Signals`. (See 3.3.6 and 2.4.1.6.) The interruptible operations involved in process creation include the open, close, and duplicate operations.

`Set_File_Action_To_Close` shall cause the file descriptor named by parameter `File` to be closed when the new process is started.

`Set_File_Action_To_Open` shall cause the file descriptor named by parameter `File` to be opened when the new process is started, using the values of the other parameters as parameters to `POSIX_IO.Open`. If the given file descriptor was already open, it shall be closed before the new file is opened.

`Set_File_Action_To_Duplicate` shall cause the file descriptor named by parameter `File` to be duplicated from the file descriptor named by `From_File` when the new process is started.

The effective order of events in processing the process template shall be as follows:

(1) The new process shall be created.

(2) The signal mask and the effective user and group IDs shall be set.

(3) The file actions specified by the `Set_File_Action_to_xxxx` operations shall be taken, in the order in which the procedures were called.

(4) The new process image shall replace the old one, and execution of that process image shall be started.

For example, the code sequence below has the effect of setting both standard output and standard error to reference a (newly opened) temporary file:

```
Set_File_Action_To_Open
        (Template=> T, File=> Standard_Output,
        Name=> "/tmp/name",...);
Set_File_Action_To_Duplicate
        (Template=> T, File=> Standard_Error,
        From_File=> Standard_Output);
```

For the Set_xxx operations on templates with a Filename or Pathname parameter, the string is simply stored in the template, without further processing. In particular, the current working directory used to resolve the Filename parameter shall be the one in force when the template is actually used to start a process, rather than the one in force at the time the template was updated. Also, no checking is performed for how well formed the filename or pathname is or for the existence of the specified file.

3.1.1.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Set_Keep_Effective_IDs, Set_Signal_Mask, Set_Creation_Signal_-Masking, Set_File_Action_To_Close, Set_File_Action_To_Open, Set_File_Action_To_Duplicate, or Close_Template was called with a parameter Template that represents a template that is closed, or for which the contents are or would become inconsistent as a result of the operation.

No exceptions shall be raised by Open_Template.

3.1.2 Process Creation

3.1.2.1 Synopsis

```
procedure Start_Process
   (Child :    out POSIX_Process_Identification.Process_ID;
    Pathname : in  POSIX.Pathname;
    Template : in  Process_Template;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
procedure Start_Process
   (Child :    out POSIX_Process_Identification.Process_ID;
    Pathname : in  POSIX.Pathname;
    Template : in  Process_Template;
    Env_List : in  POSIX_Process_Environment.Environment;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
procedure Start_Process_Search
   (Child :    out POSIX_Process_Identification.Process_ID;
    Filename : in  POSIX.Filename;
    Template : in  Process_Template;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
procedure Start_Process_Search
   (Child :    out POSIX_Process_Identification.Process_ID;
    Filename : in  POSIX.Filename;
    Template : in  Process_Template;
    Env_List : in  POSIX_Process_Environment.Environment;
    Arg_List : in  POSIX.POSIX_String_List := POSIX.Empty_String_List);
```

3.1.2.2 Description

`Start_Process` and `Start_Process_Search` shall create a new process, called the *child process*, executing the program or active partition contained in a regular file called the *new process image file*. The calling process is called the *parent process*.

NOTE: `Fork` and `Exec` operations are provided, in package `POSIX_Unsafe_Process_Primitives`, for functionality that cannot be obtained using `Start_Process` or `Start_Process_Search`.

`Start_Process` shall specify the new process image file by the value of the parameter `Pathname`. `Start_Process_Search` shall specify the new process image file by the value of the parameter `Filename`. If `Filename` does not contain a slash character, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable **PATH** . (See 2.11.) If this environment variable is not defined, the result of the search is implementation defined.

The initial state of the child process shall be the state of the calling process with modifications as specified below.

— The child process shall have a unique process ID.

— The parent process ID of the child process shall be the process ID of the parent (*i.e.*, the calling process).

— The child process shall have its own copy of the file descriptors of the parent. Each of the file descriptors of the child shall refer to the same open file description as the corresponding file descriptor of the parent, except as modified by the file actions set in the process template and the respective file modes. (See `Close_On_Exec` in 6.1.7.)

— The child process shall have no open directories.

— The per-process time-accounting information of the child (*i.e.*, user CPU time, system CPU time, user CPU time of the descendants, system time of the descendants) shall be set to zero. (See 4.2.1 for a discussion of process time accounting.)

— File locks previously set by the parent shall not be inherited by the child. (See 6.2.1.)

— The set of signals pending for the child process shall be initialized to the empty set. (See 3.3.7 for a discussion of signal sets.)

— The initial signal mask of the child process shall be the set specified by the Signal Mask attribute of the process template.

These specifications govern the initial signal mask of the new process, regardless of the language in which the new program was written. After creation of the new process, the signal mask may be modified as specified in 3.3.1.

NOTE: This operation may result in a temporary change in the signal mask of the calling process.

NOTE: It is anticipated that a future revision to this standard may specify that the mask specified by the Signal Mask attribute be overridden, in the case of realtime signals, if the process is an Ada active partition. See B.3.8 and the note in 3.3.1 for further explanation.

— The argument list of the child process shall be the list specified by the parameter `Arg_List`. By convention, the first component of `Arg_List` should be be the filename of the file specified by the parameter `Pathname` or `Filename`. (See 4.3.1 for a description of accessing arguments.)

— The environment shall be specified by the value of the parameter `Env_List` for the forms of the procedures where this parameter is present. Otherwise, the environment of the new process shall be the current environment of the calling process. (See 2.11 and 4.3.2.)

— The effective user and group IDs of the child process shall be determined by the information in the template and the permissions of the new process file, as explained further below.

— *If the Semaphores option is supported:* The child process shall not inherit any semaphores that are open in the parent process.

— *If the Memory Locking option is supported:* The child process shall not inherit any address space memory locks established by the parent process. Any locks established by the parent process shall not be affected.

— *If the Memory Mapped Files option is supported:* The child process shall not inherit any memory mappings created in the parent process.

— *If the Priority Process Scheduling option is supported:* If the `FIFO_Within_Priorities` or `Round_Robin_Within_Priorities` scheduling policy is in effect for the calling process, the child process shall inherit the policy and priority settings of the parent process. For other scheduling policies, the policy and priority settings of the child are implementation defined.

— *If the Timers option is supported:* The child process shall not inherit any timers created by the parent.

— *If the Message Queues option is supported:* The child process shall not inherit any message queue descriptors of the parent.

— *If the Asynchronous I/O option is supported:* No outstanding asynchronous input or asynchronous output operations shall be inherited by the child process.

— *If the Mutexes option is supported:* The state of mutexes and condition variables (if any) in the child process copy of the address space of the parent process is undefined.

Normally, if the template has not been set to keep effective IDs, the effective user and group IDs of the child process are set to the real user and group IDs of the creator. If the template has been set to keep effective IDs, they are set to the effective user and group IDs of the creator.

The effect of the template on the effective user and group IDs may be overridden if the new process image file has permission `Set_User_ID` or `Set_Group_ID`. (See 5.1.1.) If the new process image file has permission `Set_User_ID`, the effective user ID of the child process shall be set to the owner ID of the new process image file. Similarly, if the new process image file has permission `Set_Group_ID` (see 5.1.1), the effective group ID of the child process shall be set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the child process remain the same as those of the calling process.

If the Saved IDs option is supported: The effective user ID and effective group ID of the child process shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the `Set_User_ID` and `Set_Group_ID` operations. (See 4.1.3.)

All process attributes defined in this standard and not specified in this clause shall be inherited by the child from the parent process. The inheritance of process attributes not defined by this standard is implementation defined.

Upon successful completion, `Start_Process` and `Start_Process_Search` shall return the process ID of the child process via the parameter `Child`. They shall also mark for update the Last Access Time of the new process image file. (See 2.3.10.) The new process image file shall be considered to have been opened. The corresponding close shall be considered to occurs at a time after this open but before the new process terminates or successfully completes a subsequent call to `Start_Process`, `Start_Process_Search`, `Exec`, or `Exec_Search`. Both parent and child processes shall be capable of executing independently before either terminates.

If the operation failed, but was able to locate the new process image file, this standard does not specify whether the Last Access Time of the file is marked for update. (See 2.3.10.)

The `Start_Process` or `Start_Process_Search` operation may succeed, resulting in the creation of the child process; but the child process may be terminated before it begins execution of the program or active partition contained in the new process image file. In this case, the termination status of the terminated child process shall have the exit status value `Failed_Creation_Exit` (see 3.1.3). Specifically, the child process shall be terminated with exit status value `Failed_Creation_Exit` for the error conditions specified for `Exec` and `Exec_Search` (see 3.2.2) or for an open, close, or duplicate operation that fails for a file specified in the template.

3.1.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The template is closed.

`Resource_Temporarily_Unavailable`

The system lacked the necessary resources to create another process, or Child Processes Maximum would be exceeded. (Information on the latter limit is provided by `Child_Processes_Maximum` in the package `POSIX_Configurable_System_Limits`, described in 4.5.)

`Not_Enough_Space`

The system detected that the new process requires more storage space than the system is able to supply.

3.1.3 Process Exit

3.1.3.1 Synopsis

```
type Exit_Status is range 0 .. 2**8-1;
```

```
Normal_Exit :              constant Exit_Status := 0;
Failed_Creation_Exit :     constant Exit_Status := 41;
Unhandled_Exception_Exit : constant Exit_Status := 42;
procedure Exit_Process (Status : in Exit_Status := Normal_Exit);
```

3.1.3.2 Description

The type `Exit_Status` is used by the exiting process to communicate to its parent status information about its reason for exiting.

`Exit_Process` shall terminate the calling process, with the following consequences:

— In an Ada multitasking environment, all tasks created by the execution of the active partition shall cease execution permanently.

— All open file descriptors and open directories in the calling process shall be closed. The disposition of any output buffered by the Ada runtime system is implementation defined. (See 8.1.1.3 for a description of buffer flushing operations.)

— If the parent process of the calling process is executing a blocking `Wait_For_Child_Process` operation for the calling process, it shall be notified of the termination of the calling process and the exit status value shall be returned to the parent. (If the parent is an Ada active partition, the exit status shall be returned as part of the termination status value returned by the `Wait_For_Child_Process` operation.)

— If the parent process of the calling process is not executing a `Wait_For_Child_Process` operation for the calling process, the exit status value shall be saved for return to the parent process whenever the parent process executes an appropriate subsequent `Wait_For_Child_Process` operation.

— Termination of the process does not directly terminate its child processes. However, the sending of a `Signal_Hangup` signal to a child as a consequence of termination of its parent (see further down this list) may indirectly terminate the child process in some circumstances. Child processes of a terminated process shall be assigned a new parent process ID, corresponding to an implementation-definedsystem process.

— If the implementation supports the `Signal_Child` signal, a `Signal_Child` signal shall be sent to the parent process.

— If the process is a controlling process, the `Signal_Hangup` signal shall be sent to each process in the foreground process group of the controlling terminal belonging to the calling process.

— If the process is a controlling process, the controlling terminal associated with the session shall be disassociated from the session, allowing it to be acquired by a new controlling process.

— If the implementation supports the Job Control option, if the exit of the process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, then a `Signal_Hangup` signal followed by a `Signal_Continue` signal shall be sent to each process in the newly orphaned process group.

— *If the Semaphores option is supported:* All open named semaphores in the calling process shall be closed as if by appropriate calls to `POSIX_Semaphores.Close`.

— *If the Memory Locking option is supported:* Any memory locks established by the process via calls to `POSIX_Memory_Locking.Lock_All` or `POSIX_Memory_Range_Locking.Lock_Range` shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call to `Exit_Process` of the calling process.

— *If the Memory Mapped Files option is supported:* Memory mappings created in the process shall be unmapped before the process is destroyed.

— *If the Message Queues option is supported:* All open message queue descriptors in the calling process shall be closed, as if by appropriate calls to `POSIX_Message_Queues.Close`.

— *If the Asynchronous I/O option is supported:* Any outstanding cancelable asynchronous I/O operations may be canceled. AIO operations that are not canceled shall complete as if the `Exit_Process` operation had not yet occurred, but any associated signal notifications shall be suppressed. The `Exit_Process` operation itself may or may not block awaiting such I/O completion. Whether any I/O is canceled and which I/O may be canceled upon `Exit_Process` are implementation defined.

When an Ada active partition exits without explicitly calling `Exit_Process`, the Ada language implementation shall provide the exit status value.

The exit status value of an Ada active partition shall be `Unhandled_Exception_Exit` if completion occurs because of an unhandled exception.

For a child process that is terminated before the beginning of execution of the program or active partition in the new process image file, as described in 3.1.2, the exit status value shall be `Failed_Creation_Exit`. By convention, a POSIX/Ada application should not use the two predefined exit status values for other meanings. However, the POSIX/Ada interface shall permit these values to be used with `Exit_Process`.

3.1.3.3 Error Handling

No exceptions shall be raised by `Exit_Process`.

3.1.4 Termination Status

3.1.4.1 Synopsis

```
type Termination_Status is private;
type Termination_Cause is
   (Exited, Terminated_By_Signal, Stopped_By_Signal);
function Status_Available (Status : Termination_Status)
   return Boolean;
function Process_ID_Of (Status : Termination_Status)
   return POSIX_Process_Identification.Process_ID;
function Termination_Cause_Of (Status : Termination_Status)
   return Termination_Cause;
function Exit_Status_Of (Status : Termination_Status)
   return Exit_Status;
function Termination_Signal_Of (Status : Termination_Status)
   return POSIX_Signals.Signal;
function Stopping_Signal_Of (Status : Termination_Status)
   return POSIX_Signals.Signal;
```

3.1.4.2 Description

The type `Termination_Status` is used to provide information about the condition that caused termination of a process. If the implementation supports the Job Control option, the type `Termination_Status` is also used to provide information about the condition that caused a process to be stopped. Information may be extracted from a value of type `Termination_Status` by means of query functions.

For each value of type `Termination_Status`, it shall be possible to determine whether status is available from that value. All objects of type `Termination_Status` shall have a default initial value from which status is not available.

Each value of the type `Termination_Status` from which status is available shall have an associated value of `Process_ID`, which specifies the process for which status is available, and an associated value of `Termination_Cause`, which specifies one of the reasons an Ada process may terminate or stop. The possible termination causes are

`Exited`
: Some task in the process called `Exit_Process` (see 3.1.3), or the process was implicitly terminated by the action of the Ada language implementation in response to normal active partition completion, propagation of an unhandled exception, or receipt of a signal caught by the Ada language implementation. In all these cases the process is said to have "exited."

  NOTE: The termination of the environment task is required to await the termination of all tasks created as part of the execution of the partition, except when a task calls `Exit_Process` (see 10.2 (25) of the Ada RM {1}).

`Terminated_By_Signal`
: The process was terminated by the action of the operating system due to the receipt of a signal that was not caught by the application or the runtime system of the Ada language implementation.

`Stopped_By_Signal`
: The process was stopped by a signal.

`Status_Available` shall return `True` if and only if status is available from the `Status` parameter.

`Process_ID_Of` shall return the process ID of the `Status` parameter if status is available from it. Otherwise, this operation shall raise `POSIX_Error`.

`Termination_Cause_Of` shall return the termination cause of the `Status` parameter if status is available from it. Otherwise, this operation shall raise `POSIX_Error`.

Whether one of the other query functions returns a value for a given termination status value is determined by the associated termination cause, as follows:

`Exited`
: `Exit_Status_Of` shall return the exit status of the process, as defined in 3.1.3.

`Terminated_By_Signal`
: `Termination_Signal_Of` shall return the signal that caused the process to terminate.

`Stopped_By_Signal`

`Stopping_Signal_Of` shall return the signal that caused the process to stop.

An attempt to call these query functions for a termination status value for which status is not available, or with a termination cause other than the one specified for that function, shall cause `POSIX_Error` to be raised.

3.1.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

(1) `Process_ID_Of`, `Exit_Status_Of`, `Termination_Signal_Of`, or `Termination_Cause_Of` was called with an argument `Status` from which status is not available.

(2) `Exit_Status_Of` was called with a parameter for which `Termination_Cause_Of` would not return `Exited`.

(3) `Termination_Signal_Of` was called with an argument for which `Termination_Cause_Of` would not return `Terminated_By_Signal`.

(4) `Stopping_Signal_Of` was called with an argument for which `Termination_Cause_Of` would not return `Stopped_By_Signal`.

No exceptions shall be raised by `Status_Available`.

3.1.5 Wait for Process Termination

3.1.5.1 Synopsis

```
procedure Wait_For_Child_Process
   (Status : out Termination_Status;
    Block :  in  Boolean := True;
    Trace_Stopped :  in Boolean := True;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Wait_For_Child_Process
   (Status : out Termination_Status;
    Child :  in  POSIX_Process_Identification.Process_ID;
    Block :  in  Boolean := True;
    Trace_Stopped :  in Boolean := True;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Wait_For_Child_Process
   (Status : out Termination_Status;
    Group :  in  POSIX_Process_Identification.Process_Group_ID;
    Block :  in  Boolean := True;
    Trace_Stopped :  in Boolean := True;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

3.1.5.2 Description

`Wait_For_Child_Process` allows the calling process to enquire about the status of one of its child processes. Status shall be available for a process if the process has terminated or is stopped and its status has not been reported by `Wait_For_Child_Process` since it terminated or was last stopped.

A process that is stopped may be signaled to continue. If a child process is stopped and then is signaled to continue before its parent attempts to obtain status for it, the parent will not find status available until the process is stopped again or terminates. If a process is signaled to continue after status has been obtained, status may be obtained for that process again if it is stopped again or terminates. If status information is available for two or more processes, the order in which their status is reported is unspecified.

An implementation may define additional circumstances under which status is available for a child process. Status shall not be made available under such additional circumstances unless the calling process or one of its child processes explicitly makes use of a nonstandard extension. In these cases, the interpretation of the reported status is implementation defined.

The status of a child process has not yet been reported if either

(1) It has not terminated and is not stopped.

(2) It has terminated or is stopped and has status available.

`Wait_For_Child_Process` shall permit the calling process to obtain the status of one out of a set of child processes. This set is specified by the parameters. The parameter `Traced_Stopped` specifies whether stopped child processes are to be included if the implementation supports the Job Control option. If the implementation supports the Job Control option, stopped child processes shall be included in the set only if `Traced_Stopped` is `True`. The parameter `Masked_Signals` specifies which additional signals shall be blocked from delivery during the operation. (See 2.4.1.6.) If more than one task waits for a particular child, only one task shall return with the status of the child; what occurs for the other tasks is unspecified.

For the form of `Wait_For_Child_Process` without a parameter `Child` or `Group`, the set of processes for which status is requested contains all the child processes of the calling process. For the form of `Wait_For_Child_Process` with parameter `Child`, this set contains only the child process with the specified process ID. For the form of `Wait_For_Child_Process` with parameter `Group`, the set contains all the child processes that are members of the specified process group.

When this set contains a child process whose status has not yet been reported and for which status is available, the status of the child shall be returned in the parameter `Status`. The process ID of the child process shall be available by applying the function `Process_ID_Of` to the value returned in the parameter `Status`.

If the set of processes for which status is requested contains no child processes, `POSIX_Error` shall be raised with error code `No_Child_Process`.

If the set of processes for which status is requested contains child processes whose statuses have not yet been reported, but status is not available for any of these, the

operation shall either block until status becomes available or return without reporting status, as determined by the value of the parameter Block. If Block is False, the procedure shall return a value for the parameter Status so that Status_Available shall return False.

NOTE: The other query functions raise POSIX_Error for such a value of Status.

If Block is True, the procedure shall block the calling process until status becomes available for a child process in the set specified by the other parameters.

NOTE: In the Ada multitasking environment, waiting for status of a child process may affect the entire process, not just the calling task. Whether the entire process or just the calling task is blocked is specified by the constant POSIX.Wait_For_Child_Blocking_Behavior. (See 2.4.1.5.)

3.1.5.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

No_Child_Process
: Wait_For_Child_Process was called with parameters that specify an empty set of child processes.

Interrupted_Operation
: The operation was interrupted by a signal.

3.2 Package POSIX_Unsafe_Process_Primitives

```
with POSIX,
     POSIX_Process_Environment,
     POSIX_Process_Identification;
package POSIX_Unsafe_Process_Primitives is
    -- 3.2.1 Process Creation
   function Fork return POSIX_Process_Identification.Process_ID;
    -- 3.2.2 File Execution
   procedure Exec
      (Pathname : in POSIX.Pathname;
       Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List;
       Env_List : in POSIX_Process_Environment.Environment);
   procedure Exec
      (Pathname : in POSIX.Pathname;
       Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List);
   procedure Exec_Search
      (Filename : in POSIX.Filename;
       Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List;
       Env_List : in POSIX_Process_Environment.Environment);
   procedure Exec_Search
      (Filename : in POSIX.Filename;
       Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List);

end POSIX_Unsafe_Process_Primitives;
```

This package gives the user the capability to create a new process with Fork and to cause the current process to execute a new process image with Exec or Exec_Search.

These operations are most frequently used as part of a sequence, in which the parent process uses Fork to create a child, the child process alters its state, and then the child uses Exec or Exec_Search to execute a new program or active partition.

A Strictly Conforming POSIX.5 Application shall call Fork only if it has no unterminated tasks (other than the environment task corresponding to the active partition) and every file open for output has been flushed since the last output operation on that file. For other uses of Fork, the relationship of the state of the child process to the state of the calling process and the disposition of buffered output are unspecified.

A Strictly Conforming POSIX.5 Application shall call Exec and Exec_Search only if every file open for output has been flushed since the last output operation to that file. For other uses of these procedures, the disposition of any buffered output is unspecified.

The new process may be created with only a single task. If so, the new process contains a replica of the calling task and its entire address space, possibly including the states of mutexes and other resources. Consequently, the effect on the child process of executing any operations on mutexes or condition variables, any potentially blocking operations, or any Ada tasking operations until such time as one of the Exec family of operations is called is undefined.

For maximum portability, the application should avoid use of this package; if possible, it should use the operations of the package POSIX_Process_Primitives, instead.

3.2.1 Process Creation

3.2.1.1 Synopsis

```
function Fork return POSIX_Process_Identification.Process_ID;
```

3.2.1.2 Description

Fork shall create a new process, which is called the child process. The child process shall be an exact copy of the calling process, which is called the parent process, except for the following:

— The child process shall have a unique process ID.

— The parent process ID of the child shall be the process ID of the calling process.

— The child process shall have its own copy of the file descriptors of the parent. Each of file descriptors of the child shall refer to the same open file description as the parent.

— The per-process time-accounting information of the child (*i.e.*, user CPU time, system CPU time, user CPU time of the descendants, system time of the descendants) shall be set to zero. (See 4.2.1 for a description of process time accounting.)

— File locks previously set by the parent shall not be inherited by the child. (See 6.2.1 for a description of file locking.)

— The set of signals pending for the child process shall be initialized to the empty set. (See 3.3.7 for a description of signal sets.)

— If the parent process has more than one unterminated task, the set of unterminated tasks in the child process is unspecified.

— *If the Semaphores option is supported:* Any semaphores that are open in the parent process shall also be open in the child process.

— *If the Memory Locking option is supported:* The child process shall not inherit any address space memory locks established by the parent process via calls to `POSIX_Memory_Locking.Lock_All` or `POSIX_Memory_Range_Locking.Lock_Range`.

— *If the Memory Mapped Files option is supported:* Memory mappings created in the parent shall be retained in the child process. Private mappings (see `POSIX_Memory_Mapping.Map_Private`) inherited from the parent shall also be private mappings in the child. Any modifications to the data in these mappings made by the parent prior to calling `Fork` shall be visible to the child. Any modifications to the data in private mappings made by the parent after `Fork` returns shall be visible only to the parent. Modifications to the data in private mappings made by the child shall be visible only to the child.

— *If the Priority Process Scheduling option is supported:* If the `FIFO_Within_Priorities` or `Round_Robin_Within_Priorities` scheduling policy is in effect for the calling process, the child process shall inherit the policy and priority settings of the parent process during a `Fork` operation.. For other scheduling policies, the policy and priority settings on `Fork` are implementation defined.

— *If the Timers option is supported:* The child process shall not inherit any timers created by the parent.

— *If the Message Queues option is supported:* The child process shall have its own copy of the message queue descriptors of the parent process. Each of the message queue descriptors of the child process refers to the same open message queue description as the corresponding message queue descriptor of the parent.

— *If the Asynchronous I/O option is supported:* No AIO operations shall be inherited by the child process.

All other process attributes defined by this standard shall be the same in the parent and the child processes. The inheritance of attributes not defined by this standard is implementation defined.

Upon successful completion, `Fork` shall return the value `Null_Process_ID` to the child process and shall return the `Process_ID` of the child process to the parent. Both processes shall continue to execute from the call to `Fork`. Both parent and child processes shall be capable of executing independently before either terminates.

If `Fork` raises an exception, no process shall be created.

3.2.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Resource_Temporarily_Unavailable`

The system lacked the necessary resources to create another process, or Child Processes Maximum would be exceeded. (Information on the latter limit is provided by `Child_Processes_Maximum` in the package `POSIX_Configurable_System_Limits`, described in 4.5.)

`Not_Enough_Space`

The system detected that the process requires more storage space than the system is able to supply.

3.2.2 File Execution

3.2.2.1 Synopsis

```
procedure Exec
   (Pathname : in POSIX.Pathname;
    Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List;
    Env_List : in POSIX_Process_Environment.Environment);
procedure Exec
   (Pathname : in POSIX.Pathname;
    Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List);
procedure Exec_Search
   (Filename : in POSIX.Filename;
    Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List;
    Env_List : in POSIX_Process_Environment.Environment);
procedure Exec_Search
   (Filename : in POSIX.Filename;
    Arg_List : in POSIX.POSIX_String_List := POSIX.Empty_String_List);
```

3.2.2.2 Description

`Exec` and `Exec_Search` shall cause the calling process to execute a new process image. The new process image is constructed from a regular, executable file called the new process image file. There shall be no return from a successful call to `Exec` or `Exec_Search` because the old process image of the calling process is overlaid by the new process image.

The replacement of the calling process by the new process image is immediate and does not involve any Ada-specific finalization. In particular, if there are tasks besides the calling task, they immediately shall cease to exist along with the old process image. Similarly, if there is buffered output, it shall be discarded along with the old process image.

NOTE: The effects of `Exec` and `Exec_Search` on tasks and buffered output, described above, are similar to the effects if the process is terminated by the action of a signal or that calls `Exit_Process`.

`Exec` shall specify the new process image by the value of the parameter `Pathname`. `Exec_Search` shall specify the new process image file by the value of the parameter `Filename`. If the parameter `Filename` does not contain a slash character, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable **PATH**. (See 2.11 and 4.3.2.) If this environment variable is not defined, the results of the search are implementation defined.

The parameter `Arg_List` shall become the new argument list of the calling process. By convention, the first component of `Arg_List` should be the filename of the file specified by the parameter `Pathname` or `Filename`. (See 4.3.1.)

If the parameter `Environment` is present, it shall become the new environment of the calling process. Otherwise, the current environment of the calling process shall be maintained. (See 2.11 and 4.3.2.)

The number of POSIX characters available for the combined argument and environment lists may be limited by the implementation to Argument List Maximum. (Information on this limit is provided by `Argument_List_Maximum` in the package `POSIX_Configurable_System_Limits`, described in 4.5.)

File descriptors open in the calling process image shall remain open in the new process image, except that those with mode `Close_on_Exec` (see 6.1.7) shall be closed. For file descriptors that remain open, all attributes of the open file description, including the file locks, shall remain unchanged by this operation. (See 6.2.1.)

Any active directory iterators shall be terminated, and any associated implementation file structures shall be closed.

The operation shall leave the signal actions in the calling process unchanged, except that any bindings of signals to task entries shall be broken and the actions for these signals shall revert to the defaults.

NOTE: The statement above implies that signals that are set to be ignored by the calling process shall also be ignored in the new process image.

The set of signals blocked from delivery for the environment task of the process shall be that of the calling task.

If the new process image file has permission `Set_User_ID`, the effective user ID of the process shall be set to the owner ID of the new process image file. (See 4.1.3.) Similarly, if the new process image file has permission `Set_Group_ID`, the effective group ID of the process is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the process remain the same as those of the calling process. If the Saved IDs option is supported, the effective user ID and effective group ID of the process shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the `Set_User_ID` and `Set_Group_ID` operations. (See 4.1.4.)

The process also shall retain the following attributes from the calling process:

— Process ID

— Parent process ID

— Process group ID

— Session membership

— Real user ID

— Real group ID

— Supplementary group IDs

— Time left until an alarm clock signal paragraphs)

NOTE: The time left until an alarm clock signal is not something that would ordinarily be observable to an Ada application, since this signal is a reserved signal. (See 2.2.2.155.) However, this may be observable by the new process image if it was implemented in another programming language.

— Current working directory

— Root directory

— File mode creation mask (see 5.1.1)

— Pending signals (see 3.3)

— Process times (see 4.2.1)

If the Semaphores option is supported: Any named semaphores that are open in the calling process shall be closed as if by appropriate calls to `POSIX_Semaphores.Close`.

If the Memory Locking option is supported: Memory locks established by the calling process via calls to `POSIX_Memory_Locking.Lock_All` or `POSIX_Memory_Range_Locking.Lock_Range` shall be removed. If locked pages in address space of the calling process are also mapped into the address space of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call to `Exec` or `Exec_Search`. If `Exec` or `Exec_Search` fails, the effect on memory locks is unspecified.

If the Memory Mapped Files option is supported: Memory mappings created in the process shall be unmapped before the address space is rebuilt for the new process image.

If the Priority Process Scheduling option is supported: If the `FIFO_Within_Priorities` or `Round_Robin_Within_Priorities` scheduling policy is in effect for the calling process, the policy and priority settings shall not be changed by a call to `Exec` or `Exec_Search`. For other scheduling policies, the policy and priority settings on `Exec` or `Exec_Search` are implementation defined.

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

If the Message Queues option is supported: All open message queue descriptors in the calling process shall be closed, as described in `POSIX_Message_Queues.Close`.

If the Asynchronous I/O option is supported: Any outstanding asynchronous I/O operations may be canceled. Those AIO operations that are not canceled shall complete as if the `Exec` or `Exec_Search` had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether `Exec` or `Exec_Search` itself blocks, awaiting such I/O completion. In no event, however, shall the new process image created by `Exec` or `Exec_Search` be affected by the presence of outstanding asynchronous I/O operations at the time `Exec` or `Exec_Search` is called. Whether any I/O is canceled, and which I/O may be canceled upon `Exec` or `Exec_Search`, is implementation defined.

All process attributes defined by this standard and not specified in this clause shall remain unchanged by the operations `Exec` and `Exec_Search`. The inheritance of process attributes not defined by this standard is implementation defined.

On successful completion, the process shall begin executing the program or active partition from the new process image file. The Last Access Time of the file shall be marked for update. (See 2.3.10.) The new process image file shall be considered to have been opened. The corresponding close shall be considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to `Start_Process`, `Start_Process_Search`, `Exec`, or `Exec_Search` by this process. If the function failed but was able to locate the new process image file, this standard does not specify whether the Last Access Time of the file is marked for update.

Otherwise, if the call to either `Exec` or `Exec_Search` is not successful, the calling process shall continue execution, and `POSIX_Error` shall be raised by the call.

3.2.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Argument_List_Too_Long`

The number of POSIX characters used by the argument list and the environment list of the new process image is greater than the system-imposed limit, Argument List Maximum. (Information on this limit is provided by `Argument_List_Maximum` in the package `POSIX_Configurable_System_Limits`, described in 4.5.

NOTE: This limit may include storage needed for system overhead in maintaining the list.)

`Permission_Denied`

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

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`No_Such_File_Or_Directory`

One or more components of the pathname of the new process image file do not exist, or the parameter `Pathname` or `Filename` is a null string.

`Not_A_Directory`

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

`Exec_Format_Error`

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

`Not_Enough_Space`

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

3.3 Package POSIX_Signals

This package defines the type Signal and operations for sending, accepting, and catching signals. Signals are events that may affect the execution of a process.

```
with POSIX,
   POSIX_Process_Identification,
   Ada_Task_Identification,
   System;
package POSIX_Signals is
   -- 3.3.2 Signal Type
   type Signal is implementation-defined-integer;;
   function Image (Sig : Signal) return String;
   function Value (Str : String) return Signal;
   -- 3.3.3 Standard Signals
   Signal_Null,
   SIGNULL : constant Signal := 0;
   Signal_Abort,
   SIGABRT : constant Signal := implementation-defined;
   Signal_Alarm,
   SIGALRM : constant Signal := implementation-defined;
   Signal_Bus_Error,
   SIGBUS :  constant Signal := implementation-defined;
   Signal_Floating_Point_Error,
   SIGFPE :  constant Signal := implementation-defined;
   Signal_Hangup,
   SIGHUP :  constant Signal := implementation-defined;
   Signal_Illegal_Instruction,
   SIGILL :  constant Signal := implementation-defined;
   Signal_Interrupt,
   SIGINT :  constant Signal := implementation-defined;
   Signal_Kill,
   SIGKILL : constant Signal := implementation-defined;
   Signal_Pipe_Write,
   SIGPIPE : constant Signal := implementation-defined;
   Signal_Quit,
   SIGQUIT : constant Signal := implementation-defined;
   Signal_Segmentation_Violation,
   SIGSEGV : constant Signal := implementation-defined;
   Signal_Terminate,
   SIGTERM : constant Signal := implementation-defined;
   Signal_User_1,
   SIGUSR1 : constant Signal := implementation-defined;
   Signal_User_2,
   SIGUSR2 : constant Signal := implementation-defined;
   Signal_Child,
   SIGCHLD : constant Signal := implementation-defined;
   Signal_Continue,
   SIGCONT : constant Signal := implementation-defined;
   Signal_Stop,
   SIGSTOP : constant Signal := implementation-defined;
   Signal_Terminal_Stop,
   SIGTSTP : constant Signal := implementation-defined;
   Signal_Terminal_Input,
   SIGTTIN : constant Signal := implementation-defined;
   Signal_Terminal_Output,
   SIGTTOU : constant Signal := implementation-defined;
```

```
Signal_IO,
SIGIO :   constant Signal := implementation-defined;
Signal_Out_Of_Band_Data,
SIGURG :  constant Signal := implementation-defined;                    c
subtype Realtime_Signal is Signal range implementation-defined;
-- 3.3.7 Signal Sets
type Signal_Set is private;
procedure Add_Signal
   (Set : in out Signal_Set;
    Sig : in Signal);
procedure Add_All_Signals (Set : in out Signal_Set);
procedure Delete_Signal
   (Set : in out Signal_Set;
    Sig : in Signal);
procedure Delete_All_Signals (Set : in out Signal_Set);
function Is_Member
   (Set : Signal_Set;
    Sig : Signal)
  return Boolean;
-- 3.3.8 Block and Unblock Signals
procedure Set_Blocked_Signals
   (New_Mask : in Signal_Set;
    Old_Mask : out Signal_Set);
procedure Block_Signals
   (Mask_to_Add : in Signal_Set;
    Old_Mask :    out Signal_Set);
procedure Unblock_Signals
   (Mask_to_Subtract : in Signal_Set;
    Old_Mask :         out Signal_Set);
function Blocked_Signals return Signal_Set;
-- 3.3.9 Ignore Signals
procedure Ignore_Signal (Sig : in Signal);
procedure Unignore_Signal (Sig : in Signal);
procedure Restore_Default_Action (Sig : in Signal)
  renames Unignore_Signal;
function Is_Ignored (Sig : Signal) return Boolean;
procedure Install_Empty_Handler (Sig : in Signal);                      c
-- 3.3.10 Controlling Generation of Signal for Child Process
procedure Set_Stopped_Child_Signal (Enable : in Boolean := True);
function Stopped_Child_Signal_Enabled return Boolean;
-- 3.3.11 Examine Pending Signals
function Pending_Signals return Signal_Set;
-- 3.3.12 Signal Event Notification
type Signal_Event is private;
type Signal_Data  is private;
type Notification is range implementation-defined;
No_Notification :     constant Notification := implementation-defined;
Signal_Notification : constant Notification := implementation-defined;
function Get_Signal (Event : Signal_Event) return Signal;
procedure Set_Signal
   (Event : in out Signal_Event;
    Sig :   in Signal);
function Get_Notification (Event : Signal_Event) return Notification;
procedure Set_Notification
   (Event :  in out Signal_Event;
    Notify : in Notification);
function Get_Data (Event : Signal_Event) return Signal_Data;
```

```
procedure Set_Data
   (Event : in out Signal_Event;
    Data :  in Signal_Data);
-- 3.3.13 Signal Information
type Signal_Source is range implementation-defined;
From_Send_Signal :   constant Signal_Source := implementation-defined;
From_Queue_Signal :  constant Signal_Source := implementation-defined;
From_Timer :         constant Signal_Source := implementation-defined;
From_Async_IO :      constant Signal_Source := implementation-defined;
From_Message_Queue : constant Signal_Source := implementation-defined;
type Signal_Info is private;
function Get_Signal (Info : Signal_Info) return Signal;
procedure Set_Signal
   (Info : in out Signal_Info;
    Sig :  in Signal);
function Get_Source (Info : Signal_Info) return Signal_Source;
procedure Set_Source
   (Info :   in out Signal_Info;
    Source : in Signal_Source);
function Has_Data (Source : Signal_Source) return Boolean;
function Get_Data (Info : Signal_Info) return Signal_Data;
procedure Set_Data
   (Info : in out Signal_Info;
    Data : in Signal_Data);
-- 3.3.14 Control Signal Queueing
procedure Enable_Queueing (Sig : in Signal);
procedure Disable_Queueing (Sig : in Signal);
-- 3.3.15 Wait for Signal
function Await_Signal (Set : Signal_Set) return Signal;
function Await_Signal_Or_Timeout
   (Set :     Signal_Set;
    Timeout : POSIX.Timespec)
  return Signal;
-- 3.3.16 Wait for Signal with Information
function Await_Signal (Set : Signal_Set) return Signal_Info;
function Await_Signal_Or_Timeout
   (Set :     Signal_Set;
    Timeout : POSIX.Timespec)
  return Signal_Info;
-- 3.3.17 Signal Entries      -- obsolescent
Signal_Abort_Ref :           constant System.Address := implementation-defined;
--  Signal_Floating_Point_Error intentionally omitted.
Signal_Hangup_Ref :          constant System.Address := implementation-defined;
--  Signal_Illegal_Instruction intentionally omitted.
Signal_Interrupt_Ref :       constant System.Address := implementation-defined;
--  Signal_Kill intentionally omitted.
Signal_Pipe_Write_Ref :      constant System.Address := implementation-defined;
Signal_Quit_Ref :            constant System.Address := implementation-defined;
Signal_Terminate_Ref :       constant System.Address := implementation-defined;
Signal_User_1_Ref :          constant System.Address := implementation-defined;
Signal_User_2_Ref :          constant System.Address := implementation-defined;
Signal_Child_Ref :           constant System.Address := implementation-defined;
Signal_Continue_Ref :        constant System.Address := implementation-defined;
--  Signal_Stop intentionally omitted.
Signal_Terminal_Stop_Ref :   constant System.Address := implementation-defined;
Signal_Terminal_Input_Ref :  constant System.Address := implementation-defined;
Signal_Terminal_Output_Ref : constant System.Address := implementation-defined;
function Signal_Reference (Sig : Signal) return System.Address;
```

```
   -- 3.3.18 Send a Signal
   procedure Send_Signal
      (Process : in POSIX_Process_Identification.Process_ID;
       Sig :     in Signal);
   procedure Send_Signal
      (Group : in POSIX_Process_Identification.Process_Group_ID;
       Sig :   in Signal);
   procedure Send_Signal (Sig : in Signal);
   -- 3.3.19 Queue a Signal
   procedure Queue_Signal
      (Process : in POSIX_Process_Identification.Process_ID;
       Sig :  in Signal;
       Data : in Signal_Data);
   -- 3.3.20 Interrupt a Task
   procedure Interrupt_Task (T : in Ada_Task_Identification.Task_ID);
private
   implementation-defined
end POSIX_Signals;
```

3.3.1 Signal Model

A signal is an abstraction of an event. An occurrence of a signal is said to be *generated* when the event that causes the signal occurs. Examples of such events include detection of hardware faults, timer expiration, and terminal activity as well as the invocation of the `Send_Signal` operation. A signal may be generated for a process. A signal may also be generated for a particular task within a process. In some cases, the same event generates signals for multiple processes.

If signals can be generated for tasks, at the time a signal is generated, a determination shall be made whether the signal has been generated for the process or for a specific task within the process. Signals that are generated by some action attributable to a particular task, such as a hardware fault, shall be generated for the task that caused the signal to be generated. Signals that are generated in association with a process ID, a process group ID, or an asynchronous event such as terminal activity shall be generated for the process.

A signal may be *accepted* synchronously by a task within a process. Alternatively, it may be *delivered* asynchronously to a task within a process. The latter will only be visible to the task indirectly, through a service performed by the Ada runtime system such as the `Interrupt_Task` operation described in 3.3.20, and through the exception `POSIX_Error` being raised with error code `Interrupted_Operation` if the task is executing a call to the interruptible operations described in 3.3.6 of this standard. These two different kinds of signal notification are explained below.

A task is said to *accept a signal* when the signal is selected and returned by one of the *signal-awaiting operation*s defined by this standard, which cause a task to block and wait for the arrival of a signal. These operations are as follows:

— `POSIX_Signals.Await_Signal`

— `POSIX_Signals.Await_Signal_Or_Timeout`

See 3.3.15 and 3.3.16 for more details.

If the Signal Entries option is supported: A task is also said to *accept* a signal when it executes a rendezvous via an **accept** statement for a task entry that has been bound to the signal.

NOTE: Throughout this section, the term "accept" is used to mean both of the signal notification mechanisms defined above, and is used in contrast to the term "deliver" defined below.

The system is said to *deliver a signal* occurrence when it performs an action in response to the signal occurrence, other than holding the signal occurrence pending for a task to accept it. For each process and signal, there shall be a corresponding *signal action* that shall be taken by the system when it delivers the signal.

The possible signal actions include the following:

Ignore the signal.

Only certain signals can be *ignore*d. An attempt to call `Ignore_Signal` or `Unignore_Signal` for a signal that is not permitted to be ignored shall cause `POSIX_Error` to be raised with error code `Invalid_Argument`.

If the signal action for a signal is to ignore it, delivery of the signal shall have no effect on the process.

Setting the signal action to ignore for a signal that is pending shall cause any pending occurrences of the signal to be discarded, whether or not the signal is blocked. Any queued values pending shall be discarded and the resources used to queue them shall be released and made available to queue other signals. If a process sets the action for `Signal_Child` to ignore, the behavior is unspecified.

Catch the signal.

Other languages allow that certain signals may be caught within the process to which they are delivered, via a signal handler subprogram installed by the application (see POSIX.1). This mechanism is not directly available to Ada applications under this standard. However, handler subprograms may be installed by the Ada language implementation, for the reserved signals (see 2.2.2.155), and possibly for signals attached to task entries. In mixed-language programs, they may also be installed via interfaces not defined by this standard. The effect on an Ada active partition of calling an Ada tasking operation or transferring control from a handler installed by such means (*via, e.g.,* C-language *longjmp(),* propagation of an Ada exception, **goto**) is undefined.

Take the default action.

Every signal has a default signal action. (See 3.3.4.) The default action is taken if all of the following conditions are satisfied:

— No task is currently waiting for the signal.

— No entry is bound to the signal.

— The signal action has not been set to ignore the signal.

The effect of changing the signal action for a signal that is currently awaited by a task is unspecified.

NOTE: It is specified in POSIX.1 that when a task calls one of the signal-awaiting operations to wait for a set of signals the effect on the actions is unspecified. Thus, it is unsafe to allow a signal to be unblocked for delivery to any task after any task has performed a signal-awaiting operation on that signal in the process, until some operation has been executed to restore the signal action to a well-defined state. (See 3.3.15 of this standard.)

During the time between the generation of a signal and its acceptance or delivery, the signal is said to be *pending*. If the occurrence was generated for the process it is pending for the process; if it was generated for a specific task it is pending for that task. Ordinarily, this interval cannot be detected by an application. However, a signal can be *blocked* from delivery to a task. If a signal occurrence is generated for a task and the signal action associated with the signal is anything other than to ignore the signal, the signal occurrence shall remain pending until it is unblocked for delivery to that task, it is accepted by a call to a signal-awaiting operation by that task, or the action is set to ignore the signal. Signals generated for the process shall be accepted by or delivered to exactly one of the tasks within the process that is ready to accept the signal or for which the signal is not blocked from delivery. If no tasks are ready to accept the signal and the signal is blocked from delivery to all tasks within the process, the signal occurrence shall remain pending on the process until a task is ready to accept the signal, the signal is no longer blocked from delivery to some task, or the action associated with the signal is set to ignore the signal. If the action associated with a blocked signal is to ignore the signal and if that signal is generated for the process or task, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Generation of a signal is guaranteed to cause a subsequent attempt at delivery, but there is no guarantee that an attempt at delivery will be made for every occurrence of the signal. In the case where the signal is generated again while a previous occurrence is still pending, the system need not queue multiple occurrences of a signal.

If the Realtime Signals option is supported: Queueing may be enabled for a signal (see 3.3.14). When queueing is enabled for a signal, operations that generate occurrences of the signal are guaranteed to queue the signal if they return normally and to fail if there are insufficient resources to queue the signal.

The set of signals that is blocked from delivery to a task is called the *signal mask* of the task. It is unspecified whether the signal mask of a task is an attribute of the individual task or an attribute of the process to which the task belongs.

It is implementation defined whether the signal mask is per task or per process.

For the environment task, the initial signal mask is that specified for the process in process creation (see 3.1.2 and 3.2.1).

NOTE: It is anticipated that the paragraph above may be modified by a future revision of this standard to require that the realtime signals always be initially masked for a process that is an Ada active partition. See B.3.8 for further details.

For all other tasks, the initial signal mask shall include all the signals that are not reserved signals (see 2.2.2.155) and are not bound to entries of the task.

An application can explicitly modify the signal mask via calls to `Block_Signals`, `Unblock_Signals`, and `Set_Blocked_Signals`.

NOTE: Applications may mask signals when in critical regions. A critical region is a region of code where uninterrupted control is required to maintain consistency of the perceived and actual state of resources.

The implementation of the Ada language or this standard may temporarily block signals during the execution of operations defined by this standard and during calls to the Ada runtime system, as necessary.

The implementation is also allowed to block or unblock any of the reserved signals (see 2.2.2.155) and any of the signals bound by the application to task entries at any time that is consistent with the other specifications in this standard.

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

When any stop signal (`Signal_Stop`, `Signal_Terminal_Stop`, `Signal_Terminal_Input`, `Signal_Terminal_Output`) is generated for a process, any pending `Signal_Continue` signals for that process shall be discarded. Conversely, when `Signal_Continue` is generated for a process, all pending stop signals for that process shall be discarded. When `Signal_Continue` is generated for a process that is stopped, the process shall be continued, even if the `Signal_Continue` is blocked or ignored. If `Signal_Continue` is blocked and not ignored, it shall remain pending until it is either unblocked or a stop signal is generated for the process.

NOTE: If `Signal_Continue` is blocked and not ignored, continuing the process means the signal will cause the process to resume executing but will not cause it to execute any asynchronous signal handler.

Some signal-generating operations, such as high-resolution timer expiration, AIO completion, interprocess message arrival, and the `Queue_Signal` procedure, support the specification of an application-defined data value, which can be queued to a process associated with a signal, and then retrieved from the return value of the `Await_Signal` function (see 3.3.16).

When a signal is generated by the `Queue_Signal` procedure or any signal-generating operation that supports the specification of an application-defined data value, the signal shall be marked pending and, if queueing is enabled for that signal, the signal shall be queued to the process along with the application specified data value. Multiple occurrences of signals so generated shall be queued in FIFO order. If queueing is not enabled for the signal, it is unspecified whether signals so generated are queued, or what data value, if any, is queued with them.

Signals generated by the `Send_Signal` function or other events that cause signals to occur (such as detection of hardware faults or terminal activity) and for which the implementation does not support signal queueing shall have no effect on occurrences of the signal already queued for the same signal.

When multiple unblocked signals, all in the range `Realtime_Signal`, are pending, the behavior shall be as if the implementation delivers the pending, unblocked signal

with the lowest signal number within that range. No other ordering of signal delivery is specified.

If, when a pending signal occurrence is delivered, there are additional occurrences of that signal queued, they shall remain pending. Otherwise, the pending indication shall be removed.

3.3.2 Signal Type

3.3.2.1 Synopsis

```
type Signal is implementation-defined-integer;;
function Image (Sig : Signal) return String;
function Value (Str : String) return Signal;
```

3.3.2.2 Description

The type `Signal` shall be used to represent signals. The mapping from signals to values of type `Signal` is implementation defined, subject to the requirement that it be one-to-one.

The values of the type `Signal` shall represent valid signals in the implementation.

`Image` shall return a string identifying the signal specified by the parameter `Sig`. If `Sig` is the value of one of the signals defined by this standard, the value returned by `Image` shall be the identifier of the corresponding long-name constant, in uppercase. Otherwise, the value returned by `Image` is implementation defined, subject to the requirement that it shall be distinct for each signal supported by the implementation.

`Value` shall return the value of type `Signal` corresponding to the parameter `Str`. If `Str` matches either the short name or the long name of an identifier for a signal supported by the implementation, ignoring leading and trailing blanks and adjusting the case of letters as necessary, `Value` shall return the corresponding signal value. Otherwise, if `Str` does not match the image of any signal supported by the implementation, `Value` shall raise `Constraint_Error`. In any case, for every signal `S` supported by the implementation, `Value(Image(S))=S`.

3.3.2.3 Error Handling

`Constraint_Error` shall be raised by `Value` if `Str` does not match the image of any of the signals supported by the implementation.

No exceptions shall be raised by `Image`.

3.3.3 Standard Signals

3.3.3.1 Synopsis

```
Signal_Null,
SIGNULL : constant Signal := 0;
Signal_Abort,
SIGABRT : constant Signal := implementation-defined;
Signal_Alarm,
SIGALRM : constant Signal := implementation-defined;
```

```
   Signal_Bus_Error,
   SIGBUS :  constant Signal := implementation-defined;
   Signal_Floating_Point_Error,
   SIGFPE :  constant Signal := implementation-defined;
   Signal_Hangup,
   SIGHUP :  constant Signal := implementation-defined;
   Signal_Illegal_Instruction,
   SIGILL :  constant Signal := implementation-defined;
   Signal_Interrupt,
   SIGINT :  constant Signal := implementation-defined;
   Signal_Kill,
   SIGKILL : constant Signal := implementation-defined;
   Signal_Pipe_Write,
   SIGPIPE : constant Signal := implementation-defined;
   Signal_Quit,
   SIGQUIT : constant Signal := implementation-defined;
   Signal_Segmentation_Violation,
   SIGSEGV : constant Signal := implementation-defined;
   Signal_Terminate,
   SIGTERM : constant Signal := implementation-defined;
   Signal_User_1,
   SIGUSR1 : constant Signal := implementation-defined;
   Signal_User_2,
   SIGUSR2 : constant Signal := implementation-defined;
   Signal_Child,
   SIGCHLD : constant Signal := implementation-defined;
   Signal_Continue,
   SIGCONT : constant Signal := implementation-defined;
   Signal_Stop,
   SIGSTOP : constant Signal := implementation-defined;
   Signal_Terminal_Stop,
   SIGTSTP : constant Signal := implementation-defined;
   Signal_Terminal_Input,
   SIGTTIN : constant Signal := implementation-defined;
   Signal_Terminal_Output,
   SIGTTOU : constant Signal := implementation-defined;
   Signal_IO,
   SIGIO :   constant Signal := implementation-defined;
   Signal_Out_Of_Band_Data,
   SIGURG :  constant Signal := implementation-defined;
   subtype Realtime_Signal is Signal range implementation-defined;
```

c

3.3.3.2 Description

The *signals defined by this standard* are `Signal_Null`, the required signals, the job control signals, the memory protection signal, the realtime signals, and the sockets DNI signals.

c

The *required signals* are as follows:

`Signal_Abort`

The abnormal process termination signal. It is a reserved signal. The Ada language implementation may use this signal in the implementation of the Ada **`abort`** operation.

`Signal_Alarm`

The timeout signal. It is a reserved signal. The Ada language implementation may use this signal in the implementation of language-defined operations with timeouts.

`Signal_Floating_Point_Error`

The signal for detection of an erroneous arithmetic operation, such as division by zero or an operation resulting in overflow. It is a reserved signal. The Ada language implementation may translate this signal to `Constraint_Error`.

`Signal_Hangup`

The signal for hang-up detected on the controlling terminal or the termination of the controlling process. (See 7.1 and 3.1.3.)

`Signal_Illegal_Instruction`

The signal for detection of an invalid hardware instruction. It is a reserved signal. The Ada language implementation may translate this signal to an exception. If `Ada.Exceptions` is supported, the value of `Exception_Message` shall be the string `"Signal_Illegal_Instruction"`.

`Signal_Interrupt`

The interactive attention signal. (See 7.2.1.)

`Signal_Kill`

The process termination signal. It cannot be accepted, caught, or ignored.

`Signal_Pipe_Write`

The signal for a write operation to a pipe or a socket that has no readers. (See 6.1.4 and 18.4.13.1.)

c

`Signal_Quit`

The interactive termination signal. (See 7.2.1.)

`Signal_Segmentation_Violation`

The signal for detection of an invalid memory reference. It is a reserved signal. Any occurrences that are not identifiable by the Ada language implementation as corresponding to checks that require some other exception to be raised shall be mapped to the exception `Program_Error`, and the `Exception_Message` value shall be the string `"Signal_Segmentation_Violation"`.

`Signal_Terminate`

A process termination signal.

`Signal_User_1`

A signal set aside for the use of applications; that is, the implementation shall not generate this signal unless it is explicitly sent by an application.

`Signal_User_2`

A signal set aside for the use of applications; that is, the implementation shall not generate this signal unless it is explicitly sent by an application.

The *job control signals* are

`Signal_Child`

The signal indicating a child process has terminated or stopped.

`Signal_Continue`

The signal to continue if stopped.

`Signal_Stop`

The signal to stop. It cannot be accepted, caught, or ignored.

`Signal_Terminal_Stop`

The interactive stop signal. (See 7.1.0.12.)

`Signal_Terminal_Input`

The signal for when a read from the controlling terminal is attempted by a member of the background process group. (See 7.1.0.7.)

`Signal_Terminal_Output`

The signal for when a write to the controlling terminal is attempted by a member of the background process group. (See 7.1.0.7.)

The *memory protection signal* is

`Signal_Bus_Error`

The signal indicating access to an undefined portion of a memory object. It is a reserved signal. The Ada language implementation shall translate this signal to the exception `Program_Error`. If Ada 95 is supported, the value of `Exception_Message` shall be the string `"Signal_Bus_Error"`.

The *realtime signals* are those in the range of the subtype `Realtime_Signal`.

The required support for the realtime signals is as follows:

— All the required signals shall be supported by every implementation.

— *If the Job Control option is supported:* Job control signals shall be supported.

— *If the Memory Protection option is supported:* The memory protection signal shall be supported.

— *If the Realtime Signals option is supported:* The realtime signals shall be supported.

If the Realtime Signals option is supported: The range `Realtime_Signal` shall include at least `POSIX_Limits.Portable_Realtime_Signals_Maximum` values, and shall not overlap with the named signals defined in this standard. It shall specify a range of signal numbers that are reserved for application use, and for which the realtime signal behavior specified in this standard shall be supported.

If the Realtime Signals option is not supported, the range of `Realtime_Signal` is unspecified.

The *Sockets DNI signals* are

`Signal_IO`

The signal indicating that an I/O event occurred on an asynchronous socket (see 18.2.6).

`Signal_Out_Of_Band_Data`

The signal indicating detection of expedited or out-of-band data on a communication endpoint (see 18.2.5.5).

c

The implementation may support other signals in addition to the signals defined by this standard.

For signals that are supported, the implementation shall behave in accordance with this standard. For other signals, the implementation shall not generate these signals, and attempts to send these signals or to examine or specify their signal actions shall raise `POSIX_Error`.

It is implementation defined whether the realtime signal behavior specified in this standard (specifically, the queueing of signals and the passing of application defined data values) is supported for signals outside of the range `Realtime_Signal`.

An Ada application is not permitted to explicitly accept, block, catch, or ignore any of the reserved signals (see 2.2.2.155). (A process executing an Ada program may accept, block, catch, or ignore the reserved signals, but only indirectly, via actions of the Ada runtime system.) The effect of any of these signals being generated for the process of an Ada active partition, or for any task within it, is implementation defined.

It is not permitted to accept, catch, or ignore `Signal_Kill` or `Signal_Stop`.

An implementation shall not impose restrictions on the ability of an application to send, accept, block, or ignore the signals defined by this standard, except as specified in this standard.

Some of the signals defined by this standard are named by the constants of type `Signal` declared in this package. The values of these constants are implementation defined. No additional names are permitted to be declared in the visible part of the specification of any package defined by this standard. Two names are provided for each signal. The short name is defined for historical reasons; use of the long name in an Ada program is preferred. `Signal_Null` is a value of type `Signal` that cannot be sent, accepted, or caught. It shall have the value zero.

NOTE: In other words, the representation of the value of `Signal_Null` shall be the same as the integer zero.

3.3.4 Default Signal Actions

The following default signal actions shall be supported:

Terminate
: Terminate the process abnormally.

Ignore
: Ignore the signal.

Stop
: Stop the process.

Continue
: Continue the process if it is currently stopped; otherwise, ignore the signal.

If the default action is to stop the process, the execution of that process (including all tasks within it) shall be temporarily suspended. When a process stops, a `Signal_Child` signal shall be generated for its parent process, unless the parent process has disabled this feature, by calling `Set_Stopped_Child_Signal` with parameter `Enable` set to `False`. (See 3.3.10.)

While a process is stopped, any additional signals (except `Signal_Continue`) that are sent to the process shall not be delivered until the process is continued, except `Signal_Kill`, which shall always terminate the receiving process. A process that is a member of an orphaned process group shall not be allowed to stop in response to the `Signal_Terminal_Stop`, `Signal_Terminal_Input`, or `Signal_Terminal_Output` signals. In cases where delivery of one of these signals would stop such a process, the signal shall be discarded.

The default signal action for all the required signals defined by this standard that are not reserved signals shall be to terminate the process abnormally. Terminating the process associated with an Ada active partition shall cause the active partition and all the tasks within it to cease execution permanently.

The default action for the reserved signals is implementation defined.

The default actions for the job-control signals shall be as shown in Table 3.1.

Table 3.1 – Default Actions for Job Control Signals

Signal	Default Action
`Signal_Child`	Ignore
`Signal_Continue`	Continue
`Signal_Stop`	Stop
`Signal_Terminal_Stop`	Stop
`Signal_Terminal_Input`	Stop
`Signal_Terminal_Output`	Stop

The default signal actions for the realtime signals shall be to terminate the process abnormally.

The default action for the Sockets DNI signals shall be to ignore the signal. |c

3.3.5 Tasking Safety

In general, a signal may cause an Ada task to preempt another Ada task directly through unblocking of a task that was blocked on a signal-awaiting operation or indirectly through reserved signals that are caught by the Ada language implementation. Except for operations where special restrictions are explicitly imposed for safe use with Ada tasking (for example, see `Fork` in 3.2.1), all POSIX operations shall be tasking-safe; that is, they shall be safely callable from any place within an Ada task.

NOTE: The requirement for tasking safety does not apply if Ada code is executed via an asynchronous signal handler, as it might be via an interface not defined by this standard.

NOTE: The requirement for tasking safety does not imply any greater degree of safety for concurrent use than is required of the standard Ada libraries by the Ada RM {1}. That is, unless it is so specified elsewhere in this standard, operations are not necessarily atomic and are not necessarily safe to execute concurrently on the same data object.

All operations defined by this standard shall also be safe for asynchronous abortion. To ensure safety, the implementation may defer abortion during operations that are not specified as being interruptible.

3.3.6 Interruptibility

Certain operations defined by this standard are defined to be *interruptible operations*. If a signal is delivered to a task while it is executing one of these operations, the behavior of the operation may be affected. If the signal action of the signal is to terminate the process, the process shall be terminated and the operation shall not return. If the action of the signal is to stop the process, the process shall stop until continued or terminated. Subsequent generation of a `Signal_Continue` signal for the stopped process shall cause the process to be continued, and the original operation shall continue at the point where the process was stopped. If the action of the signal is to invoke an asynchronous signal handler, the signal handler shall be invoked; in this case, the original operation is said to be *interrupted by* the signal. When the signal handler returns, the behavior of the interrupted operation shall be as described individually for that operation.

These rules shall also apply to reserved signals that are caught by the Ada language implementation, unless they are translated by the Ada runtime system into an exception to satisfy a requirement of this standard or the Ada RM {1}.

Signals that are ignored shall not affect the behavior of any operation.

Signals that are blocked shall not affect the behavior of any operation until they are delivered.

The interruptible operations defined in this standard are

— `POSIX_Asynchronous_IO.Await_IO_Or_Timeout`
— `POSIX_Asynchronous_IO.Await_IO`
— `POSIX_Asynchronous_IO.List_IO_Wait`
— `POSIX_Event_Management.Poll`
— `POSIX_Event_Management.Select_File`
— `POSIX_File_Locking.Wait_to_Set_Lock`
— `POSIX_Generic_Shared_Memory.Open_And_Map_Shared_Memory`
— `POSIX_Generic_Shared_Memory.Open_Or_Create_And_Map_Shared_Memory`
— `POSIX_IO.Open_Or_Create`
— `POSIX_IO.Open`
— `POSIX_IO.Duplicate`
— `POSIX_IO.Close`
— `POSIX_IO.Read`
— `POSIX_IO.Write`
— `POSIX_Message_Queues.Generic_Message_Queues.Send`
— `POSIX_Message_Queues.Generic_Message_Queues.Receive`

— POSIX_Message_Queues.Open
— POSIX_Message_Queues.Open_Or_Create
— POSIX_Message_Queues.Send
— POSIX_Message_Queues.Receive
— POSIX_Process_Primitives.Wait_For_Child_Process
— POSIX_Semaphores.Wait (optionally interruptible)
— POSIX_Shared_Memory_Objects.Open_Or_Create_Shared_Memory
— POSIX_Shared_Memory_Objects.Open_Shared_Memory
— POSIX_Signals.Await_Signal
— POSIX_Signals.Await_Signal_Or_Timeout
— POSIX_Sockets.Accept_Connection
— POSIX_Sockets.Bind
— POSIX_Sockets.Connect
— POSIX_Sockets.Create
— POSIX_Sockets.Create_Pair
— POSIX_Sockets.Get_Peer_Name
— POSIX_Sockets.Get_Socket_Broadcast
— POSIX_Sockets.Get_Socket_Debugging
— POSIX_Sockets.Get_Socket_Error_Status
— POSIX_Sockets.Get_Socket_Keep_Alive
— POSIX_Sockets.Get_Socket_Linger_Time
— POSIX_Sockets.Get_Socket_Name
— POSIX_Sockets.Get_Socket_No_Routing
— POSIX_Sockets.Get_Socket_OOB_Data_Inline
— POSIX_Sockets.Get_Socket_Receive_Buffer_Size
— POSIX_Sockets.Get_Socket_Receive_Low_Water_Mark
— POSIX_Sockets.Get_Socket_Receive_Timeout
— POSIX_Sockets.Get_Socket_Reuse_Addresses
— POSIX_Sockets.Get_Socket_Send_Buffer_Size
— POSIX_Sockets.Get_Socket_Send_Low_Water_Mark
— POSIX_Sockets.Get_Socket_Send_Timeout
— POSIX_Sockets.Get_Socket_Type
— POSIX_Sockets.Listen
— POSIX_Sockets.Receive
— POSIX_Sockets.Receive_Message
— POSIX_Sockets.Send

— POSIX_Sockets.Send_Message
— POSIX_Sockets.Set_Socket_Broadcast
— POSIX_Sockets.Set_Socket_Debugging
— POSIX_Sockets.Set_Socket_Keep_Alive
— POSIX_Sockets.Set_Socket_Linger_Time
— POSIX_Sockets.Set_Socket_No_Routing
— POSIX_Sockets.Set_Socket_OOB_Data_Inline
— POSIX_Sockets.Set_Socket_Receive_Buffer_Size
— POSIX_Sockets.Set_Socket_Receive_Low_Water_Mark
— POSIX_Sockets.Set_Socket_Receive_Timeout
— POSIX_Sockets.Set_Socket_Reuse_Addresses
— POSIX_Sockets.Set_Socket_Send_Buffer_Size
— POSIX_Sockets.Set_Socket_Send_Low_Water_Mark
— POSIX_Sockets.Set_Socket_Send_Timeout
— POSIX_Sockets.Shutdown
— POSIX_Sockets.Socket_Is_At_OOB_Mark
— POSIX_Terminal_Functions.Set_Terminal_Characteristics
— POSIX_Terminal_Functions.Drain
— POSIX_XTI.Accept_Connection
— POSIX_XTI.Acknowledge_Orderly_Release
— POSIX_XTI.Acknowledge_Orderly_Release_With_Data
— POSIX_XTI.Bind
— POSIX_XTI.Close
— POSIX_XTI.Confirm_Connection
— POSIX_XTI.Connect
— POSIX_XTI.Gather_And_Send_Data
— POSIX_XTI.Gather_And_Send_Data_Unit
— POSIX_XTI.Get_Current_State
— POSIX_XTI.Get_Info
— POSIX_XTI.Get_Protocol_Address
— POSIX_XTI.Initiate_Orderly_Release
— POSIX_XTI.Initiate_Orderly_Release_With_Data
— POSIX_XTI.Listen
— POSIX_XTI.Look
— POSIX_XTI.Manage_Options
— POSIX_XTI.Open

— POSIX_XTI.Receive
— POSIX_XTI.Receive_And_Scatter_Data
— POSIX_XTI.Receive_And_Scatter_Data_Unit
— POSIX_XTI.Receive_Data_Unit
— POSIX_XTI.Retrieve_Data_Unit_Error
— POSIX_XTI.Retrieve_Disconnect_Info
— POSIX_XTI.Send
— POSIX_XTI.Send_Data_Unit
— POSIX_XTI.Send_Disconnect_Request
— POSIX_XTI.Synchronize_Endpoint
— POSIX_XTI.Unbind

c

Return from a call to an interruptible POSIX operation, whether by normal return or exception propagation, shall be an abort completion point, as defined in 9.8 (15) of the Ada RM {1}.

While a task is blocked in an interruptible POSIX operation with the Signal_Masking value of No_Signals or RTS_Signals, abort of the task shall cause the operation to be interrupted.

NOTE: A consequence of these rules is that the task must reach an abort completion point, since interruption is required to raise POSIX_Error, and propagation of the exception is an abort completion point. Whether the task is aborted at that point depends on whether the task is executing an abort deferred operation.

Unless it is stated otherwise in the description of the specific operation, it is a consequence of the rules stated above that a Strictly Conforming POSIX.5 Application must assume that every invocation of an interruptible operation may raise POSIX_Error with error code Interrupted_Operation, unless the application has explicitly blocked delivery of signals.

Interruptible operations are provided with a parameter Masked_Signals to permit the application to block signals during the operation. (See 2.4.1.6.) If the value of Masked_Signals is No_Signals, the operation shall be interruptible by all signals not already blocked or ignored. If the value of Masked_Signals is All_Signals, the operation shall not be interrupted. If the value of Masked_Signals is RTS_Signals, then the operation shall not be interrupted by reserved signals, except for Signal_Abort which may be unblocked if it is used by the Ada language implementation of abort. The operation shall be interruptible by any other signals not already explicitly blocked by the application.

NOTE: The Masked_Signals parameter is the only way an application can protect an interruptible call against interruption by the reserved signals. (The procedures Block_Signals and Ignore_Signal do not affect the reserved signals.) Therefore, if Masked_Signals is No_Signals it is advisable for the application to provide an exception handler for POSIX_Error.

3.3.7 Signal Sets

3.3.7.1 Synopsis

```
type Signal_Set is private;
procedure Add_Signal
   (Set : in out Signal_Set;
    Sig : in Signal);
procedure Add_All_Signals (Set : in out Signal_Set);
procedure Delete_Signal
   (Set : in out Signal_Set;
    Sig : in Signal);
procedure Delete_All_Signals (Set : in out Signal_Set);
function Is_Member
   (Set : Signal_Set;
    Sig : Signal)
  return Boolean;
```

3.3.7.2 Description

The type Signal_Set is used to represent sets of signals. It is used to indicate signals to be accepted or blocked. Objects of type Signal_Set shall be implicitly initialized to include no signals.

Add_Signal shall add the signal specified by Sig to the set of signals specified by Set. Any other members of the set shall remain.

Add_All_Signals shall update Set so that it includes all values of the type Signal.

Delete_Signal shall update Set so that it does not include the signal specified by Sig. No other signals shall be deleted from the set.

Delete_All_Signals shall update the set specified by Set so that it includes no signals.

NOTE: Add_All_Signals adds all the implementation-defined signals and Delete_All_Signals deletes all implementation defined signals, as well as all the signals defined in this standard.

Is_Member shall return the value True if and only if the set specified by the parameter Set includes the signal specified by the parameter Sig or the value of the parameter Sig is Signal_Null.

3.3.7.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Add_Signal, Delete_Signal, or Is_Member was called with a value of Sig that is not a valid signal in this implementation.

Delete_All_Signals and Add_All_Signals shall not raise any exception in this case.

3.3.8 Block and Unblock Signals

3.3.8.1 Synopsis

```
procedure Set_Blocked_Signals
   (New_Mask : in Signal_Set;
    Old_Mask : out Signal_Set);
procedure Block_Signals
   (Mask_to_Add : in Signal_Set;
    Old_Mask :    out Signal_Set);
procedure Unblock_Signals
   (Mask_to_Subtract : in Signal_Set;
    Old_Mask :         out Signal_Set);
function Blocked_Signals return Signal_Set;
```

3.3.8.2 Description

Operations are provided to change and inquire about the signal mask of the calling task.

NOTE: The main intended use of these operations is with the default signal action, in which case the signal should be kept blocked in all but one task. Blocking and unblocking the signal in that one task will control when the default action may occur (for the whole process). In contrast, it is recommended that applications keep all signals that are used with signal-awaiting operations blocked in all tasks at all times.

The `Set_Blocked_Signals` operation shall replace the specified signal mask by the parameter `New_Mask` and return in the parameter `Old_Mask` the value that is replaced. The operation shall be atomic, even in the presence of multiple Ada tasks.

The `Block_Signals` operation shall add the signals specified by the parameter `Mask_to_Add` to the current signal mask and return in the parameter `Old_Mask` the current mask before this operation. The operation shall be atomic, even in the presence of multiple Ada tasks.

The `Unblock_Signals` operation shall remove any signals specified by the parameter `Mask_to_Subtract` from the current mask and return in the parameter `Old_Mask` the current mask before this operation. The operation shall be atomic, even in the presence of multiple Ada tasks.

`Set_Blocked_Signals`, `Block_Signals`, and `Unblock_Signals` may raise `POSIX_Error` if the effect would be to unblock a signal that is currently unblocked by another task in the same process.

NOTE: It is anticipated that a future revision of this standard may remove the implementation permission stated in the paragraph above.

`Blocked_Signals` shall return the current signal mask of the calling task.

It is not possible to block the signals `Signal_Kill` and `Signal_Stop`; this requirement shall be enforced by the system without raising an exception in any of the operations specified in this clause. Similarly, it is not possible for an application to block the reserved signals; this requirement shall be enforced by the implementation of this standard without raising an exception in any of the operations specified in this clause.

If the implementation supports additional signals not defined in package POSIX_Signals, the effect of masking all signals shall include the masking of all maskable implementation defined signals as well as all maskable POSIX signals.

NOTE: The recommended practice for multitasking applications is that at most one task *be responsible for managing the signal actions and blocking/unblocking of each signal.*

3.3.8.3 Error Handling

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Permitted

An attempt was made to unblock a signal that was already unblocked by another task in the same process.

3.3.9 Ignore Signals

3.3.9.1 Synopsis

```
procedure Ignore_Signal (Sig : in Signal);
procedure Unignore_Signal (Sig : in Signal);
procedure Restore_Default_Action (Sig : in Signal)
  renames Unignore_Signal;
function Is_Ignored (Sig : Signal) return Boolean;
procedure Install_Empty_Handler (Sig : in Signal);
```
C

3.3.9.2 Description

Ignore_Signal shall cause the signal action for the signal specified by the parameter Sig to be set to ignore. (See 3.3.1.)

Unignore_Signal shall cause the signal action for the signal specified by the parameter Sig to be set to the default action for that signal.

An Ada application is not permitted to explicitly set the signal action for any of the reserved signals (see 2.2.2.155).

A process is also not permitted to accept, catch, or ignore Signal_Kill or Signal_Stop. An attempt to set the action for any of the signals named in this paragraph, whether to ignore or not ignore, shall cause POSIX_Error to be raised with error code Invalid_Argument.

Is_Ignored shall return True if and only if the signal action for the signal specified by the parameter Sig is to ignore it.

The Install_Empty_Handler operation shall have the effect of installing an empty asynchronous handler procedure that simply executes and returns, so that the action associated with the signal is for this empty handler to catch the signal. When a signal is generated for a task (or for a process, respectively) and an empty asynchronous handler procedure has been installed for the process containing the task, the effect of the signal depends on whether the signal is currently masked by the task (or by all the tasks in the process, respectively), as follows:
C

— When the signal is unmasked, the signal shall be delivered to the empty handler, which will execute and return, possibly having the side effect of interrupting any blocked system call in the same task.

— When the signal is masked, the signal shall be held pending on the process or task.

The effect of `Install_Empty_Handler` shall be overridden by any subsequent call to `Ignore_Signal` or `Unignore_Signal`.

NOTE: This operation is provided in order to allow an application to accept a signal whose default action is to ignore the signal. By first calling `Ignore_Signal` for that signal, an Ada application can ensure that the signal will not be discarded while the signal is blocked.

C

3.3.9.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

`Ignore_Signal`, `Unignore_Signal`, or `Is_Ignored` was called with a value for the parameter `Sig` that specifies a signal that is invalid or not supported by the implementation or a signal for which the signal action is not permitted to be set by the application.

3.3.10 Controlling Generation of Signal for Child Process

3.3.10.1 Synopsis

```
procedure Set_Stopped_Child_Signal (Enable : in Boolean := True);
function Stopped_Child_Signal_Enabled return Boolean;
```

3.3.10.2 Description

`Set_Stopped_Child_Signal` shall control the generation of the `Signal_Child` signal if the implementation supports the `Signal_Child` signal. If the parameter `Enable` has the value `True`, the `Signal_Child` signal shall be generated for the calling process whenever any of its child processes stop. If `Enable` is `False`, the implementation shall not generate the `Signal_Child` signal in this way.

NOTE: The effect of calling this procedure with `Enable=>False` is not the same as blocking or ignoring `Signal_Child` since that signal will still be delivered if it is generated by means other than the stopping of a child process.

`Stopped_Child_Signal_Enabled` shall return `True` if and only if the signal specified by `Signal_Child` will be generated for the calling process whenever any of its child processes stop.

3.3.10.3 Error Handling

No exceptions shall be raised by these operations.

3.3.11 Examine Pending Signals

3.3.11.1 Synopsis

```
function Pending_Signals return Signal_Set;
```

3.3.11.2 Description

`Pending_Signals` shall return the set of signals that are blocked from delivery and are pending for either the process or the calling task.

NOTE: The set of signals returned should be the union of the set of signals pending for the process and the set of signals pending for the calling task.

3.3.11.3 Error Handling

No exceptions shall be raised by this operation.

3.3.12 Signal Event Notification

3.3.12.1 Synopsis

```
type Signal_Event is private;
type Signal_Data  is private;
type Notification is range implementation-defined;
No_Notification :     constant Notification := implementation-defined;
Signal_Notification : constant Notification := implementation-defined;
function Get_Signal (Event : Signal_Event) return Signal;
procedure Set_Signal
   (Event : in out Signal_Event;
    Sig :   in Signal);
function Get_Notification (Event : Signal_Event) return Notification;
procedure Set_Notification
   (Event :  in out Signal_Event;
    Notify : in Notification);
function Get_Data (Event : Signal_Event) return Signal_Data;
procedure Set_Data
   (Event : in out Signal_Event;
    Data :  in Signal_Data);
```

3.3.12.2 Description

Values of `Notification` are used to specify the notification mechanism to use when an asynchronous event occurs. This standard defines the two following constants of this type:

`No_Notification`

Specifies that no asynchronous notification shall be delivered when the event of interest occurs.

`Signal_Notification`

Specifies that the signal specified by the Signal attribute associated with the event shall be sent to the process when the event of interest occurs.

If the Realtime Signals option is supported: If queueing is enabled for that signal, then the signal will be queued to the process and the Data attribute of the event shall be the Data attribute of the generated signal occurrence. If queueing is not enabled for that signal, it is unspecified whether the signal is queued and what Data attribute, if any, is sent.

An implementation may add notification mechanisms to the Notification type, as permitted in 1.3.1.1 (2). Such extensions, which may change the behavior of the application with respect to this standard when those attributes are uninitialized, also require that the extension be enabled as required by 1.3.1.1.

The Signal_Event type is used to specify how notification of an event is to be delivered. A value of this type has (at least) the following attributes:

Notification

The notification mechanism to use. This value is of type Notification.

Signal

The signal to be generated if the specified notification mechanism makes use of a signal.

Data

The information to be sent with the signal if the signal is one for which signal queueing is supported and enabled. The Data value is the application-defined data value to be passed as part of the Signal_Info return value of one of the Await_Signal functions (see 3.3.14).

The type Signal_Data is used as a carrier for untyped information. The size of this type shall be large enough to hold any value of type Integer or type System.Address. Instantiation of Unchecked_Conversion shall be supported for conversions between the type Signal_Data and the types System.Address, Standard.Integer, and POSIX_Timers.Timer_ID. Unchecked conversion of a value of one of the latter types to Signal_Data and then back to the original type shall yield the original value back again.

The function whose name is of the form Get_X, where X is the name of one of the attributes of Signal_Event, shall return the value of the X attribute of the argument.

The procedure whose name is of the form Set_X shall set the X attribute of the object specified by the first argument to the value specified by the second argument.

3.3.12.3 Error Handling

No error conditions are specified for these operations.

3.3.13 Signal Information

3.3.13.1 Synopsis

```
type Signal_Source is range implementation-defined;
From_Send_Signal :   constant Signal_Source := implementation-defined;
From_Queue_Signal :  constant Signal_Source := implementation-defined;
From_Timer :         constant Signal_Source := implementation-defined;
From_Async_IO :      constant Signal_Source := implementation-defined;
From_Message_Queue : constant Signal_Source := implementation-defined;
type Signal_Info is private;
function Get_Signal (Info : Signal_Info) return Signal;
procedure Set_Signal
   (Info : in out Signal_Info;
    Sig :  in Signal);
function Get_Source (Info : Signal_Info) return Signal_Source;
```

```
procedure Set_Source
   (Info :   in out Signal_Info;
    Source : in Signal_Source);
function Has_Data (Source : Signal_Source) return Boolean;
function Get_Data (Info : Signal_Info) return Signal_Data;
procedure Set_Data
   (Info : in out Signal_Info;
    Data : in Signal_Data);
```

3.3.13.2 Description

The functionality described in this subclause is optional. If the Realtime Signals option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

Values of `Signal_Source` are used to identify the cause, or source of a signal occurrence. The following constants of type `Signal_Source` are defined by this standard, with the indicated specifications:

`From_Send_Signal`
: The signal was generated by the `Send_Signal` procedure.

`From_Queue_Signal`
: The signal was generated by the `Queue_Signal` procedure.

`From_Timer`
: The signal was generated by the expiration of a timer (see 14.1).

`From_Async_IO`
: The signal was generated by the completion of an AIO request (see 6.3).

`From_Message_Queue`
: The signal was generated by the arrival of a message on an empty message queue for which notification via signal was requested (see 15.1).

The `Signal_Info` type is used to specify information that can be associated with an occurrence of a signal. It has the following attributes:

Signal
: The signal of the occurrence.

Source
: The cause, or source, that generated the signal. If the signal was generated by one of the operations or events listed in this subclause (above), the value of Source shall be as specified for that operation or event. If the signal was not generated by one of the operations or events listed in this subclause, the Source attribute of a signal occurrence shall be an implementation-defined value that is not equal to any of the values defined in this subclause.

Data
: Additional information that is provided for certain signal sources. If Source is equal to one of `From_Queue_Signal`, `From_Timer`, `From_Async_IO`, or `From_Message_Queue`, Data shall be the application specified signal data value. Otherwise, the Data attributed is undefined.

An implementation may add other attributes to the Signal_Info type, as permitted in 1.3.1.1 (2). Such extensions, which may change the behavior of the application with respect to this standard when those attributes are uninitialized, also require that the extension be enabled as required by 1.3.1.1.

The function whose name is of the form Get_X, where X is the name of one of the attributes of Signal_Info, shall return the value of the X attribute of the argument.

The procedure whose name is of the form Set_X shall set the X attribute of the object specified by the first argument to the value specified by the second argument.

Has_Data shall return True if and only if Source is a signal source for which the Data attribute is defined.

3.3.13.3 Error Handling

No error conditions are specified for these operations.

3.3.14 Control Signal Queueing

3.3.14.1 Synopsis

```
procedure Enable_Queueing (Sig : in Signal);
procedure Disable_Queueing (Sig : in Signal);
```

3.3.14.2 Description

The functionality described in this subclause is optional. If the Realtime Signals option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

Enable_Queueing and Disable_Queueing are used to control whether data are queued for delivery with the designated signal by the functions that are able to do so. See Queue_Signal (3.3.19), Request_Notify (15.1.8), Create_Timer (14.1.5), and the AIO operations (6.3).

Enable_Queueing shall enable queueing for the specified signal.

Disable_Queueing shall disable queueing for the specified signal. It is unspecified whether occurrences already queued for the signal are retained.

If queueing is enabled for the signal, a signal-awaiting operation for the signal shall return a value of type Signal_Info, which is used to pass information about the event associated with a signal occurrence.

The initial state of signal queuing for all signals in a new process is unspecified.

3.3.14.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Sig is not a signal that is supported by the implementation, or Sig is a signal that is not permitted to be accepted or caught.

Operation_Not_Supported

The implementation does not support the Realtime Signals option.

3.3.15 Wait for Signal

3.3.15.1 Synopsis

```
function Await_Signal (Set : Signal_Set) return Signal;
function Await_Signal_Or_Timeout
   (Set :     Signal_Set;
    Timeout : POSIX.Timespec)
  return Signal;
```

3.3.15.2 Description

NOTE: The behavior of Await_Signal *is defined first. Then, the behavior of* Await_Signal_Or_Timeout *is defined by analogy.*

Await_Signal selects a pending signal from Set, clears it from the set of pending signals in the system, and returns that signal number. The selection and clearing of the signal from the pending set shall be done atomically. If no signal in Set is pending at the time of the call, the calling task shall remain blocked until one or more signals specified in Set becomes pending.

The effect of a call to Await_Signal is undefined if any of the signals in Set are not blocked from delivery to all tasks within the process at the time of the call. Otherwise, after return from a call to Await_Signal, the signals in Set shall be blocked.

The effect of a call to Await_Signal on the signal actions for the signals in Set is unspecified.

A Strictly Conforming POSIX.5 Application should not call Await_Signal unless all the signals in Set are blocked from delivery to all tasks in the process at the time of the call. Thereafter, no task should unblock any of those signals from delivery, until Ignore_Signal or Unignore_Signal has been called for that signal in the interim to redefine the associated signal action.

If prior to the function call there are multiple pending occurrences of a single signal and the Realtime Signals option is not supported, it is implementation defined whether upon return there are any remaining pending occurrences of that signal.

If more than one task is using any of the operations Await_Signal and Await_Signal_Or_Timeout to wait for the same signal, no more than one of these tasks shall return from the function call with the signal number. The task that returns from the function call if more than a single task is waiting is unspecified.

If Set contains any reserved signals, Await_Signal shall raise POSIX_Error.

NOTE: If, while either `Await_Signal` *or* `Await_Signal_Or_Timeout` *is waiting, a signal occurs that is caught by an asynchronous signal handler of the Ada language implementation, and the delivery of the signal causes the wait to be interrupted, the implementation is required to restart the operation.*

The following additional specifications apply if the Realtime Signals option is supported, the following apply:

— When multiple signals in the range `Realtime_Signal` are pending and any of them is selected, it shall be the lowest numbered one. The selection order between realtime and nonrealtime signals or between multiple pending nonrealtime signals is unspecified. If no signal in `Set` is pending at the time of the call, the calling task shall be blocked until one or more signals in `Set` become pending.

— If any signal data value is queued for the selected signal, the first such data value shall be dequeued, and the data value shall be discarded. The system resource used to queue the signal shall be released and made available to queue other signals. If no further signals are queued for the selected signal, the pending indication for that signal shall be reset.

NOTE: To capture information queued with a signal, the functions described in 3.3.16 can be used.

The implementation shall raise `POSIX_Error` if the application attempts to call `Await_Signal` to await a signal that is bound to a task entry.

`Await_Signal_Or_Timeout` shall behave the same as `Await_Signal`, except that if none of the signals specified by `Set` is pending, it shall wait for the time interval specified by `Timeout`. If `Timeout` is zero-valued and if none of the signals specified by `Set` is pending, then `POSIX_Error` shall be raised. If none of the signals specified by `Set` occurs within the time specified by `Timeout`, then `POSIX_Error` shall be raised.

3.3.15.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

At least one of the signals in `Set` is attached to a task entry or is a reserved signal.

`Resource_Temporarily_Unavailable`

No signal specified by `Set` was delivered within the specified timeout period.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

`Set` contains an invalid, or unsupported signal number.

The `Timeout` argument specified an invalid value (*e.g.*, uninitialized data). An implementation should only check for this error if no signal is pending in `Set` and it is necessary to wait.

3.3.16 Wait for Signal with Information

3.3.16.1 Synopsis

```
function Await_Signal (Set : Signal_Set) return Signal_Info;
function Await_Signal_Or_Timeout
   (Set :     Signal_Set;
    Timeout : POSIX.Timespec)
  return Signal_Info;
```

3.3.16.2 Description

The functionality described in this subclause is optional. If the Realtime Signals option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

The effects of these signal-awaiting operations are like those of the signal-awaiting operations defined in 3.3.15, except as follows:

— The Signal attribute of the value returned by the signal-awaiting operation shall be the selected signal.

— The Source attribute of the value returned by the signal-awaiting operation shall be the cause of the signal.

— If any occurrence of the selected signal is queued, the first such occurrence shall be dequeued, and the Data attribute shall be returned as the Data attribute of the value returned by the function. Otherwise, the Data attribute of the value returned is undefined.

NOTE: If the signal was sent by one of the signal-queueing operations defined in this standard and `Has_Data(Get_Source(Info))` *is* `True`*, the Data attribute of* `Info` *will generally be defined. Otherwise, the recipient of the signal must rely on application-specific conventions established between the sender and recipient of the signal.*

3.3.16.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

At least one of the signals in `Set` is attached to a task entry or is a reserved signal.

`Operation_Not_Implemented`

The functions `Await_Signal` and `Await_Signal_Or_Timeout` are not supported by this implementation.

`Resource_Temporarily_Unavailable`

No signal specified by `Set` was delivered within the specified timeout period.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

Invalid_Argument

Set contains an invalid or unsupported signal number.

The Timeout argument specifies an invalid value (*e.g.*, uninitialized data). An implementation should only check for this error if no signal is pending in Set and it is necessary to wait.

3.3.17 Signal Entries

3.3.17.1 Synopsis

```
-- obsolescent
Signal_Abort_Ref :           constant System.Address := implementation-defined;
--  Signal_Floating_Point_Error intentionally omitted.
Signal_Hangup_Ref :          constant System.Address := implementation-defined;
--  Signal_Illegal_Instruction intentionally omitted.
Signal_Interrupt_Ref :       constant System.Address := implementation-defined;
--  Signal_Kill intentionally omitted.
Signal_Pipe_Write_Ref :      constant System.Address := implementation-defined;
Signal_Quit_Ref :            constant System.Address := implementation-defined;
Signal_Terminate_Ref :       constant System.Address := implementation-defined;
Signal_User_1_Ref :          constant System.Address := implementation-defined;
Signal_User_2_Ref :          constant System.Address := implementation-defined;
Signal_Child_Ref :           constant System.Address := implementation-defined;
Signal_Continue_Ref :        constant System.Address := implementation-defined;
--  Signal_Stop intentionally omitted.
Signal_Terminal_Stop_Ref :   constant System.Address := implementation-defined;
Signal_Terminal_Input_Ref :  constant System.Address := implementation-defined;
Signal_Terminal_Output_Ref : constant System.Address := implementation-defined;
function Signal_Reference (Sig : Signal) return System.Address;
```

3.3.17.2 Description

The functionality described in this subclause is optional. If the Signal Entries option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error*. Otherwise, the behavior shall be as specified in this subclause.*

The semantic model for accepting a signal via a task entry is based on the concept of interrupt entry, as defined in I.7.1 of the Ada RM {1} (where it is classified as an obsolescent feature of the Ada language) and is described further in this subclause. The task accepting the signal views it as an interrupt entry call.

The binding of a signal to an entry of a task shall be by means of an Ada address clause. In order to permit an application to bind task entries to signals, there shall be a means for obtaining a value of type System.Address corresponding to each value of type Signal that can be accepted by an Ada application.

The constants defined by this standard with names of the form Signal_*XXX*_Ref shall be usable in an address clause that binds the signal with name Signal_*XXX* to an entry. In addition, for all signals, the function Signal_Reference shall return a value that can be used in an address clause to bind a signal entry for the parameter Sig. If Signal_*XXX* is the name of a signal declared in POSIX_Signals, then Signal_Reference(Signal_*XXX*) = Signal_*XXX*_Ref.

For a signal that is bound to an entry of a task, the effect shall be as if there were a virtual task that uses `Await_Signal` (see 3.3.15) to accept the signal, attempts to rendezvous with the associated entry each time an occurrence of the signal is accepted, and is ready to accept the signal again after the rendezvous. This virtual task shall behave as if it has priority at least as high as that of the task to whose entry the signal is bound.

There may be an implementation-defined limit on the number of these virtual tasks that can simultaneously exist, but there must be at least one for each signal that can be accepted or caught. Even if a limit greater than one is provided, it is unspecified whether a generated signal results in a new entry call and whether such delivery results in another queued virtual task if there is at least one virtual task with an entry call already queued for that signal.

NOTE: In other words, it is unspecified whether multiple generations of a signal can cause the value of the `'Count` attribute of the associated entry to exceed one.

*NOTE: A significant semantic change was made here between POSIX.5 and POSIX.5b in the effect of the signal mask on signal entries. The signal mask can no longer prevent delivery of the signal to a task with an open **`accept`** statement for the corresponding entry. This change was forced by POSIX.1c.*

Clearing pending occurrences of a given signal that is bound to a task entry may abort any entry calls from virtual tasks that are due to previously accepted occurrences of that signal. Clearing pending signals shall have no effect on any calls from normal Ada tasks that may be queued for the entry.

Given the virtual task model described in this subclause, other details of the semantics of signal accepting via task entries are determined by the Ada language standard. For example, the times at which entries are bound to signals are defined by the semantics of Ada task elaboration, and the conditions under which a signal entry call may be accepted are defined by the semantics of rendezvous.

One restriction imposed by the Ada RM {1} is that a task should not attempt to bind an entry to a signal to which there is already another bound entry. If a program attempts to bind an entry to a signal to which another entry is already bound, `Program_Error` shall be raised. However, on termination of a task, the bindings of any of its entries to interrupts shall be broken so that these interrupts shall become available for rebinding. When the binding is broken, the signal action shall revert to the default for that signal.

For the `Ignore_Signal` operation if the signal is bound to a task entry, the effect shall be to discard any pending or subsequent deliveries of that signal. The binding to the entry may remain in force. Any queued signal entry calls to that entry may be discarded. (The discarding of pending signal entry calls may depend on whether the signal is blocked; see 3.3.3).

NOTE: For the `Await_Signal` and `Await_Signal_Or_Timeout` operations, if the parameter `Set` includes a signal to which an interrupt entry is attached, `POSIX_Error` shall be raised with error code `Invalid_Argument`. (See 3.3.15.)

3.3.17.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

`Signal_Reference` is called with a value of `Sig` that is not a valid signal or is a signal for which the implementation does not support user-defined signal entries.

3.3.18 Send a Signal

3.3.18.1 Synopsis

```
procedure Send_Signal
   (Process : in POSIX_Process_Identification.Process_ID;
    Sig :     in Signal);
procedure Send_Signal
   (Group : in POSIX_Process_Identification.Process_Group_ID;
    Sig :   in Signal);
procedure Send_Signal (Sig : in Signal);
```

3.3.18.2 Description

`Send_Signal` shall send the signal specified by the parameter `Sig` to a process or to all the members of a set of processes. The set of processes is specified by the `Process` or `Group` parameter. The form of `Send_Signal` with parameter `Process` shall send the signal to the process whose process ID is equal to `Process`, if the calling process has permission to send the signal to it. The form with parameter `Group` shall send the signal to all the processes (excluding an implementation-defined set of system processes) whose process group ID is equal to `Group` and to which the calling process has permission to send the signal. The form without a parameter `Process` or `Group` shall be equivalent to calling the form with `Group` parameter using `Get_Process_Group_ID` as the parameter value, sending the signal to the process group of the caller.

If `Send_Signal` causes the signal specified by the parameter `Sig` to be generated for the sending process, if the signal is not blocked for the calling task, and if no other task is waiting to accept the signal or has it unblocked, either that signal or at least one pending unblocked signal shall be delivered to the sending process before `Send_Signal` returns.

NOTE: See 3.3.1 for additional requirements on the order of delivery for realtime signals.

If the parameter `Sig` is equal to the value `Signal_Null`, no signal shall be sent, but error checking shall be performed.

NOTE: Sending `Signal_Null` can be used to check the validity of a process ID or process group ID.

For a process to have permission to send a signal to a process, the real or effective user ID of the sending process shall match the real or effective user ID of the receiving process, unless the sending process has appropriate privileges.

If the Saved IDs option is supported: The saved set-user-ID of the receiving process shall be checked in place of its effective user ID.

If the effective user ID of a receiving process has been altered through use of the permission `Set_User_ID`, the implementation may still permit the application to receive a signal sent by the parent process or by a process with the same real user ID. (See 5.1.1.)

If the implementation supports `Signal_Continue`, the user ID tests described in the previous paragraph shall not be applied when sending `Signal_Continue` to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-defined restrictions on the sending of signals, including `Signal_Null`. In particular, the system may deny the existence of some or all of the processes specified by the parameters or silently fail to deliver the signal.

3.3.18.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: The value specified by `Sig` is an invalid signal or a signal not supported by the implementation.

`No_Such_Process`
: The set of processes specified is empty; that is, no process can be found whose process ID is equal to `Process`, or no process group can be found whose process group ID is equal to `Group`.

`Operation_Not_Permitted`
: The process does not have permission to send the signal to any of the processes in the specified set.

3.3.19 Queue a Signal

3.3.19.1 Synopsis

```
procedure Queue_Signal
   (Process : in POSIX_Process_Identification.Process_ID;
    Sig :  in Signal;
    Data : in Signal_Data);
```

3.3.19.2 Description

The functionality described in this subclause is optional. If the Realtime Signals option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this subclause.

`Queue_Signal` shall cause the signal specified by `Sig` to be sent with the data value specified by `Data` to the process specified by `Process`. The conditions required for

a process to have permission to send a signal to another process are the same as for the procedure Send_Signal.

If the receiving process is the same as the sending process, if the signal specified by the parameter Sig is not currently blocked for the calling task, and if no other task is waiting to accept the signal or has it unblocked, either that signal or at least one pending unblocked signal shall be delivered to the sending process before the Queue_Signal operation returns.

NOTE: See 3.3.1 for additional requirements on the order of delivery for realtime signals.

If the signal specified by Sig is currently blocked for the receiving process, Queue_-Signal shall return immediately. If signal queueing is enabled for Sig and if the resources are available to queue the signal, the signal and the data value shall be queued and left pending to the receiving process.

If signal queueing is not enabled for Sig by the specified process, then Sig shall be queued at least once to the receiving process; in this circumstance it is unspecified whether Data shall be sent to the receiving process as a result of this call.

If the parameter Sig is equal to the value Signal_Null, no signal shall be queued, but error checking shall be performed.

NOTE: Queueing Signal_Null can be used to check the validity of a process ID or process group ID.

3.3.19.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Temporarily_Unavailable
: No resources are available to queue the signal. The process has already queued Queued_Signals_Maximum signals that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.

Invalid_Argument
: The value of Sig is an invalid signal or a signal not supported by the implementation.

Operation_Not_Implemented
: The procedure Queue_Signal is not supported by this implementation.

Operation_Not_Permitted
: The requesting process does not have the appropriate privilege to send the signal to the specified receiving process.

No_Such_Process
: Process is not a valid process ID.

3.3.20 Interrupt a Task

3.3.20.1 Synopsis

```
procedure Interrupt_Task (T : in Ada_Task_Identification.Task_ID);
```

3.3.20.2 Description

If the task identified by `T` is blocked in an interruptible POSIX operation with the `Signal_Masking` value of `No_Signals` or `RTS_Signals`, `Interrupt_Task` shall interrupt the POSIX operation by raising `POSIX_Error` with error code `Interrupted_Operation`. If the task identified by `T` is not executing an interruptible operation, this procedure shall have no effect. If the task identified by `T` is executing an interruptible operation but has not yet blocked, `Interrupt_Task` may have no effect.

NOTE: If the `Blocking_Behavior` for a POSIX operation is `Program`, no task will be able to execute the `Interrupt_Task` procedure while any task is blocked in the POSIX operation.

NOTE: This operation may be implemented by sending a signal to the task that is specified by the parameter `T`.

3.3.20.3 Error Handling

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The parameter `T` does not identify a task in the current process. Alternatively, the exception `Program_Error` may be raised.

NOTE: `Program_Error` is specified by C.7.1 (17) of the Ada RM {1} for some uses of invalid values of type `Task_ID`.

Section 4: Process Environment

The environment for a POSIX process provides information about the system state from which the process was started. This section specifies operations that access this information, as defined in the packages POSIX_Process_Identification, POSIX_Process_Times, POSIX_Process_Environment, POSIX_Calendar, and POSIX_Configurable_System_Limits.

4.1 Package POSIX_Process_Identification

This package contains operations to

— Identify self and parent processes

— Set and get the process group ID

— Set the process group ID for job control and create a session

— Get the real and the effective identity of the user and the group of the user

— Get a list of the supplementary groups of the user

— Get the effective user login name

```
with POSIX;
package POSIX_Process_Identification is
   -- 4.1.1 Process Identification Operations
   type Process_ID is private;
   Null_Process_ID :   constant Process_ID;
   System_Process_ID : constant Process_ID;
   function Get_Process_ID return Process_ID;
   function Get_Parent_Process_ID return Process_ID;
   function Image (ID : Process_ID) return Standard.String;
   function Value (Str : Standard.String) return Process_ID;
   -- 4.1.2 Process Group Identification
   type Process_Group_ID is private;
   function Get_Process_Group_ID return Process_Group_ID;
   procedure Set_Process_Group_ID
      (Process :       in Process_ID := Get_Process_ID;
       Process_Group : in Process_Group_ID := Get_Process_Group_ID);
   procedure Create_Process_Group
      (Process :       in  Process_ID;
       Process_Group : out Process_Group_ID);
   procedure Create_Session (Session_Leader : out Process_Group_ID);
   function Image (ID : Process_Group_ID) return Standard.String;
   function Value (Str : Standard.String) return Process_Group_ID;
   -- 4.1.3 User Identification
   type User_ID is private;
   function Get_Real_User_ID return User_ID;
   function Get_Effective_User_ID return User_ID;
   procedure Set_User_ID (ID : in User_ID);
   function Get_Login_Name return POSIX.POSIX_String;
   function Image (ID : User_ID) return Standard.String;
   function Value (Str : Standard.String) return User_ID;
   -- 4.1.4 User and Group Identification
   type Group_ID is private;
   function Get_Real_Group_ID return Group_ID;
   function Get_Effective_Group_ID return Group_ID;
```

```
   procedure Set_Group_ID (ID : in Group_ID);
   subtype Group_List_Index is positive range 1 .. POSIX.Groups_Maxima'Last;
   type Group_List is array (Group_List_Index range <>) of Group_ID;
   function Get_Groups return Group_List;
   function Image (ID : Group_ID) return Standard.String;
   function Value (Str : Standard.String) return Group_ID;
private
   implementation-defined
end POSIX_Process_Identification;
```

4.1.1 Process Identification Operations

4.1.1.1 Synopsis

```
   type Process_ID is private;
   Null_Process_ID :   constant Process_ID;
   System_Process_ID : constant Process_ID;
   function Get_Process_ID return Process_ID;
   function Get_Parent_Process_ID return Process_ID;
   function Image (ID : Process_ID) return Standard.String;
   function Value (Str : Standard.String) return Process_ID;
```

4.1.1.2 Description

The type `Process_ID` shall define the values for process IDs. `Null_Process_ID` is a value of `Process_ID` that shall never represent any process in the system. `System_Process_ID` is a value of `Process_ID` reserved by the system for system processes. A process that is not a system process shall not have this value as its process ID.

`Get_Process_ID` shall return the process ID of the calling process. `Get_Parent_Process_ID` shall return the process ID of the parent of the calling process.

`Image` shall return a character string representing the parameter `ID`. The string representation is implementation defined, but shall be such that if two process IDs are different, then their images shall be different, and if two values of process ID are the same, then their images shall be the same.

`Value` translates the parameter `Str` into a `Process_ID`. `Value` shall translate any string produced by `Image` into the process ID value that was originally the input parameter to `Image`. It is implementation defined whether any other string representation may be successfully translated into a `Process_ID`.

4.1.1.3 Error Handling

`Constraint_Error` shall be raised by `Value` if the parameter `Str` cannot be successfully translated to a `Process_ID`.

No exceptions are specified by this standard for `Get_Process_ID` and `Get_Parent_Process_ID`.

No exception shall be raised by `Image`.

4.1.2 Process Group Identification

4.1.2.1 Synopsis

```
type Process_Group_ID is private;
function Get_Process_Group_ID return Process_Group_ID;
procedure Set_Process_Group_ID
   (Process :       in Process_ID := Get_Process_ID;
    Process_Group : in Process_Group_ID := Get_Process_Group_ID);
procedure Create_Process_Group
   (Process :       in  Process_ID;
    Process_Group : out Process_Group_ID);
procedure Create_Session (Session_Leader : out Process_Group_ID);
function Image (ID : Process_Group_ID) return Standard.String;
function Value (Str : Standard.String) return Process_Group_ID;
```

4.1.2.2 Description

The type `Process_Group_ID` shall define the values for *process group ID*s.

`Get_Process_Group_ID` shall return the process group ID of the calling process.

The effect of `Set_Process_Group_ID` depends on whether the Job Control option is supported.

If the Job Control option is supported, `Set_Process_Group_ID` is used to join an existing process group. Upon successful completion, the process group ID of the process whose process ID matches the parameter `Process` shall be the value of the parameter `Process_Group`.

If the Job Control option is not supported, `Set_Process_Group_ID` shall raise the exception `POSIX_Error` with error code `Operation_Not_Implemented`.

The effect of `Create_Process_Group` depends on whether the Job Control option is supported.

If the Job Control option is supported, `Create_Process_Group` shall make the process specified by `Process` a process group leader. If the specified process was not previously a process group leader, a new process group is created within the session of the calling process with the specified process as the process group leader. The process group ID of a session leader shall not change. If the specified process is already a process group leader, the operation completes successfully without any change. Upon successful completion, the process group ID of the process specified by the parameter `Process` shall be returned as the value of the parameter `Process_Group`.

If the Job Control option is not supported, `Create_Process_Group` shall raise the exception `POSIX.POSIX_Error` with error code `Operation_Not_Implemented`.

`Create_Session` shall create a new session if the calling process is not a process group leader. If a new session is created, the calling process shall

— Be the session leader of the new session

— Be the process group leader of a new process group

— Have no controlling terminal

— Be the only process in the new process group

— Be the only process in the new session

Upon successful completion, `Create_Session` shall return the new process group ID of the calling process.

`Image` shall return a character string representing the parameter `ID`. The string representation is implementation defined, but it shall be such that if two process group IDs are different, then their images are different.

`Value` translates the parameter `Str` into a `Process_Group_ID`. `Value` shall translate any string produced by `Image` into the `Process_Group_ID` value that was originally the input parameter to `Image`. It is implementation defined whether any other string representation may be successfully translated into a value of type `Process_Group_ID`.

4.1.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The value for the parameter `Process` or `Process_Group` for `Set_Process_Group_ID` or `Create_Process_Group` is a value not supported by this implementation.

`Permission_Denied`

The parameter `Process` of the procedure `Set_Process_Group_ID` or `Create_Process_Group` matches the process ID of a child process of the calling process, and the child process has successfully executed a call to `Exec`, `Exec_Search`, `Start_Process`, or `Start_Process_Search`.

`Operation_Not_Implemented`

`Set_Process_Group_ID` or `Create_Process_Group` was called, but it is not supported by this implementation.

`No_Such_Process`

The parameter `Process` of the procedure `Set_Process_Group_ID` or `Create_Process_Group` does not match the process ID of either the calling process or of a child process of the calling process.

`Operation_Not_Permitted`

(1) The process indicated by the parameter `Process` of `Set_Process_Group_ID` is a session leader or is a child process of the calling process, but that child process is not in the same session as the calling process.

(2) The process group indicated by the parameter `Process_Group` of the procedure `Set_Process_Group_ID` or `Create_Process_Group` does not match the process group ID of the indicated process, and no process with that process group ID is in the same session as the calling process.

(3) The process calling `Create_Session` is already a process group leader.

`Constraint_Error` shall be raised if the parameter `Str` to `Value` cannot be successfully translated to a `Process_Group_ID`.

No exceptions are specified by this standard for `Get_Process_Group_ID`.

No exception shall be raised by `Image`.

4.1.3 User Identification

4.1.3.1 Synopsis

```
type User_ID is private;
function Get_Real_User_ID return User_ID;
function Get_Effective_User_ID return User_ID;
procedure Set_User_ID (ID : in User_ID);
function Get_Login_Name return POSIX.POSIX_String;
function Image (ID : User_ID) return Standard.String;
function Value (Str : Standard.String) return User_ID;
```

4.1.3.2 Description

The type `User_ID` shall identify users of the system. The concepts of user ID, real user ID, effective user ID, and saved set-user-ID are defined in 2.2.2.

The `Get_Real_User_ID` function shall return the real user ID of the calling process.

The `Get_Effective_User_ID` function shall return the effective user ID of the calling process.

The effect of `Set_User_ID` depends on whether the Saved IDs option is supported.

If the Saved IDs option is supported, and the calling process has appropriate privileges, `Set_User_ID` shall set the real user ID, effective user ID, and saved set-user-ID to the value of the parameter `ID`.

If the Saved IDs option is supported, and the calling process does not have appropriate privileges, but the parameter `ID` is equal to the real user ID or the saved set-user-ID, `Set_User_ID` shall set the effective user ID to the value of the parameter `ID`. The real user ID and the saved set-user-ID shall remain unchanged.

If the Saved IDs option is not supported, and the process has appropriate privileges, `Set_User_ID` shall set the real user ID and the effective user ID to the value of the parameter `ID`.

If the Saved IDs option is not supported, and the process does not have appropriate privileges, but the parameter `ID` is equal to the real user ID, `Set_User_ID` shall change the effective user ID to the value of the parameter `ID`. The real user ID shall remain unchanged.

`Get_Login_Name` shall return a value of type `POSIX.POSIX_String` containing the name associated by the login activity with the controlling terminal for the calling process. If the login name of the user cannot be found, this function shall return a null `POSIX_String`.

`Image` shall return a character string representing the parameter `ID`. The string representation is implementation defined, but shall be such that, if two user IDs are different, then their images are different.

`Value` translates the parameter `Str` into a `User_ID`. `Value` shall translate any string produced by `Image` into the `User_ID` value that was originally the input parameter to `Image`. It is implementation defined whether any other string representation may be successfully translated into a `User_ID`.

4.1.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The parameter `ID` to `Set_User_ID` is not a valid user ID supported by the implementation.

`Operation_Not_Permitted`

The process calling `Set_User_ID` does not have appropriate privileges, the parameter `ID` does not equal the real user ID, and either the Saved IDs option is not supported or the parameter `ID` does not equal the saved set-user-ID.

`Constraint_Error` shall be raised if the parameter `Str` to `Value` cannot be interpreted as a user ID.

No exceptions are specified by this standard for `Get_Real_User_ID`, `Get_Effective_User_ID`, and `Get_Login_Name`.

No exception shall be raised by `Image`.

4.1.4 User and Group Identification

4.1.4.1 Synopsis

```
type Group_ID is private;
function Get_Real_Group_ID return Group_ID;
function Get_Effective_Group_ID return Group_ID;
procedure Set_Group_ID (ID : in Group_ID);
subtype Group_List_Index is positive range 1 .. POSIX.Groups_Maxima'Last;
type Group_List is array (Group_List_Index range <>) of Group_ID;
function Get_Groups return Group_List;
function Image (ID : Group_ID) return Standard.String;
function Value (Str : Standard.String) return Group_ID;
```

4.1.4.2 Description

The type `Group_ID` shall identify groups of the system. The concepts of (user) group ID, real group ID, effective group ID, and saved set-group-ID are defined in 2.2.2.

The type `Group_List` shall represent an arbitrary list of groups.

`Get_Real_Group_ID` shall return the real (user) group ID of the calling process.

`Get_Effective_Group_ID` shall return the effective group ID of the calling process.

`Get_Groups` shall return a list containing the supplementary group IDs of the calling process. This standard does not specify whether the effective group ID of the calling process is included in the returned list of supplementary group IDs.

`Image` shall return a character string representing the parameter `ID`. The string representation is implementation defined, but shall be such that if two group IDs are different, then their images are different.

`Value` translates the parameter `Str` into a `Group_ID`. `Value` shall translate any string produced by `Image` into the `Group_ID` value that was originally the input parameter to `Image`. It is implementation defined whether any other string representation may be successfully translated into a `Group_ID`.

The effect of `Set_Group_ID` depends on whether the Saved IDs option is supported.

If the Saved IDs option is supported, and the calling process has appropriate privileges, `Set_Group_ID` shall set the real group ID, effective group ID, and saved set-group-ID to the value of the parameter `ID`.

If the Saved IDs option is supported, and the calling process does not have appropriate privileges, but the parameter `ID` is equal to the real group ID or the saved set-group-ID, `Set_Group_ID` shall set the effective group ID to the parameter `ID`. The real group ID and the saved set-group-ID shall remain unchanged.

If the Saved IDs option is not supported, and the process has appropriate privileges, `Set_Group_ID` shall set the real group ID and the effective group ID to the value of the parameter `ID`.

If the Saved IDs option is not supported, and the process does not have appropriate privileges, but the parameter `ID` is equal to the real group ID, `Set_Group_ID` shall change the effective group ID to the value of the parameter `ID`. The real group ID shall remain unchanged.

4.1.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The parameter `ID` to `Set_Group_ID` is not a valid group ID supported by the implementation.

`Operation_Not_Permitted`

The process calling `Set_Group_ID` does not have appropriate privileges, the parameter `ID` does not equal the real group ID, and either the Saved IDs option is not supported or the parameter `ID` does not equal the saved set-group-ID.

`Constraint_Error` shall be raised if the parameter `Str` to `Value` cannot be interpreted as a group ID.

No exceptions are specified by this standard for `Get_Real_Group_ID`, `Get_Effective_Group_ID`, and `Get_Groups`.

No exception shall be raised by `Image`.

4.2 Package POSIX_Process_Times

This package provides operations to retrieve system time-accounting information.

```
with POSIX;
package POSIX_Process_Times is
   Ticks_Per_Second : constant := implementation-defined;
   -- 4.2.1 Process Time Accounting
   type Tick_Count is implementation-defined-integer;
   -- Minimally 0 to 24 hours
   function Elapsed_Real_Time return Tick_Count;
   type Process_Times is private;
   function Get_Process_Times return Process_Times;
   function Elapsed_Real_Time_Of
      (Times : Process_Times) return Tick_Count;
   function User_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
   function System_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
   function Descendants_User_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
   function Descendants_System_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
private
   implementation-defined
end POSIX_Process_Times;
```

4.2.1 Process Time Accounting

4.2.1.1 Synopsis

```
   type Tick_Count is implementation-defined-integer;
   -- Minimally 0 to 24 hours
   function Elapsed_Real_Time return Tick_Count;
   type Process_Times is private;
   function Get_Process_Times return Process_Times;
   function Elapsed_Real_Time_Of
      (Times : Process_Times) return Tick_Count;
   function User_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
   function System_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
   function Descendants_User_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
   function Descendants_System_CPU_Time_Of
      (Times : Process_Times) return Tick_Count;
```

4.2.1.2 Description

The type Tick_Count shall express a time duration as a count of the number of ticks of the system clock. The constant Ticks_Per_Second defines the number of ticks of the system clock that occur each second; if that number is not integral, this constant shall be the nearest integer approximation to that number.

The type Process_Times shall comprise real-time and system-accounting time information about a process.

`Elapsed_Real_Time` shall return the elapsed real time (sometimes informally called call clock time), in ticks, between the start of the program and the time of its call.

NOTE: Although the values returned by `Elapsed_Real_Time` and `Elapsed_Real_Time_Of` are called "real time", the values returned are not sufficiently accurate or reliable for use as a time base in realtime applications. Instead, see the realtime clock and timer facilities which defined in 14.1.

`Get_Process_Times` shall return a value of type `Process_Times` containing time information about the current process at the time of the call. This value may then be inspected with `Elapsed_Real_Time_Of`, `User_CPU_Time_Of`, `System_CPU_Time_Of`, `Descendants_User_CPU_Time_Of`, and `Descendants_System_CPU_Time_Of` to extract specific time values. All time values shall be of the type `Tick_Count`.

`Elapsed_Real_Time_Of` shall return the elapsed real time, in ticks, between the start of the program and the time of its call as recorded in the parameter `Times`.

`User_CPU_Time_Of` shall return the CPU time used by the process as recorded in the parameter `Times`.

`System_CPU_Time_Of` shall return the CPU time used by the system on behalf of the process as recorded in the parameter `Times`.

`Descendants_User_CPU_Time_Of` shall return the sum of the values that would be returned by `User_CPU_Time_Of (Times)` and `Descendants_User_CPU_Time_Of (Times)` for all terminated child processes for which `Wait_For_Child_Process` has returned the child `Process_ID`. If a child process has not waited for its terminated children, their times are not included in its `Descendants_User_CPU_Time_Of` and thus are not included in this sum.

`Descendants_System_CPU_Time_Of` shall return the sum of the values that would be returned by `System_CPU_Time_of (Times)` and `Descendants_System_CPU_Time_Of (Times)` for all terminated child processes for which `Wait_For_Child_Process` has returned the child `Process_ID`. If a child process has not waited for its terminated children, their times are not included in its `Descendants_System_CPU_Time_Of` and thus are not included in this sum.

4.2.1.3 Error Handling

No exceptions are specified by this standard for `Elapsed_Real_Time_Of`, `User_CPU_Time_Of`, `System_CPU_Time_Of`, `Descendants_User_CPU_Time_Of`, or `Descendants_System_CPU_Time_Of`.

No exceptions are specified by this standard for `Elapsed_Real_Time` or `Get_Process_Times`. Calling `Elapsed_Real_Time` or `Get_Process_Times` may raise `POSIX_Error` with implementation-defined error codes for implementation-defined conditions.

4.3 Package `POSIX_Process_Environment`

This package defines access to the environment that exists when a process is started.

Specific functions are provided to

— Determine the strings that are passed as command line arguments to a POSIX process, including the first argument, which is by convention the name of the command that initiated the process

— Get and set the current process environment, which provides global control of process options

— Determine whether an environment variable has been defined in an environment

— Get and modify the values of environment variables

— Iterate execution of a procedure for each of variables in an environment.

— Get and change the current working directory, which is used in resolving relative pathnames

```
with POSIX;
package POSIX_Process_Environment is
   -- 4.3.1 Argument List
   function Argument_List return POSIX.POSIX_String_List;
   -- 4.3.2 Environment Variables
   type Environment is limited private;
   procedure Copy_From_Current_Environment
      (Env : in out Environment);
   procedure Copy_To_Current_Environment
      (Env : in Environment);
   procedure Copy_Environment
      (Source : in Environment;
       Target : in out Environment);
   function Environment_Value_Of
      (Name :      POSIX.POSIX_String;
       Env :       Environment;
       Undefined : POSIX.POSIX_String := "")
     return POSIX.POSIX_String;
   function Environment_Value_Of
      (Name :      POSIX.POSIX_String;
       Undefined : POSIX.POSIX_String := "")
     return POSIX.POSIX_String;
   function Is_Environment_Variable
      (Name : POSIX.POSIX_String;
       Env :  Environment)
     return Boolean;
   function Is_Environment_Variable
      (Name : POSIX.POSIX_String)
     return Boolean;
   procedure Clear_Environment
      (Env : in out Environment);
   procedure Clear_Environment;
   procedure Set_Environment_Variable
      (Name :  in POSIX.POSIX_String;
       Value : in POSIX.POSIX_String;
       Env :   in out Environment);
   procedure Set_Environment_Variable
      (Name :  in POSIX.POSIX_String;
       Value : in POSIX.POSIX_String);
   procedure Delete_Environment_Variable
      (Name : in POSIX.POSIX_String;
       Env :  in out Environment);
```

```
   procedure Delete_Environment_Variable
      (Name : in POSIX.POSIX_String);
   function Length (Env : Environment) return Natural;
   function Length return Natural;
   generic
      with procedure Action
         (Name :  in POSIX.POSIX_String;
          Value : in POSIX.POSIX_String;
          Quit :  in out Boolean);
   procedure For_Every_Environment_Variable
      (Env : in Environment);
   generic
      with procedure Action
         (Name :  in POSIX.POSIX_String;
          Value : in POSIX.POSIX_String;
          Quit :  in out Boolean);
   procedure For_Every_Current_Environment_Variable;
   -- 4.3.3 Process Working Directory
   procedure Change_Working_Directory
      (Directory_Name : in POSIX.Pathname);
   function Get_Working_Directory return POSIX.Pathname;  private

   type Environment is implementation-defined;
end POSIX_Process_Environment;
```

4.3.1 Argument List

4.3.1.1 Synopsis

```
   function Argument_List return POSIX.POSIX_String_List;
```

4.3.1.2 Description

`Argument_List` shall return the argument list of the current process as a value of type `POSIX.POSIX_String_List`. If the current process was created by a `Start_Process`, `Start_Process_Search`, `Exec`, or `Exec_Search` operation, this shall be the value provided by the `Arg_List` parameter of that operation. If no arguments were provided to the process at startup, the argument list shall contain at most a single element. By convention, this element is the name of the program being executed.

4.3.1.3 Error Handling

No exceptions shall be raised by `Argument_List`.

4.3.2 Environment Variables

4.3.2.1 Synopsis

```
   type Environment is limited private;
   procedure Copy_From_Current_Environment
      (Env : in out Environment);
   procedure Copy_To_Current_Environment
      (Env : in Environment);
   procedure Copy_Environment
      (Source : in Environment;
```

```
    Target : in out Environment);
function Environment_Value_Of
   (Name :      POSIX.POSIX_String;
    Env :       Environment;
    Undefined : POSIX.POSIX_String := "")
  return POSIX.POSIX_String;
function Environment_Value_Of
   (Name :      POSIX.POSIX_String;
    Undefined : POSIX.POSIX_String := "")
  return POSIX.POSIX_String;
function Is_Environment_Variable
   (Name : POSIX.POSIX_String;
    Env :  Environment)
  return Boolean;
function Is_Environment_Variable
   (Name : POSIX.POSIX_String)
  return Boolean;
procedure Clear_Environment
   (Env : in out Environment);
procedure Clear_Environment;
procedure Set_Environment_Variable
   (Name :  in POSIX.POSIX_String;
    Value : in POSIX.POSIX_String;
    Env :   in out Environment);
procedure Set_Environment_Variable
   (Name :  in POSIX.POSIX_String;
    Value : in POSIX.POSIX_String);
procedure Delete_Environment_Variable
   (Name : in POSIX.POSIX_String;
    Env :  in out Environment);
procedure Delete_Environment_Variable
   (Name : in POSIX.POSIX_String);
function Length (Env : Environment) return Natural;
function Length return Natural;
generic
   with procedure Action
      (Name :  in POSIX.POSIX_String;
       Value : in POSIX.POSIX_String;
       Quit :  in out Boolean);
procedure For_Every_Environment_Variable
   (Env : in Environment);
generic
   with procedure Action
      (Name :  in POSIX.POSIX_String;
       Value : in POSIX.POSIX_String;
       Quit :  in out Boolean);
procedure For_Every_Current_Environment_Variable;
```

4.3.2.2 Description

The type `Environment` shall be used to represent sets of environment variables. Each environment variable is a (name, value) pair, where the components are both of type `POSIX.POSIX_String`. The name shall be a nonnull string and cannot contain the character '`=`' or the null character. The value may contain any valid `POSIX_Character` except the null character, but may include the character '`=`'. The case of characters in the environment variable name is significant; the case of characters in the environment variable value is retained.

Each process shall have a set of environment variables, called the current environment of the process, which is provided at the time the process is created and may be modified by the process. There is one current environment per process, and that environment is shared by all tasks in that process.

A POSIX implementation may (but need not) dynamically allocate storage for (name, value) pairs. The operations that may allocate such storage are `Copy_From_Current_Environment`, `Copy_To_Current_Environment`, `Copy_Environment`, and `Set_Environment_Variable`. If these operations do allocate storage dynamically, then these operations and the operations `Clear_Environment` and `Delete_Environment_Variable` shall recover any storage that may have been allocated to (name, value) pairs that have been deleted or values that have been changed.

A POSIX implementation may also (but need not) impose limits on the amount of memory available for storing (name, value) pairs in an object of type `Environment`. Any operation that would cause such a limit to be exceeded shall raise the exception `POSIX_Error` with error code `Argument_List_Too_Long`. If this error condition occurs for an environment, `Clear_Environment` and `Delete_Environment_Variable` shall continue to operate on the environment that caused the error and recover storage where possible.

This package provides several operations that read or modify a specified environment. These operations each have two (overloaded) forms. The first form has a formal parameter of type `Environment`, called `Env`, which specifies explicitly the environment to which the operation is applied. The second form has an implicit parameter, which is the current environment of the calling process. Modification operations with this second form operate on the (global) process environment. In the descriptions of operations in the following paragraphs, the phrase “specified environment” means the environment specified by the formal parameter or the current environment, according to whether there is an `Environment` parameter.

The operations provided by this package cannot produce an environment with multiple (name, value) pairs with identical names. Such environments are considered unspecified by this standard. However, it is possible for the environment supplied to the process at startup to contain such multiple-name definitions. The result of calling operations in this package with such an environment is undefined.

A null environment, one with no (name, value) pairs, is a valid environment.

An environment variable name may be undefined in an environment, defined with a null value, or defined with a nonnull value. Operations that access the environment can distinguish these three cases.

In a multitasking program, the effect of one task calling an operation that modifies an environment while another task is performing an operation on the same environment is undefined.

For iteration, distinct generic procedures are provided for the explicit and implicit iterator forms, since Ada overloading rules require distinct names.

`Copy_From_Current_Environment` shall copy into `Env` a value of type `Environment` containing the same set of (name, value) pairs as in the current environment

of the calling process. The implementation of this procedure shall recover all dynamically allocated storage that may have been allocated for the old value.

`Copy_To_Current_Environment` shall copy `Env` into the current environment of the calling process, deleting all previous (name, value) pairs for the current environment. The implementation of this procedure shall recover all dynamically allocated storage that may have been allocated for the old value.

`Copy_Environment` shall copy the value of the parameter `Source` into the parameter `Target`. All previous values in `Target` are deleted. The implementation of this procedure shall recover all dynamically allocated storage that may have been allocated for the old value.

`Environment_Value_Of` shall search the specified environment for a variable whose name matches the parameter `Name`. If such a variable is found, its value, possibly null, shall be returned as a value of type `POSIX.POSIX_String`. If no matching variable is found, the value provided by the parameter `Undefined` shall be returned.

`Is_Environment_Variable` shall return `True` if a variable with the name specified by the parameter `Name` is in the specified environment. Otherwise, `False` shall be returned.

`Clear_Environment` shall remove all variables from the specified environment. It shall also recover all memory that may have been allocated dynamically by the POSIX implementation for (name, value) pairs belonging to the specified environment.

`Set_Environment_Variable` shall add the (name, value) pair specified by `Name` and `Value` to the specified environment. The value of the parameter `Value` may be a null `POSIX.POSIX_String`. If the parameter `Name` is already the name of a variable in the specified environment, then the parameter `Value` shall become the new value associated with the variable denoted by `Name`. The implementation of this procedure shall recover all dynamically allocated storage that may have been allocated for the old value.

`Delete_Environment_Variable` shall delete the variable named by the parameter `Name` from the specified environment. If the name `Name` is not defined in the specified environment, the procedure returns normally without any effect. The implementation of this procedure shall recover all dynamically allocated storage that may have been allocated for the (name, value) pair.

`Length` shall return the number of (name, value) pairs defined in the specified environment.

The generic procedures `For_Every_Environment_Variable` and `For_Every_Current_Environment_Variable` shall be instantiated by the application with an actual procedure for the generic formal procedure `Action`. When called, the instance shall call the actual procedure supplied for `Action` once for each (name, value) pair in the specified environment.

`Action` shall be able to force termination of the generic instance either by setting `Quit` to `True` or by raising an exception. Prior to calling `Action`, the instance shall set `Quit` to `False`. The order in which the entries are presented is unspecified. The

behavior of the iterator is undefined if Action modifies the specified environment as a side effect.

Exceptions raised by Action shall terminate iteration and shall be propagated back to the caller of the instance. After an exception is raised by Action or Action returns with Quit set to True, no more calls to Action shall occur.

4.3.2.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The value for the parameter Name for the subprograms Environment_Value_Of, Is_Environment_Variable, Set_Environment_Variable, or Delete_Environment_Variable is a null string or contains the character '='. If this exception is raised, the specified environment shall be unchanged.

Argument_List_Too_Long

Set_Environment_Variable was called with an environment and variable value that would exceed system-imposed limits if the variable were added to the environment.

No exceptions are specified by this standard for Copy_From_Current_Environment, Copy_To_Current_Environment, Copy_Environment, Clear_Environment, Length, For_Every_Environment_Variable, and For_Every_Current_Environment_Variable.

4.3.3 Process Working Directory

4.3.3.1 Synopsis

```
procedure Change_Working_Directory
   (Directory_Name : in POSIX.Pathname);
function Get_Working_Directory return POSIX.Pathname;
```

4.3.3.2 Description

The process working directory is used for interpreting relative pathnames (pathnames that do not begin with a slash). The process working directory is the starting point for resolving a relative pathname, as described in 2.3.11.

There is one working directory per process; the value is shared by all tasks in that process.

Get_Working_Directory shall return the absolute pathname of the current working directory as a value of type POSIX.Pathname.

Change_Working_Directory shall change the process working directory to the value of the parameter Directory_Name. If an exception is raised by Change_Working_Directory, the current working directory shall be unchanged.

4.3.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

(1) The parameter `Directory_Name` to `Change_Working_Directory` contains a component directory for which read or search permission is denied.

(2) Read or search permission is denied for a component of the current process working directory that would be returned by `Get_Working_Directory`.

`Filename_Too_Long`

The length of the parameter `Directory_Name` to `Change_Working_Directory` exceeds the system-imposed limit for that pathname. Information on this limit is provided by Pathname Maximum (see 5.4.1).

`Not_A_Directory`

The parameter `Directory_Name` to `Change_Working_Directory` contains a component that is not a directory.

`No_Such_File_Or_Directory`

The parameter `Directory_Name` to `Change_Working_Directory` defines a directory that does not exist or is null.

4.4 Package POSIX_Calendar

This package provides an analog to the facilities defined in the Ada standard package `Calendar`. However, the time type defined in `POSIX_Calendar` includes sufficient information to permit values of this type to be interpreted using the **TZ** environment variable. (See 2.11.1.)

```
with POSIX,
     Calendar;
package POSIX_Calendar is
   -- 4.4.1 Time Information
   type POSIX_Time is private;
   function Clock return POSIX_Time;
   function To_Time (Date : POSIX_Time) return Calendar.Time;
   function To_POSIX_Time (Date : Calendar.Time) return POSIX_Time;
   function To_POSIX_Time (Date : POSIX.Timespec) return POSIX_Time;
   function To_Timespec (Date : POSIX_Time) return POSIX.Timespec;
   function "+" (L : POSIX_Time; R : Duration) return POSIX_Time;
   function "+" (L : Duration;   R : POSIX_Time) return POSIX_Time;
   function "-" (L : POSIX_Time; R : Duration) return POSIX_Time;
   function "-" (L : POSIX_Time; R : POSIX_Time) return Duration;
   function "<" (L, R : POSIX_Time) return Boolean;
   function "<="(L, R : POSIX_Time) return Boolean;
   function ">" (L, R : POSIX_Time) return Boolean;
   function ">="(L, R : POSIX_Time) return Boolean;
   -- 4.4.2 Operations on POSIX Times
   subtype Year_Number    is Calendar.Year_Number;
```

```
   subtype Month_Number   is Calendar.Month_Number;
   subtype Day_Number     is Calendar.Day_Number;
   subtype Day_Duration   is Calendar.Day_Duration;
   function Year  (Date : POSIX_Time) return Year_Number;
   function Month (Date : POSIX_Time) return Month_Number;
   function Day   (Date : POSIX_Time) return Day_Number;
   function Seconds (Date : POSIX_Time) return Day_Duration;
   procedure Split
      (Date :    in  POSIX_Time;
       Year :    out Year_Number;
       Month :   out Month_Number;
       Day :     out Day_Number;
       Seconds : out Day_Duration);
   function Time_Of
      (Year :    Year_Number;
       Month :   Month_Number;
       Day :     Day_Number;
       Seconds : Day_Duration := 0.0)
      return POSIX_Time;
   Time_Error : exception renames Calendar.Time_Error;
private
   implementation-defined;
end POSIX_Calendar;
```

4.4.1 Time Information

4.4.1.1 Synopsis

```
   type POSIX_Time is private;
   function Clock return POSIX_Time;
   function To_Time (Date : POSIX_Time) return Calendar.Time;
   function To_POSIX_Time (Date : Calendar.Time) return POSIX_Time;
   function To_POSIX_Time (Date : POSIX.Timespec) return POSIX_Time;
   function To_Timespec (Date : POSIX_Time) return POSIX.Timespec;
   function "+" (L : POSIX_Time; R : Duration) return POSIX_Time;
   function "+" (L : Duration;   R : POSIX_Time) return POSIX_Time;
   function "-" (L : POSIX_Time; R : Duration) return POSIX_Time;
   function "-" (L : POSIX_Time; R : POSIX_Time) return Duration;
   function "<" (L, R : POSIX_Time) return Boolean;
   function "<="(L, R : POSIX_Time) return Boolean;
   function ">" (L, R : POSIX_Time) return Boolean;
   function ">="(L, R : POSIX_Time) return Boolean;
```

4.4.1.2 Description

The type POSIX_Time is used to obtain times and dates from the system. Values of this type shall be independent of the current time zone in effect for the system, so that an arbitrary value of this type may be interpreted at a later date using information in the **TZ** environment variable. (See 2.11.1.)

The function Clock shall return a value of type POSIX_Time reflecting the current system time.

The functions named To_POSIX_Time shall convert a value of type POSIX.Timespec or Calendar.Time to a value of type POSIX_Time.

The function To_Timespec shall convert a value of type POSIX_Time to a value of type POSIX.Timespec.

The function To_Time shall convert a value of type POSIX_Time to a value of type Calendar.Time.

When converting a value of the type POSIX.Timespec, the value shall be interpreted as an offset from the Epoch. (See 2.2.2.58.) If a value of Calendar.Time does not contain sufficient information to interpret its value in a time-zone-independent fashion, the value shall be interpreted using the current value of the **TZ** environment variable.

The operators "+" and "-" provide for arithmetic on values of POSIX_Time and the Ada standard type Duration. The operators "<", "<=", ">" and ">=" provide relational operations on values of type POSIX_Time. The arithmetic and relational operators have their conventional meanings.

4.4.1.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The value of the environment variable **TZ** is not recognized by this implementation as specifying valid time-zone information.

Time_Error may be raised by the operators "+" or "-" if the operations cannot return a valid value of type POSIX_Time whose year number is in the range of the subtype Year_Number or whose seconds value is not in the range of the type Duration.

Constraint_Error may be raised by To_Time, To_POSIX_Time, or To_Timespec for implementation-defined reasons, including the situation where a value of the source type cannot be converted to a value of the target type.

4.4.2 Operations on POSIX Times

4.4.2.1 Synopsis

```
subtype Year_Number    is Calendar.Year_Number;
subtype Month_Number   is Calendar.Month_Number;
subtype Day_Number     is Calendar.Day_Number;
subtype Day_Duration   is Calendar.Day_Duration;
function Year  (Date : POSIX_Time) return Year_Number;
function Month (Date : POSIX_Time) return Month_Number;
function Day   (Date : POSIX_Time) return Day_Number;
function Seconds (Date : POSIX_Time) return Day_Duration;
procedure Split
   (Date :    in  POSIX_Time;
    Year :    out Year_Number;
    Month :   out Month_Number;
    Day :     out Day_Number;
    Seconds : out Day_Duration);
function Time_Of
   (Year :    Year_Number;
```

```
      Month :   Month_Number;
      Day :     Day_Number;
      Seconds : Day_Duration := 0.0)
     return POSIX_Time;
  Time_Error : exception renames Calendar.Time_Error;
```

4.4.2.2 Description

The procedure `Split` shall take a value of type `POSIX_Time` and shall interpret it according to the current value of the **TZ** environment variable to provide time-related information. (See 2.11.1.) The values returned by `Split` shall indicate the `Year`, `Month`, `Day`, and `Seconds` represented by the `POSIX_Time` value. The `Day_Duration` value in the parameter `Seconds` shall indicate the total number of seconds in the indicated day. For instance, a value of `Seconds` of "3600.0" shall mean 1:00 AM on the given day.

The function `Year` shall return the year number corresponding to the date indicated by the value of type `POSIX_Time`, interpreted using the current value of the **TZ** environment variable. The function `Month` shall return the month number corresponding to the date indicated by the value of type `POSIX_Time`, interpreted using the current value of the **TZ** environment variable. The function `Day` shall return the day number corresponding to the date indicated by the value of type `POSIX_Time`, interpreted using the current value of the **TZ** environment variable. The function `Seconds` shall return the number of seconds in the day corresponding to the date indicated by the value of type `POSIX_Time`, interpreted using the current value of the **TZ** environment variable.

The function `Time_Of` shall return a value of type `POSIX_Time`, interpreted in the time zone indicated by the **TZ** environment variable.

If the value of **TZ** is null, then values of `POSIX_Time` shall be interpreted using default system time-zone information.

4.4.2.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The value of the environment variable **TZ** is not recognized by this implementation as specifying valid time-zone information.

`Time_Error` shall be raised by `Time_Of` if the parameters cannot be interpreted to form a valid date.

4.5 Package `POSIX_Configurable_System_Limits`

This package provides operations to retrieve the configurable system limits, options, or versions at runtime.

```
with POSIX,
     POSIX_Limits,
     POSIX_Options;
package POSIX_Configurable_System_Limits is
   -- 4.5.1 Get Configurable System Options
   function Asynchronous_IO_Is_Supported
     return POSIX_Options.Asynchronous_IO_Support;
   function File_Synchronization_Is_Supported
     return POSIX_Options.File_Synchronization_Support;
   function Internet_Datagram_Is_Supported
     return POSIX_Options.Internet_Datagram_Support;
   function Internet_Protocol_Is_Supported
     return POSIX_Options.Internet_Protocol_Support;
   function Internet_Stream_Is_Supported
     return POSIX_Options.Internet_Stream_Support;
   function ISO_OSI_Protocol_Is_Supported
     return POSIX_Options.ISO_OSI_Protocol_Support;                    c
   function Job_Control_Supported               -- obsolescent
     return POSIX.Job_Control_Support;          -- obsolescent
   function Job_Control_Is_Supported
     return POSIX_Options.Job_Control_Support
     renames Job_Control_Supported;
   function Memory_Mapped_Files_Are_Supported
     return POSIX_Options.Memory_Mapped_Files_Support;
   function Memory_Locking_Is_Supported
     return POSIX_Options.Memory_Locking_Support;
   function Memory_Range_Locking_Is_Supported
     return POSIX_Options.Memory_Range_Locking_Support;
   function Memory_Protection_Is_Supported
     return POSIX_Options.Memory_Protection_Support;
   function Message_Queues_Are_Supported
     return POSIX_Options.Message_Queues_Support;
   function Mutex_Priority_Ceiling_Is_Supported
     return POSIX_Options.Mutex_Priority_Ceiling_Support;
   function Mutex_Priority_Inheritance_Is_Supported
     return POSIX_Options.Mutex_Priority_Inheritance_Support;
   function Mutexes_Are_Supported
     return POSIX_Options.Mutexes_Support;
   function Network_Management_Is_Supported
     return POSIX_Options.Network_Management_Support;
   function OSI_Connectionless_Is_Supported
     return POSIX_Options.OSI_Connectionless_Support;
   function OSI_Connection_Is_Supported
     return POSIX_Options.OSI_Connection_Support;
   function OSI_Minimal_Is_Supported
     return POSIX_Options.OSI_Minimal_Support;
   function Poll_Is_Supported
     return POSIX_Options.Poll_Support;                                c
   function Prioritized_IO_Is_Supported
     return POSIX_Options.Prioritized_IO_Support;
   function Process_Shared_Is_Supported
     return POSIX_Options.Process_Shared_Support;
   function Priority_Process_Scheduling_Is_Supported
     return POSIX_Options.Priority_Process_Scheduling_Support;
   function Priority_Task_Scheduling_Is_Supported
     return POSIX_Options.Priority_Task_Scheduling_Support;
   function Realtime_Signals_Are_Supported
     return POSIX_Options.Realtime_Signals_Support;
```

```
function Saved_IDs_Supported                    -- obsolescent
  return POSIX.Saved_IDs_Support;               -- obsolescent
function Saved_IDs_Are_Supported
  return POSIX_Options.Saved_IDs_Support
  renames Saved_IDs_Supported;
function Select_Is_Supported                                                    |
  return POSIX_Options.Select_Support;                                          |c
function Semaphores_Are_Supported
  return POSIX_Options.Semaphores_Support;
function Shared_Memory_Objects_Are_Supported
  return POSIX_Options.Shared_Memory_Objects_Support;
function Sockets_DNI_Is_Supported                                               |
  return POSIX_Options.Sockets_DNI_Support;                                     |c
function Synchronized_IO_Is_Supported
  return POSIX_Options.Synchronized_IO_Support;
function Timers_Are_Supported
  return POSIX_Options.Timers_Support;
function XTI_DNI_Is_Supported                                                   |
  return POSIX_Options.XTI_DNI_Support;                                         |c
-- 4.5.2 Get Configurable System Limits
type POSIX_Version is implementation-defined-integer;
function System_POSIX_Version return POSIX_Version;
function System_POSIX_Ada_Version return POSIX_Version;
function Argument_List_Maximum
  return POSIX_Limits.Argument_List_Maxima;
function Asynchronous_IO_Maximum
  return POSIX_Limits.Asynchronous_IO_Maxima;
function Asynchronous_IO_Priority_Delta_Maximum
  return POSIX_Limits.Asynchronous_IO_Priority_Delta_Maxima;
function Child_Processes_Maximum
  return POSIX_Limits.Child_Processes_Maxima;
function Groups_Maximum
  return POSIX_Limits.Groups_Maxima;
function List_IO_Maximum
  return POSIX_Limits.List_IO_Maxima;
function Open_Message_Queues_Maximum
  return POSIX_Limits.Open_Message_Queues_Maxima;
function Message_Priority_Maximum
  return POSIX_Limits.Message_Priority_Maxima;
function Open_Files_Maximum
  return POSIX_Limits.Open_Files_Maxima;
function Page_Size
  return POSIX_Limits.Page_Size_Range;
function Queued_Signals_Maximum
  return POSIX_Limits.Queued_Signals_Maxima;
function Realtime_Signals_Maximum
  return POSIX_Limits.Realtime_Signals_Maxima;
function Semaphores_Maximum
  return POSIX_Limits.Semaphores_Maxima;
function Semaphores_Value_Maximum
  return POSIX_Limits.Semaphores_Value_Maxima;
function Socket_IO_Vector_Maximum                                               |
  return POSIX_Limits.Socket_IO_Vector_Maxima;                                  |c
function Stream_Maximum         -- obsolescent
  return POSIX.Stream_Maxima;           -- obsolescent
function Streams_Maximum
  return POSIX_Limits.Streams_Maxima
  renames Stream_Maximum;
```

```
   function Timers_Maximum
     return POSIX_Limits.Timers_Maxima;
   function Timer_Overruns_Maximum
     return POSIX_Limits.Timer_Overruns_Maxima;
   function Time_Zone_String_Maximum
     return POSIX_Limits.Time_Zone_String_Maxima;
   function XTI_IO_Vector_Maximum
     return POSIX_Limits.XTI_IO_Vector_Maxima;                    c

end POSIX_Configurable_System_Limits;
```

4.5.1 Get Configurable System Options

4.5.1.1 Synopsis

```
   function Asynchronous_IO_Is_Supported
     return POSIX_Options.Asynchronous_IO_Support;
   function File_Synchronization_Is_Supported
     return POSIX_Options.File_Synchronization_Support;
   function Internet_Datagram_Is_Supported
     return POSIX_Options.Internet_Datagram_Support;
   function Internet_Protocol_Is_Supported
     return POSIX_Options.Internet_Protocol_Support;
   function Internet_Stream_Is_Supported
     return POSIX_Options.Internet_Stream_Support;
   function ISO_OSI_Protocol_Is_Supported
     return POSIX_Options.ISO_OSI_Protocol_Support;               c
   function Job_Control_Supported               -- obsolescent
     return POSIX.Job_Control_Support;          -- obsolescent
   function Job_Control_Is_Supported
     return POSIX_Options.Job_Control_Support
     renames Job_Control_Supported;
   function Memory_Mapped_Files_Are_Supported
     return POSIX_Options.Memory_Mapped_Files_Support;
   function Memory_Locking_Is_Supported
     return POSIX_Options.Memory_Locking_Support;
   function Memory_Range_Locking_Is_Supported
     return POSIX_Options.Memory_Range_Locking_Support;
   function Memory_Protection_Is_Supported
     return POSIX_Options.Memory_Protection_Support;
   function Message_Queues_Are_Supported
     return POSIX_Options.Message_Queues_Support;
   function Mutex_Priority_Ceiling_Is_Supported
     return POSIX_Options.Mutex_Priority_Ceiling_Support;
   function Mutex_Priority_Inheritance_Is_Supported
     return POSIX_Options.Mutex_Priority_Inheritance_Support;
   function Mutexes_Are_Supported
     return POSIX_Options.Mutexes_Support;
   function Network_Management_Is_Supported
     return POSIX_Options.Network_Management_Support;
   function OSI_Connectionless_Is_Supported
     return POSIX_Options.OSI_Connectionless_Support;
   function OSI_Connection_Is_Supported
     return POSIX_Options.OSI_Connection_Support;
   function OSI_Minimal_Is_Supported
     return POSIX_Options.OSI_Minimal_Support;
   function Poll_Is_Supported
     return POSIX_Options.Poll_Support;                           c
   function Prioritized_IO_Is_Supported
```

```
    return POSIX_Options.Prioritized_IO_Support;
  function Process_Shared_Is_Supported
    return POSIX_Options.Process_Shared_Support;
  function Priority_Process_Scheduling_Is_Supported
    return POSIX_Options.Priority_Process_Scheduling_Support;
  function Priority_Task_Scheduling_Is_Supported
    return POSIX_Options.Priority_Task_Scheduling_Support;
  function Realtime_Signals_Are_Supported
    return POSIX_Options.Realtime_Signals_Support;
  function Saved_IDs_Supported                  -- obsolescent
    return POSIX.Saved_IDs_Support;             -- obsolescent
  function Saved_IDs_Are_Supported
    return POSIX_Options.Saved_IDs_Support
    renames Saved_IDs_Supported;
  function Select_Is_Supported                                          |c
    return POSIX_Options.Select_Support;
  function Semaphores_Are_Supported
    return POSIX_Options.Semaphores_Support;
  function Shared_Memory_Objects_Are_Supported
    return POSIX_Options.Shared_Memory_Objects_Support;
  function Sockets_DNI_Is_Supported                                     |c
    return POSIX_Options.Sockets_DNI_Support;
  function Synchronized_IO_Is_Supported
    return POSIX_Options.Synchronized_IO_Support;
  function Timers_Are_Supported
    return POSIX_Options.Timers_Support;
  function XTI_DNI_Is_Supported                                         |c
    return POSIX_Options.XTI_DNI_Support;
```

4.5.1.2 Description

These functions can be used to discover the support provided for certain options (see 2.5) by a particular implementation. For system-wide `Boolean` valued options, the function shall return a `True` if the corresponding option is supported, and `False` if the option is not supported. For pathname-specific `Boolean` valued options, the function shall return `True` if the corresponding option is supported on all or some files, and `False` if it is not supported on any files. For multivalued options, the function shall return the value that indicates the support for the option. The correspondence of functions to options (see 2.5) shall be as shown in Table 4.1.

NOTE: For pathname-specific options, the application can determine reliably that an option is supported on a specific file only by calling the corresponding pathname-specific function, declared in the package `POSIX_Configurable_File_Limits` and described in 5.4.

NOTE: Alternate names are provided for `Job_Control_Is_Supported` and `Saved_IDs_Are_Supported`, for compatibility with the names of these operations in POSIX.5.

4.5.1.3 Error Handling

None of these functions shall raise any exception.

Table 4.1 – Functions for System-Wide Options

Function	Option	
Asynchronous_IO_Is_Supported	Asynchronous I/O	
File_Synchronization_Is_Supported	File Synchronization	
Internet_Datagram_Is_Supported	Internet Datagram	
Internet_Protocol_Is_Supported	Internet Protocol	
Internet_Stream_Is_Supported	Internet Stream	
ISO_OSI_Protocol_Is_Supported	ISO/OSI Protocol	c
Job_Control_Is_Supported	Job Control	
Memory_Mapped_Files_Are_Supported	Memory Mapped Files	
Memory_Locking_Is_Supported	Memory Locking	
Memory_Range_Locking_Is_Supported	Memory Range Locking	
Memory_Protection_Is_Supported	Memory Protection	
Message_Queues_Are_Supported	Message Queues	
Mutex_Priority_Ceiling_Is_Supported	Mutex Priority Ceiling	
Mutex_Priority_Inheritance_Is_Supported	Mutex Priority Inheritance	
Mutexes_Are_Supported	Mutexes	
Network_Management_Is_Supported	Network Management	
OSI_Connectionless_Is_Supported	OSI Connectionless	
OSI_Connection_Is_Supported	OSI Connection	
OSI_Minimal_Is_Supported	OSI Minimal [for XTI only]	
Poll_Is_Supported	Poll	c
Prioritized_IO_Is_Supported	Prioritized I/O	
Priority_Process_Scheduling_Is_Supported	Priority Process Scheduling	
Priority_Task_Scheduling_Is_Supported	Priority Task Scheduling	
Process_Shared_Is_Supported	Process Shared	
Realtime_Signals_Are_Supported	Realtime Signals	
Saved_IDs_Are_Supported	Saved IDs	
Select_Is_Supported	Select	c
Semaphores_Are_Supported	Semaphores	
Shared_Memory_Objects_Are_Supported	Shared Memory Objects	
Sockets_DNI_Is_Supported	Sockets DNI	c
Synchronized_IO_Is_Supported	Synchronized I/O	
Timers_Are_Supported	Timers	
XTI_DNI_Is_Supported	XTI DNI	c

4.5.2 Get Configurable System Limits

4.5.2.1 Synopsis

```
type POSIX_Version is implementation-defined-integer;
function System_POSIX_Version return POSIX_Version;
function System_POSIX_Ada_Version return POSIX_Version;
function Argument_List_Maximum
  return POSIX_Limits.Argument_List_Maxima;
function Asynchronous_IO_Maximum
  return POSIX_Limits.Asynchronous_IO_Maxima;
function Asynchronous_IO_Priority_Delta_Maximum
  return POSIX_Limits.Asynchronous_IO_Priority_Delta_Maxima;
function Child_Processes_Maximum
  return POSIX_Limits.Child_Processes_Maxima;
```

```
function Groups_Maximum
  return POSIX_Limits.Groups_Maxima;
function List_IO_Maximum
  return POSIX_Limits.List_IO_Maxima;
function Open_Message_Queues_Maximum
  return POSIX_Limits.Open_Message_Queues_Maxima;
function Message_Priority_Maximum
  return POSIX_Limits.Message_Priority_Maxima;
function Open_Files_Maximum
  return POSIX_Limits.Open_Files_Maxima;
function Page_Size
  return POSIX_Limits.Page_Size_Range;
function Queued_Signals_Maximum
  return POSIX_Limits.Queued_Signals_Maxima;
function Realtime_Signals_Maximum
  return POSIX_Limits.Realtime_Signals_Maxima;
function Semaphores_Maximum
  return POSIX_Limits.Semaphores_Maxima;
function Semaphores_Value_Maximum
  return POSIX_Limits.Semaphores_Value_Maxima;
function Socket_IO_Vector_Maximum
  return POSIX_Limits.Socket_IO_Vector_Maxima;                    c
function Stream_Maximum        -- obsolescent
  return POSIX.Stream_Maxima;          -- obsolescent
function Streams_Maximum
  return POSIX_Limits.Streams_Maxima
  renames Stream_Maximum;
function Timers_Maximum
  return POSIX_Limits.Timers_Maxima;
function Timer_Overruns_Maximum
  return POSIX_Limits.Timer_Overruns_Maxima;
function Time_Zone_String_Maximum
  return POSIX_Limits.Time_Zone_String_Maxima;
function XTI_IO_Vector_Maximum
  return POSIX_Limits.XTI_IO_Vector_Maxima;                       c
```

4.5.2.2 Description

These functions can be used to obtain information about capacity limits (see 2.6) and other configurable values for a particular implementation of this standard. Each function shall return the actual system-wide value of the corresponding limit or other variable shown in Table 4.2.

`System_POSIX_Version` shall return a value describing the implemented version or revision of POSIX.1. The preferred interpretation of this integer is six decimal digits representing when the version of the standard was approved by the IEEE Standards Board; the year is the first four digits, and the month is the last two digits. This interpretation is not required by this standard.

`System_POSIX_Ada_Version` shall return a value describing the implemented version or revision of this standard. The preferred interpretation of this integer is six decimal digits representing when the version of the standard was approved by the IEEE Standards Board; the year is the first four digits, and the month is the last two digits. This interpretation is not required by this standard.

4.5.2.3 Error Handling

None of these functions shall raise any exception.

Table 4.2 – Configurable System Limits

Function	Limit	
Argument_List_Maximum	Argument List Maximum	
Asynchronous_IO_Maximum	Asynchronous I/O Maximum	
Asynchronous_IO_Priority_Delta_Maximum	Asynchronous I/O Priority Delta Maximum	
Child_Processes_Maximum	Child Processes Maximum	
Groups_Maximum	Groups Maximum	
List_IO_Maximum	List I/O Maximum	
Message_Priority_Maximum	Message Priority Maximum	
Open_Files_Maximum	Open Files Maximum	
Open_Message_Queues_Maximum	Open Message Queues Maximum	
Page_Size	Page Size	
Queued_Signals_Maximum	Queued Signals Maximum	
Realtime_Signals_Maximum	Realtime Signals Maximum	
Semaphores_Maximum	Semaphores Maximum	
Semaphores_Value_Maximum	Semaphores Value Maximum	
Socket_IO_Vector_Maximum	Socket IO Vector Maximum	c
Streams_Maximum	Streams Maximum	
Timers_Maximum	Timers Maximum	
Timer_Overruns_Maximum	Timer Overruns Maximum	
Time_Zone_String_Maximum	Time Zone String Maximum	
XTI_IO_Vector_Maximum	XTI IO Vector Maximum	c

Section 5: Files and Directories

This section defines POSIX operations on files and various properties of files as defined in the four packages POSIX_Permissions, POSIX_Files, POSIX_File_Status, and POSIX_Configurable_File_Limits. It does not define the I/O operations, which are defined in the packages POSIX_IO and POSIX_File_Locking in Section 6.

5.1 Package POSIX_Permissions

POSIX file protection is defined in terms of file permissions. The list of permissions is provided by the Permission enumeration and the constants making use of it. The permissions for a file are established when it is created and may be changed with POSIX_Files.Change_Permissions. They are then analyzed during program execution to determine whether the desired access can be allowed.

```
package POSIX_Permissions is
   -- 5.1.1 File Permissions
   type Permission is
      (Others_Execute, Others_Write, Others_Read,
       Group_Execute,  Group_Write,  Group_Read,
       Owner_Execute,  Owner_Write,  Owner_Read,
       Set_Group_ID,   Set_User_ID);
   type Permission_Set is array (Permission) of Boolean;
   Owner_Permission_Set : constant Permission_Set := Permission_Set'
      (Owner_Read | Owner_Write | Owner_Execute => true,
       others => false);
   Group_Permission_Set : constant Permission_Set := Permission_Set'
      (Group_Read | Group_Write | Group_Execute => true,
       others => false);
   Others_Permission_Set : constant Permission_Set := Permission_Set'
      (Others_Read | Others_Write | Others_Execute => true,
       others => false);
   Access_Permission_Set : constant Permission_Set := Permission_Set'
      (Owner_Read  | Owner_Write  | Owner_Execute => true,
       Group_Read  | Group_Write  | Group_Execute => true,
       Others_Read | Others_Write | Others_Execute => true,
       others => false);
   Set_Group_ID_Set : constant Permission_Set := Permission_Set'
      (Set_Group_ID => true,
       others => false);
   Set_User_ID_Set : constant Permission_Set := Permission_Set'
      (Set_User_ID => true,
       others => false);
   -- 5.1.2 Process Permission Set
   function Get_Allowed_Process_Permissions return Permission_Set;
   procedure Set_Allowed_Process_Permissions
      (Permissions : in  Permission_Set);
   procedure Set_Allowed_Process_Permissions
      (Permissions : in  Permission_Set;
       Old_Perms :   out Permission_Set);

end POSIX_Permissions;
```

5.1.1 File Permissions

5.1.1.1 Synopsis

```
type Permission is
   (Others_Execute, Others_Write, Others_Read,
    Group_Execute,  Group_Write,  Group_Read,
    Owner_Execute,  Owner_Write,  Owner_Read,
    Set_Group_ID,   Set_User_ID);
type Permission_Set is array (Permission) of Boolean;
Owner_Permission_Set : constant Permission_Set := Permission_Set'
   (Owner_Read | Owner_Write | Owner_Execute => true,
    others => false);
Group_Permission_Set : constant Permission_Set := Permission_Set'
   (Group_Read | Group_Write | Group_Execute => true,
    others => false);
Others_Permission_Set : constant Permission_Set := Permission_Set'
   (Others_Read | Others_Write | Others_Execute => true,
    others => false);
Access_Permission_Set : constant Permission_Set := Permission_Set'
   (Owner_Read  | Owner_Write  | Owner_Execute => true,
    Group_Read  | Group_Write  | Group_Execute => true,
    Others_Read | Others_Write | Others_Execute => true,
    others => false);
Set_Group_ID_Set : constant Permission_Set := Permission_Set'
   (Set_Group_ID => true,
    others => false);
Set_User_ID_Set : constant Permission_Set := Permission_Set'
   (Set_User_ID => true,
    others => false);
```

5.1.1.2 Description

The enumeration type `Permission` shall list the standard POSIX file modes. Sets of `Permission`s are represented by objects of the type `Permission_Set`. When assigned to a file, permissions in the `Access_Permission_Set` control access to the file. Permissions in `Set_User_ID_Set` and `Set_Group_ID_Set` control which effective group and user IDs are assigned when the file is executed. The meaning of each of these sets and the individual `Permission`s are as follows. In each case, a value of `True` in the set indicates that the corresponding permission is granted.

`Owner_Permission_Set`

Read, write, search (if a directory), or execute (otherwise) permission for the file owner class.

`Owner_Read`

Read permission for the file owner class.

`Owner_Write`

Write permission for the file owner class.

`Owner_Execute`

Search (if a directory) or execute (otherwise) permission for the file owner class.

`Group_Permission_Set`

Read, write, search (if a directory), or execute (otherwise) permission for the file group class.

Group_Read
: Read permission for the file group class.

Group_Write
: Write permission for the file group class.

Group_Execute
: Search (if a directory) or execute (otherwise) permission for the file group class.

Others_Permission_Set
: Read, write, search (if a directory), or execute (otherwise) permission for the file other class.

  Others_Read
  : Read permission for the file other class.

  Others_Write
  : Write permission for the file other class.

  Others_Execute
  : Search (if a directory) or execute (otherwise) permission for the file other class.

Set_Group_ID_Set
: Contains only the single element Set_Group_ID. The effective group ID of the process shall be set to that of the group of the file when the file is run as a program. On a regular file, Set_Group_ID shall be cleared on any write.

Set_User_ID_Set
: Contains only the single element Set_User_ID. The effective user ID of the process shall be set to that of the owner of the file when the file is run as a program. On a regular file, Set_User_ID shall be cleared on any write.

5.1.2 Process Permission Set

5.1.2.1 Synopsis

```
function Get_Allowed_Process_Permissions return Permission_Set;
procedure Set_Allowed_Process_Permissions
   (Permissions : in  Permission_Set);
procedure Set_Allowed_Process_Permissions
   (Permissions : in  Permission_Set;
    Old_Perms :   out Permission_Set);
```

5.1.2.2 Description

Each process shall have a set of allowed process permissions associated with it. The process permission set shall contain only Permissions that represent file access permissions, that is, those Permissions contained in the Access_Permission_Set. It does not contain either Set_Group_ID or Set_User_ID.

The set is implicitly used whenever the process creates a new file (operations Create_Directory, Create_FIFO, and POSIX_IO.Open_Or_Create; see 5.2.1 and 6.1.1). The access permissions established for the newly created file shall be those for which the permission is True in both the creation operation and the allowed process permission set.

`Get_Allowed_Process_Permissions` shall return the current allowed process permissions set.

`Set_Allowed_Process_Permissions` shall set the allowed process permissions set of the process to `Permissions`. One form of `Set_Allowed_Process_Permissions` permits the user to obtain the previous values of the permissions in the parameter `Old_Perms`. If any `Permissions` other than those in `Access_Permission_Set` are present in `Permissions`, their meaning is implementation defined.

5.1.2.3 Error Handling

No exceptions shall be raised by `Get_Allowed_Process_Permissions` or `Set_Allowed_Process_Permissions`.

5.2 Package POSIX_Files

This package provides operations for creating and removing files and directories and modifying their characteristics.

```
with POSIX,
     POSIX_Permissions,
     POSIX_Process_Identification,
     POSIX_Calendar;
package POSIX_Files is
   -- 5.2.1 Create and Remove Files
   procedure Create_Directory
      (Pathname :   in POSIX.Pathname;
       Permission : in POSIX_Permissions.Permission_Set);
   procedure Create_FIFO
      (Pathname :   in POSIX.Pathname;
       Permission : in POSIX_Permissions.Permission_Set);
   procedure Unlink (Pathname : in POSIX.Pathname);
   procedure Remove_Directory (Pathname : in POSIX.Pathname);
   -- 5.2.2 Inquiries on File Types
   function Is_File (Pathname : POSIX.Pathname) return Boolean;
   function Is_Directory (Pathname : POSIX.Pathname) return Boolean;
   function Is_FIFO (Pathname : POSIX.Pathname) return Boolean;
   function Is_Character_Special_File
      (Pathname : POSIX.Pathname) return Boolean;
   function Is_Block_Special_File
      (Pathname : POSIX.Pathname) return Boolean;
   function Is_Socket (Pathname : POSIX.Pathname)
     return Boolean;
   -- 5.2.3 Modify File Pathnames
   procedure Link
      (Old_Pathname : in POSIX.Pathname;
       New_Pathname : in POSIX.Pathname);
   procedure Rename
      (Old_Pathname : in POSIX.Pathname;
       New_Pathname : in POSIX.Pathname);
   -- 5.2.4 Directory Iteration
   type Directory_Entry is limited private;
   function Filename_Of (D_Entry : Directory_Entry) return POSIX.Filename;
```

```
   generic
       with procedure Action
          (D_Entry : in Directory_Entry;
           Quit :    in out Boolean);
   procedure For_Every_Directory_Entry
      (Pathname : in POSIX.Pathname);
   -- 5.2.5 Update File Status Information
   procedure Change_Owner_And_Group
      (Pathname : in POSIX.Pathname;
       Owner :    in POSIX_Process_Identification.User_ID;
       Group :    in POSIX_Process_Identification.Group_ID);
   procedure Change_Permissions
      (Pathname :   in POSIX.Pathname;
       Permission : in POSIX_Permissions.Permission_Set);
   procedure Set_File_Times
      (Pathname :          in POSIX.Pathname;
       Access_Time :       in POSIX_Calendar.POSIX_Time;
       Modification_Time : in POSIX_Calendar.POSIX_Time);
   procedure Set_File_Times (Pathname : in POSIX.Pathname);
   -- 5.2.6 Check File Accessibility
   type Access_Mode is (Read_Ok, Write_Ok, Execute_Ok);
   type Access_Mode_Set is array (Access_Mode) of Boolean;
   function Is_Accessible
      (Pathname :     POSIX.Pathname;
       Access_Modes : Access_Mode_Set)
     return Boolean;
   function Accessibility
      (Pathname :     POSIX.Pathname;
       Access_Modes : Access_Mode_Set)
     return POSIX.Error_Code;
   function Is_File_Present (Pathname : POSIX.Pathname) return Boolean;
   function Existence (Pathname : POSIX.Pathname) return POSIX.Error_Code;

private
   implementation-defined
end POSIX_Files;
```

5.2.1 Create and Remove Files

5.2.1.1 Synopsis

```
   procedure Create_Directory
      (Pathname :   in POSIX.Pathname;
       Permission : in POSIX_Permissions.Permission_Set);
   procedure Create_FIFO
      (Pathname :   in POSIX.Pathname;
       Permission : in POSIX_Permissions.Permission_Set);
   procedure Unlink (Pathname : in POSIX.Pathname);
   procedure Remove_Directory (Pathname : in POSIX.Pathname);
```

5.2.1.2 Description

A directory is a file that contains directory entries. Each directory entry shall have a unique name within that directory and associate a filename with a file. A directory entry is also referred to as a link, in that it links the filename to its file. A file may be referenced by more than one directory entry. Implementations may support linking of files across file systems, and they may also support links to directories. A file exists

as long as at least one directory entry (link) refers to it or there is at least one process has the file open.

One of the attributes maintained for a file shall be its *link count*, the number of directory entries (links) that refer to it.

`Create_Directory` shall create a directory. The name of the directory shall be specified by `Pathname`. `Create_FIFO` shall create a FIFO. The name of the FIFO shall be specified by `Pathname`. For each of the file creation operations, the file permission given to the new file shall be obtained from `Permission` as modified by the process permission set. (See 5.1.2.) The owner of the file shall be set to the effective user ID of the creating process, and the group of the file shall be set to the effective group ID of the process. Upon successful completion of a file creation operation, the Last Access Time, Last Modification Time, and Last Status Change Time of the new file shall be marked for update. (See 2.3.10.) Also, the Last Modification Time and Last Status Change Time values for the parent directory of the new file shall be marked for update.

File removal is accomplished through `Unlink`. `Unlink` shall remove the directory entry (link) named by the `Pathname` and decrement the link count of the file referenced by the entry. If the decremented link count is zero and no process has the file open, the space occupied by the file shall be freed, and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, removal of the file contents shall be postponed until all references to the file have been closed.

The parameter `Pathname` shall not name a directory unless the process has appropriate privileges and the implementation supports using `Unlink` on directories. An application should use `Remove_Directory` to remove a directory.

Upon successful completion, the Last Modification Time and Last Status Change Time of the parent directory shall be marked for update. (See 2.3.10.) Additionally, if the link count of the file is not zero, the Last Status Change Time of the file shall be marked for update.

A call to `Unlink` with a pathname that specifies a connected stream socket (*i.e.*, a socket of type `Stream_Socket` in states Connected, Sending Only, or Receiving Only, described in 18.2) shall have no effect on the socket. A call to `Unlink` specifying the pathname of a socket is required before a subsequent bind to the pathname will succeed. All further effects of `Unlink` on sockets are unspecified.

The effect of `Unlink` when `Pathname` specifies a character special file for use with XTI calls is unspecified. C

`Remove_Directory` shall remove a directory named by the `Pathname`. The directory shall only be removed if it is an empty directory. (See 2.2.2.54.) If the directory is the root directory or the current working directory of any process, the effect of this procedure is implementation defined.

Removal of a directory shall cause the link named by the `Pathname` to be removed and the link count of the directory to be decremented. When the link count becomes zero and no process has the directory open, the space occupied by the directory shall be freed, and the directory shall no longer be accessible. If one or more processes

have the directory open when the last link is removed, the dot and dot-dot entries, if present, shall be removed, and no new entries may be created in the directory. Removal of the directory is postponed until all references to the directory have been closed. Upon successful completion of the `Remove_Directory` operation, the Last Modification Time and Last Status Change Time of the parent directory shall be marked for update. (See 2.3.10.)

5.2.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

Search permission is denied for any component of the `Pathname` prefix or write permission is denied on the parent directory of `Pathname`.

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.) To determine these limits, the limit functions are applied to the parent directory of `Pathname`.

`File_Exists`

There is an entry in the file system named by `Pathname` for a call to `Create_Directory` or `Create_FIFO`.

`Too_Many_Links`

The link count in the parent directory of the file named by `Pathname` exceeds Links Maximum, for a call to `Create_Directory`.

`No_Such_File_Or_Directory`

(1) A component of the path prefix of the file named by `Pathname` does not exist, or it is a null string.

(2) `Pathname` does not exist, or it is a null string for a call to `Unlink` or `Remove_Directory`.

`No_Space_Left_On_Device`

The file system does not contain enough space to hold the new directory entry or to extend the contents of the parent directory of the new directory for a call to `Create_Directory` or `Create_FIFO`.

`Read_Only_File_System`

The parent directory of the file named by `Pathname` resides on a read-only file system.

`Not_A_Directory`

(1) A component of the path prefix of `Pathname` is not a directory for a call to `Create_Directory`, `Create_FIFO`, or `Unlink`.

(2) A component of the path named by `Pathname` is not a directory for a call to `Remove_Directory`.

Resource_Busy

The directory named by Pathname cannot be removed because it is being used by another process, during a call to Unlink or Remove_Directory, and the implementation considers attempting to unlink or remove a file that is in use to be an error.

Directory_Not_Empty or File_Exists

The directory named by Pathname is not an empty directory, when using Remove_Directory. Directory_Not_Empty is preferred in this circumstance.

In situations where multiple error codes may be set, which possible error code is set is unspecified.

5.2.2 Inquiries on File Types

5.2.2.1 Synopsis

```
function Is_File (Pathname : POSIX.Pathname) return Boolean;
function Is_Directory (Pathname : POSIX.Pathname) return Boolean;
function Is_FIFO (Pathname : POSIX.Pathname) return Boolean;
function Is_Character_Special_File
   (Pathname : POSIX.Pathname) return Boolean;
function Is_Block_Special_File
   (Pathname : POSIX.Pathname) return Boolean;
function Is_Socket (Pathname : POSIX.Pathname)
  return Boolean;
```
C

5.2.2.2 Description

Is_File shall return True if Pathname names a regular file and False otherwise. False shall also be returned if the file named by Pathname does not exist or if any other error occurred in determining the type of the file.

Is_Directory shall return True if Pathname names a directory and False otherwise. False shall also be returned if the file named by Pathname does not exist or if any other error occurred in determining the type of the file.

Is_FIFO shall return True if Pathname names a FIFO and False otherwise. False also shall be returned if the file named by Pathname does not exist or if any other error occurred in determining the type of the file.

Is_Character_Special_File shall return True if Pathname names a character special file and False otherwise. False also shall be returned if the file named by Pathname does not exist or if any other error occurred in determining the type of the file.

Is_Block_Special_File shall return True if Pathname names a block special file and False otherwise. False is also returned if the file named by Pathname does not exist or if any other error occurred in determining the type of the file.

Is_Socket shall return True if Pathname names a socket and False otherwise. False also shall be returned if the file named by Pathname does not exist or if any other error occurred in determining the type of the file.
C

5.2.2.3 Error Handling

No exceptions shall be returned by Is_File, Is_Directory, Is_FIFO, Is_Character_Special_File, , Is_Block_Special_File, or Is_Socket. c

5.2.3 Modify File Pathnames

5.2.3.1 Synopsis

```
procedure Link
   (Old_Pathname : in POSIX.Pathname;
    New_Pathname : in POSIX.Pathname);
procedure Rename
   (Old_Pathname : in POSIX.Pathname;
    New_Pathname : in POSIX.Pathname);
```

5.2.3.2 Description

Link shall atomically create an alternate pathname for a given file specified by Old_Pathname. New_Pathname is the pathname to be created. Link shall create a link to the file named by Old_Pathname under the pathname named by New_Pathname and increment the link count of the file by one.

If Link fails, no link shall be created, and the link count shall remain unchanged.

Old_Pathname shall not name a directory unless the user has appropriate privileges and the implementation supports using Link on directories.

Implementations may permit or forbid linking of files across file systems and also may require that the process have permission to access the file named by Old_Pathname.

Upon successful completion of Link, the Last Status Change Time of the file shall be marked for update. (See 2.3.10.) Additionally, the Last Modification Time and Last Status Change Time of the directory containing the new entry shall be marked for update.

The effect of Link on a socket or a character special file for use with XTI calls is unspecified. c

Rename shall change the name of the file from the name specified by Old_Pathname to the name specified by New_Pathname. If the two pathnames both refer to links to the same existing file, Rename shall return successfully and perform no other action.

If Old_Pathname names a file that is not a directory, New_Pathname shall not be the pathname of a directory. If the link named by New_Pathname exists, it shall be removed, and the file named by Old_Pathname shall be renamed to New_Pathname. In this case, a link named New_Pathname shall exist throughout the renaming operation and shall refer either to the file named by New_Pathname or Old_Pathname before the operation began. Write access permission is required for both the directory containing Old_Pathname and the directory containing New_Pathname.

If Old_Pathname names a directory, New_Pathname shall not name the pathname of a file that is not a directory. If the directory named by New_Pathname exists, it shall be removed, and the directory named by Old_Pathname shall be renamed to

New_Pathname. In this case, a link named by New_Pathname shall exist throughout the renaming operation and shall refer either to the file named by New_Pathname or Old_Pathname before the operation began. Thus, if New_Pathname names an existing directory, it shall be required to be an empty directory.

New_Pathname shall not contain a path prefix that names Old_Pathname. Write access permission shall be required for the directory containing Old_Pathname and the directory containing New_Pathname. If Old_Pathname names a directory, write access permission may be required for the directory named by Old_Pathname and, if it exists, the directory named by New_Pathname.

If the link named by New_Pathname exists and the link count of the file becomes zero when it is removed and no process has the file open, the space occupied by the file shall be freed, and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before Rename returns, but the removal of the file contents shall be postponed until all references to the file have been closed.

Upon successful completion of the Rename operation, the Last Modification Time and Last Status Change Time of both parent directories shall be marked for update. (See 2.3.10.)

The behavior of Rename is unspecified if either pathname is that of a socket or a character special file for use with XTI calls. c

5.2.3.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Permission_Denied

(1) Search permission is denied for any component named by Pathname.

(2) Read permission is denied on the pathname named by Old_Pathname, or write permission is denied on the parent directory of the pathname named by New_Pathname when using Link.

(3) Write permission is required and is denied for a directory named by Old_Pathname or New_Pathname for a call to Rename.

Filename_Too_Long

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.) To determine these limits, the limit functions are applied to the directory in which the Old_Pathname resides and the one in which New_Pathname is to be created.

No_Space_Left_On_Device

The directory that would contain New_Pathname cannot be extended.

Read_Only_File_System

The operation requires writing on a read-only file system.

`Not_A_Directory`

(1) A component of the path prefix of either `Old_Pathname` or `New_Pathname` is not a directory for a call to `Link`.

(2) A component of either path prefix is not a directory, or `Old_Pathname` names a directory and `New_Pathname` names a nondirectory file for a call to `Rename`.

`No_Such_File_Or_Directory`

(1) The file named by `Old_Pathname` does not exist, or either `Old_Pathname` or `New_Pathname` is an empty directory for a call to `Rename`.

(2) A component of either path prefix does not exist, the file named by `Old_Pathname` does not exist, or either `Old_Pathname` or `New_Pathname` is an empty directory, for a call to `Link`.

`Too_Many_Links`

For a call to `Link`, the number of links to the file named by `Old_Pathname` would exceed Links Maximum.

`File_Exists`

The file named by `New_Pathname` exists in the file system for a call to `Link`.

`Operation_Not_Permitted`

The file referenced by `Old_Pathname` is a directory, and the process does not have privilege to link directories or the implementation does not permit linking directories for a call to `Link`.

`Improper_Link`

The file named by `Old_Pathname` and the file named by `New_Pathname` reside on different file systems, and the implementation does not support links across file systems.

`Resource_Busy`

The parent directory of either `Old_Pathname` or `New_Pathname` cannot be modified because it is being used by another process, and the implementation considers this an error for a call to `Rename`.

`File_Exists` or `Directory_Not_Empty`

The file named by `New_Pathname` is an existing directory that contains entries other than dot and dot-dot for a call to `Rename`. `Directory_Not_Empty` is preferred in this circumstance.

`Invalid_Argument`

`New_Pathname` contains a path prefix that names the same directory as a component of `Old_Pathname`, for a call to `Rename`.

`Is_A_Directory`

`New_Pathname` names a directory and `Old_Pathname` does not name a directory for a call to `Rename`.

In situations where multiple error codes may be returned, which possible error code returned is unspecified.

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

Permission_Denied

A component of either path prefix denies search permission; or the requested link requires writing in a directory that denies write permissions; or the calling process does not have permission to access the file named by Old_Pathname, and permission is required by the implementation of Link.

5.2.4 Directory Iteration

5.2.4.1 Synopsis

```
type Directory_Entry is limited private;
function Filename_Of (D_Entry : Directory_Entry) return POSIX.Filename;
generic
    with procedure Action
       (D_Entry : in Directory_Entry;
        Quit :    in out Boolean);
procedure For_Every_Directory_Entry
   (Pathname : in POSIX.Pathname);
```

5.2.4.2 Description

Filename_Of shall return the filename provided by Directory_Entry.

The application program instantiates the generic procedure For_Every_Directory_Entry, providing an actual procedure for the generic formal procedure Action. When called, the newly created instance of For_Every_Directory_Entry shall call the actual procedure supplied for Action once for each directory entry in the directory named by Pathname. Action shall be able to force termination of For_Every_Directory_Entry either by setting Quit to True or by raising an exception. Prior to each call to Action, For_Every_Directory_Entry shall set Quit to False. The order in which the entries are presented is unspecified.

Exceptions raised by Action shall be propagated back to the caller of the instance of For_Every_Directory_Entry . After an exception is raised by Action, no more calls to Action shall occur. Action shall not be called with names whose length attribute is zero. If entries for dot or dot-dot exist, Action shall be called once for dot and once for dot-dot; otherwise, Action shall not be called for those entries.

If Action is called, the Last Access Time of the directory referenced by Pathname shall be marked for update at least once. It may be marked for update more than once. (See 2.3.10.)

If an entry is added or removed from the directory referenced by Pathname during execution of the instance of For_Every_Directory_Entry, whether Action is called for that entry is unspecified.

5.2.4.3 Error Handling

Filename_Of shall not raise any exceptions.

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Not_A_Directory

A component of the path prefix of Pathname does not refer to a directory.

Permission_Denied

Search permission is denied for a component of Pathname, or read permission is denied on the directory named by Pathname.

No_Such_File_Or_Directory

The directory named by Pathname does not exist.

Too_Many_Open_Files

An instance of For_Every_Directory_Entry causes a file to be opened, and the number of open files in the process exceeds Open Files Maximum.

Too_Many_Open_Files_In_System

An instance of For_Every_Directory_Entry causes a file to be opened, and the limit to the number of open files in the system is exceeded.

Filename_Too_Long

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.) To determine these limits, the limit functions are applied to Pathname.

5.2.5 Update File Status Information

5.2.5.1 Synopsis

```
procedure Change_Owner_And_Group
   (Pathname : in POSIX.Pathname;
    Owner :    in POSIX_Process_Identification.User_ID;
    Group :    in POSIX_Process_Identification.Group_ID);
procedure Change_Permissions
   (Pathname :   in POSIX.Pathname;
    Permission : in POSIX_Permissions.Permission_Set);
procedure Set_File_Times
   (Pathname :          in POSIX.Pathname;
    Access_Time :       in POSIX_Calendar.POSIX_Time;
    Modification_Time : in POSIX_Calendar.POSIX_Time);
procedure Set_File_Times (Pathname : in POSIX.Pathname);
```

5.2.5.2 Description

Change_Owner_And_Group shall change the user ID and the group ID of the file named by Pathname to the Owner_ID and Group_ID, respectively. Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file.

If the Change Owner Restriction option is imposed on Pathname,

— Changing the owner is restricted to processes with appropriate privileges.

— Changing the group is permitted by a process with an effective user ID equal to the user ID of the file, but without appropriate privileges, if and only if Owner is equal to the user ID of the file and Group is equal either to the effective group ID of the calling process or to one of its supplementary group IDs.

If the Pathname refers to a regular file, Set_User_ID and Set_Group_ID of the file mode shall be cleared upon successful return from Change_Owner_And_Group unless the call is made from a process with appropriate privileges, in which case it is implementation defined whether those modes are altered. The Last Status Change Time of the file shall be marked for update upon successful completion of this operation. (See 2.3.10.)

Change_Permissions shall change the permission set associated with the file named by Pathname. If the effective user ID of the calling process matches the file owner or the calling process has appropriate privileges, Change_Permissions shall change the permissions of the file to those specified in Permission. The Last Status Change Time of the file shall be marked for update upon completion of this operation. (See 2.3.10.)

Additional implementation-defined restrictions may cause Set_Group_ID and Set_User_ID in Permission to be ignored.

If the calling process does not have appropriate privileges, the group ID of the file does not match the effective group ID or one of the supplementary group IDs, one or more of — Owner_Execute, Group_Execute, and Others_Execute — are specified in Permission, and the file is a regular file, then Set_Group_ID in the permission set of the file shall be cleared upon successful return from Change_Permissions.

The effect on open file descriptions for files open at the time of the call to Change_Permissions is implementation defined.

The Last Status Change Time of the file shall be marked for update upon completion of this operation. (See 2.3.10.)

If the effective user ID of the process matches the owner of the file or if the process has appropriate privileges, Set_File_Times shall change the Last Access Time and Last Modification Time associated with the file named by Pathname. If both Access_Time and Modification_Time are specified, then the access time of the file shall be changed to Access_Time, and the Last Modification Time of the file shall be changed to Modification_Time.

If neither Access_Time nor Modification_Time are specified, then the Last Access Time and Last Modification Time of the file are set to the same value, the current value of POSIX_Calendar.Clock.

The Last Status Change Time of the file shall be marked for update upon successful completion of these operations. (See 2.3.10.)

5.2.5.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Permission_Denied

Search permission is denied for a component of Pathname, or the process does not have sufficient privilege to perform the operation.

Filename_Too_Long

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.) To determine these limits, the limit functions are applied to Pathname.

No_Such_File_Or_Directory

The file named by Pathname does not exist.

Not_A_Directory

A component of the path prefix of Pathname does not refer to a directory.

5.2.6 Check File Accessibility

5.2.6.1 Synopsis

```
type Access_Mode is (Read_Ok, Write_Ok, Execute_Ok);
type Access_Mode_Set is array (Access_Mode) of Boolean;
function Is_Accessible
   (Pathname :     POSIX.Pathname;
    Access_Modes : Access_Mode_Set)
  return Boolean;
function Accessibility
   (Pathname :     POSIX.Pathname;
    Access_Modes : Access_Mode_Set)
  return POSIX.Error_Code;
function Is_File_Present (Pathname : POSIX.Pathname) return Boolean;
function Existence (Pathname : POSIX.Pathname) return POSIX.Error_Code;
```

5.2.6.2 Description

The type Access_Mode shall represent access modes for a file. The values of this type are

Read_Ok

The file can be opened for input, or other read operations may be performed on the file by the process.

Write_Ok

The file can be opened for output, or other write operations may be performed on the file by the process.

Execute_Ok

The file may be executed via a call to Exec or Start_Process; or if the file is a directory, it may be searched.

Is_Accessible and Accessibility shall check the accessibility of the file named by Pathname for the file access modes, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

If any access permission is to be checked, each shall be checked individually, as described in 2.3.7. If the process has appropriate privileges, an implementation may indicate success even if none of the execute file permissions are granted.

`Is_Accessible` shall return `True` if the file named by `Pathname` supports the access modes indicated by `Access_Modes` and `False` otherwise.

`Accessibility` shall return `POSIX.No_Error` if the file named by `Pathname` supports the access modes indicated by `Access_Modes`. If the file does not support the indicated access mode, the appropriate `Error_Code` value shall be returned.

`Is_File_Present` shall return `True` if the file named by `Pathname` exists and `False` otherwise.

`Existence` shall return an error code representing the status of a file in the file system. If the file exists, the value `POSIX.No_Error` shall be returned. Otherwise, an appropriate error code value shall be returned.

5.2.6.3 Error Handling

No exceptions shall be raised by `Is_Accessible`, `Accessibility`, `Is_File_Present`, or `Existence`.

The error codes that may be returned by `Accessibility` and `Existence` are

`Permission_Denied`
: Search permission is denied for a component of `Pathname`, or any of the requested `Permissions` are denied for the file named by `Pathname`.

`Filename_Too_Long`
: The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.) To determine these limits, the limit functions are applied to `Pathname`.

`No_Such_File_Or_Directory`
: The file named by `Pathname` does not exist.

`Not_A_Directory`
: A component of the path prefix of `Pathname` does not refer to a directory.

5.3 Package POSIX_File_Status

The package `POSIX_File_Status` defines the standard attributes of files in the file system and provides the operations for examining them.

```
with POSIX,
     POSIX_Permissions,
     POSIX_Process_Identification,
     POSIX_IO,
     POSIX_Calendar;
package POSIX_File_Status is
   -- 5.3.1 Access File Status
   type Status is private;
   function Get_File_Status (Pathname : POSIX.Pathname) return Status;
   function Get_File_Status
      (File : POSIX_IO.File_Descriptor) return Status;
```

```
   -- 5.3.2 Access Status Information
   type File_ID is private;
   type Device_ID is private;
   subtype Links is natural range 0 .. POSIX.Link_Limit_Maxima'Last;
   function Permission_Set_Of
      (File_Status : Status) return POSIX_Permissions.Permission_Set;
   function File_ID_Of (File_Status : Status) return File_ID;
   function Device_ID_Of (File_Status : Status) return Device_ID;
   function Link_Count_Of (File_Status : Status) return Links;
   function Owner_Of
      (File_Status : Status) return POSIX_Process_Identification.User_ID;
   function Group_Of
      (File_Status : Status) return POSIX_Process_Identification.Group_ID;
   function Size_Of (File_Status : Status) return POSIX.IO_Count;
   function Last_Access_Time_Of
      (File_Status : Status) return POSIX_Calendar.POSIX_Time;
   function Last_Modification_Time_Of
      (File_Status : Status) return POSIX_Calendar.POSIX_Time;
   function Last_Status_Change_Time_Of
      (File_Status : Status) return POSIX_Calendar.POSIX_Time;
   function Is_Directory (File_Status : Status) return Boolean;
   function Is_Character_Special_File
      (File_Status : Status) return Boolean;
   function Is_Block_Special_File
      (File_Status : Status) return Boolean;
   function Is_Regular_File (File_Status : Status) return Boolean;
   function Is_FIFO (File_Status : Status) return Boolean;
   function Is_Shared_Memory
      (File_Status : in Status) return Boolean;
   function Is_Message_Queue
      (File_Status : in Status) return Boolean;
   function Is_Semaphore
      (File_Status : in Status) return Boolean;
   function Is_Socket (File_Status : Status) return Boolean;                  |c
private
   implementation-defined
end POSIX_File_Status;
```

5.3.1 Access File Status

5.3.1.1 Synopsis

```
   type Status is private;
   function Get_File_Status (Pathname : POSIX.Pathname) return Status;
   function Get_File_Status
      (File : POSIX_IO.File_Descriptor) return Status;
```

5.3.1.2 Description

The type `Status` shall hold status information for a specific file or `POSIX_IO.File_Descriptor`. There are two overloaded functions, both named `Get_File_Status`. One identifies the file by its name through a `POSIX_String` and the other takes an open file, represented by its `POSIX_IO.File_Descriptor`. Each operation shall return the status information for the file named by `Pathname` or `File`.

5.3.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: Search permission is denied for a component of `Pathname`.

`Filename_Too_Long`
: The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.) To determine these limits, the limit functions are applied to `Pathname`.

`No_Such_File_Or_Directory`
: The file named by `Pathname` does not exist.

`Not_A_Directory`
: A component of the path prefix of `Pathname` does not refer to a directory.

`Bad_File_Descriptor`
: `File` is not a valid `File_Descriptor`.

5.3.2 Access Status Information

5.3.2.1 Synopsis

```
type File_ID is private;
type Device_ID is private;
subtype Links is natural range 0 .. POSIX.Link_Limit_Maxima'Last;
function Permission_Set_Of
   (File_Status : Status) return POSIX_Permissions.Permission_Set;
function File_ID_Of (File_Status : Status) return File_ID;
function Device_ID_Of (File_Status : Status) return Device_ID;
function Link_Count_Of (File_Status : Status) return Links;
function Owner_Of
   (File_Status : Status) return POSIX_Process_Identification.User_ID;
function Group_Of
   (File_Status : Status) return POSIX_Process_Identification.Group_ID;
function Size_Of (File_Status : Status) return POSIX.IO_Count;
function Last_Access_Time_Of
   (File_Status : Status) return POSIX_Calendar.POSIX_Time;
function Last_Modification_Time_Of
   (File_Status : Status) return POSIX_Calendar.POSIX_Time;
function Last_Status_Change_Time_Of
   (File_Status : Status) return POSIX_Calendar.POSIX_Time;
function Is_Directory (File_Status : Status) return Boolean;
function Is_Character_Special_File
   (File_Status : Status) return Boolean;
function Is_Block_Special_File
   (File_Status : Status) return Boolean;
function Is_Regular_File (File_Status : Status) return Boolean;
function Is_FIFO (File_Status : Status) return Boolean;
function Is_Shared_Memory
   (File_Status : in Status) return Boolean;
function Is_Message_Queue
   (File_Status : in Status) return Boolean;
function Is_Semaphore
   (File_Status : in Status) return Boolean;
function Is_Socket (File_Status : Status) return Boolean;                    |c
```

5.3.2.2 Description

Values of the type `File_ID` shall uniquely identify each file on a specific device. Values of the type `Device_ID` shall uniquely identify each device in the system. Together, they shall uniquely identify a file in the system.

Values of the subtype `Links` shall contain the number of links to a specific file (identified by a `File_ID` and/or `Device_ID`).

`Permission_Set_Of` shall return the file permissions for the specified file.

`File_ID_Of` shall return the file serial number associated with the file.

`Device_ID_Of` shall return the device serial number for the device that contains the file.

`Link_Count_Of` shall return the number of links to the specified file.

`Owner_Of` shall return the owner associated with the file.

`Group_Of` shall return the group associated with the file.

For regular files, `Size_Of` shall return the number of bytes stored in the file. For other file types, the use of this function is unspecified.

`Last_Access_Time_Of` shall return the last time the file was accessed by the system.

`Last_Modification_Time_Of` shall return the last time the file was modified by the system.

`Last_Status_Change_Time_Of` shall return the last time status information associated with the file was changed.

Values of type `Status` can contain status information for one of several types of objects. The functions described in this section are provided to determine the type of object. Each of the functions takes a single input parameter of type `Status`.

`Is_Directory` shall return `True` if the file is a directory and `False` otherwise.

`Is_Character_Special_File` shall return `True` if the file is a character special file and `False` otherwise.

`Is_Block_Special_File` shall return `True` if the file is a block special file and `False` otherwise.

`Is_Regular_File` shall return `True` if the file is a regular file and `False` otherwise.

`Is_FIFO` shall return `True` if the file is a FIFO special file and `False` otherwise.

`Is_Shared_Memory` shall return `True` if the file is that of a POSIX shared memory object and it is implemented as a distinct file type; otherwise, it shall return `False`.

`Is_Message_Queue` shall return `True` if the file is that of a POSIX message queue object and it is implemented as a distinct file type; otherwise, it shall return `False`.

`Is_Semaphore` shall return `True` if the file is that of a POSIX semaphore object and it is implemented as a distinct file type; otherwise, it shall return `False`.

NOTE: Message queues and semaphores may, but need not be, associated with file descriptors. If they are associated with file descriptors, they may, but need not be, implemented as distinct file types. Shared memory objects are always associated with file descriptors, and they may, but need not be, implemented as a distinct file type. Therefore, it is implementation defined whether `Is_Message_Queue`, `Is_Semaphore`, *or* `Is_Shared_Memory` *return* `True`.

`Is_Socket` shall return `True` if the file is a socket and `False` otherwise. |c

5.3.2.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: `File_Status` is invalid.

If the following condition occurs for a call to `Size_Of`, the exception `POSIX_Error` shall be raised, with the corresponding error code:

`Invalid_Argument`
: The file is not a regular file.

5.4 Package POSIX_Configurable_File_Limits

The package `POSIX_Configurable_File_Limits` provides the application developer with execution-time access to implementation-defined limitations and other characteristics that may vary across the file system. Each separate path may have different characteristics. It is also possible that these characteristics may vary for a given path over time, particularly if the device may be removed and replaced on the system.

```
with POSIX,
     POSIX_IO,
     POSIX_Limits,
     POSIX_Options;
package POSIX_Configurable_File_Limits is
   -- 5.4.1 File Limits
   --  Link Limits
   function Link_Is_Limited (Pathname : POSIX.Pathname)         -- obsolescent
     return Boolean;                                            -- obsolescent
   function Link_Is_Limited (File : POSIX_IO.File_Descriptor)   -- obsolescent
     return Boolean;                                            -- obsolescent
   function Link_Limit (Pathname : POSIX.Pathname)              -- obsolescent
     return POSIX.Link_Limit_Maxima;                            -- obsolescent
   function Link_Limit (File : POSIX_IO.File_Descriptor)        -- obsolescent
     return POSIX.Link_Limit_Maxima;                            -- obsolescent
   function Links_Are_Limited (Pathname : POSIX.Pathname)
     return Boolean
     renames Link_Is_Limited;
   function Links_Are_Limited (File : POSIX_IO.File_Descriptor)
     return Boolean
     renames Link_Is_Limited;
   function Links_Maximum (Pathname : POSIX.Pathname)
     return POSIX_Limits.Links_Maxima
     renames Link_Limit;
```

```
function Links_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Links_Maxima
  renames Link_Limit;
-- Input line limits
function Input_Line_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Input_Line_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Input_Line_Limit (Pathname : POSIX.Pathname)        -- obsolescent
  return POSIX.Input_Line_Limit_Maxima;                      -- obsolescent
function Input_Line_Limit (File : POSIX_IO.File_Descriptor)  -- obsolescent
  return POSIX.Input_Line_Limit_Maxima;                      -- obsolescent
function Input_Line_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Input_Line_Maxima
  renames Input_Line_Limit;
function Input_Line_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Input_Line_Maxima
  renames Input_Line_Limit;
-- Input queue limits
function Input_Queue_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Input_Queue_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Input_Queue_Limit (Pathname : POSIX.Pathname)       -- obsolescent
  return POSIX.Input_Queue_Limit_Maxima;                     -- obsolescent
function Input_Queue_Limit (File : POSIX_IO.File_Descriptor) -- obsolescent
  return POSIX.Input_Queue_Limit_Maxima;                     -- obsolescent
function Input_Queue_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Input_Queue_Maxima
  renames Input_Queue_Limit;
function Input_Queue_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Input_Queue_Maxima
  renames Input_Queue_Limit;
-- Filename and pathname limits
function Filename_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Filename_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Filename_Limit (Pathname : POSIX.Pathname)          -- obsolescent
  return POSIX.Filename_Limit_Maxima;                        -- obsolescent
function Filename_Limit (File : POSIX_IO.File_Descriptor)    -- obsolescent
  return POSIX.Filename_Limit_Maxima;                        -- obsolescent
function Filename_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Filename_Maxima
  renames Filename_Limit;
function Filename_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Filename_Maxima
  renames Filename_Limit;
function Pathname_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Pathname_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Pathname_Limit (Pathname : POSIX.Pathname)          -- obsolescent
  return POSIX.Pathname_Limit_Maxima;                        -- obsolescent
function Pathname_Limit (File : POSIX_IO.File_Descriptor)    -- obsolescent
  return POSIX.Pathname_Limit_Maxima;                        -- obsolescent
```

```
   function Pathname_Maximum (Pathname : POSIX.Pathname)
     return POSIX_Limits.Pathname_Maxima
     renames Pathname_Limit;
   function Pathname_Maximum (File : POSIX_IO.File_Descriptor)
     return POSIX_Limits.Pathname_Maxima
     renames Pathname_Limit;
   -- Pipe length limits
   function Pipe_Length_Is_Limited (Pathname : POSIX.Pathname)
     return Boolean;
   function Pipe_Length_Is_Limited (File : POSIX_IO.File_Descriptor)
     return Boolean;
   function Pipe_Length_Limit (Pathname : POSIX.Pathname)       -- obsolescent
     return POSIX.Pipe_Limit_Maxima;                            -- obsolescent
   function Pipe_Length_Limit (File : POSIX_IO.File_Descriptor) -- obsolescent
     return POSIX.Pipe_Limit_Maxima;                            -- obsolescent
   function Pipe_Length_Maximum (Pathname : POSIX.Pathname)
     return POSIX_Limits.Pipe_Length_Maxima
     renames Pipe_Length_Limit;
   function Pipe_Length_Maximum (File : POSIX_IO.File_Descriptor)
     return POSIX_Limits.Pipe_Length_Maxima
     renames Pipe_Length_Limit;
   -- Socket buffer limits
   function Socket_Buffer_Is_Limited (File : POSIX_IO.File_Descriptor)
     return Boolean;
   function Socket_Buffer_Is_Limited (Pathname : POSIX.Pathname)
     return Boolean;
   function Socket_Buffer_Limit (File : POSIX_IO.File_Descriptor)
     return POSIX_Limits.Socket_Buffer_Maxima;
   function Socket_Buffer_Limit (Pathname : POSIX.Pathname)
     return POSIX_Limits.Socket_Buffer_Maxima;
   -- 5.4.2 File Restrictions
   function Change_Owner_Is_Restricted (Pathname : POSIX.Pathname)
     return POSIX_Options.Change_Owner_Restriction;
   function Change_Owner_Is_Restricted (File : POSIX_IO.File_Descriptor)
     return POSIX_Options.Change_Owner_Restriction;
   function Filename_Is_Truncated (Pathname : POSIX.Pathname)
     return POSIX_Options.Filename_Truncation;
   function Filename_Is_Truncated (File : POSIX_IO.File_Descriptor)
     return POSIX_Options.Filename_Truncation;
   -- 5.4.3 Pathname-Specific Options
   function Synchronized_IO_Is_Supported (Pathname : POSIX.Pathname)
     return Boolean;
   function Synchronized_IO_Is_Supported (File : POSIX_IO.File_Descriptor)
     return Boolean;
   function Asynchronous_IO_Is_Supported (Pathname : POSIX.Pathname)
     return Boolean;
   function Asynchronous_IO_Is_Supported (File : POSIX_IO.File_Descriptor)
     return Boolean;
   function Prioritized_IO_Is_Supported (Pathname : POSIX.Pathname)
     return Boolean;
   function Prioritized_IO_Is_Supported (File : POSIX_IO.File_Descriptor)
     return Boolean;

end POSIX_Configurable_File_Limits;
```

5.4.1 File Limits

5.4.1.1 Synopsis

```
-- Link Limits
function Link_Is_Limited (Pathname : POSIX.Pathname)          -- obsolescent
  return Boolean;                                             -- obsolescent
function Link_Is_Limited (File : POSIX_IO.File_Descriptor)    -- obsolescent
  return Boolean;                                             -- obsolescent
function Link_Limit (Pathname : POSIX.Pathname)               -- obsolescent
  return POSIX.Link_Limit_Maxima;                             -- obsolescent
function Link_Limit (File : POSIX_IO.File_Descriptor)         -- obsolescent
  return POSIX.Link_Limit_Maxima;                             -- obsolescent
function Links_Are_Limited (Pathname : POSIX.Pathname)
  return Boolean
  renames Link_Is_Limited;
function Links_Are_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean
  renames Link_Is_Limited;
function Links_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Links_Maxima
  renames Link_Limit;
function Links_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Links_Maxima
  renames Link_Limit;
-- Input line limits
function Input_Line_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Input_Line_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Input_Line_Limit (Pathname : POSIX.Pathname)         -- obsolescent
  return POSIX.Input_Line_Limit_Maxima;                       -- obsolescent
function Input_Line_Limit (File : POSIX_IO.File_Descriptor)   -- obsolescent
  return POSIX.Input_Line_Limit_Maxima;                       -- obsolescent
function Input_Line_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Input_Line_Maxima
  renames Input_Line_Limit;
function Input_Line_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Input_Line_Maxima
  renames Input_Line_Limit;
-- Input queue limits
function Input_Queue_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Input_Queue_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Input_Queue_Limit (Pathname : POSIX.Pathname)        -- obsolescent
  return POSIX.Input_Queue_Limit_Maxima;                      -- obsolescent
function Input_Queue_Limit (File : POSIX_IO.File_Descriptor) -- obsolescent
  return POSIX.Input_Queue_Limit_Maxima;                      -- obsolescent
function Input_Queue_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Input_Queue_Maxima
  renames Input_Queue_Limit;
function Input_Queue_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Input_Queue_Maxima
  renames Input_Queue_Limit;
-- Filename and pathname limits
function Filename_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Filename_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
```

```
function Filename_Limit (Pathname : POSIX.Pathname)          -- obsolescent
  return POSIX.Filename_Limit_Maxima;                        -- obsolescent
function Filename_Limit (File : POSIX_IO.File_Descriptor)    -- obsolescent
  return POSIX.Filename_Limit_Maxima;                        -- obsolescent
function Filename_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Filename_Maxima
  renames Filename_Limit;
function Filename_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Filename_Maxima
  renames Filename_Limit;
function Pathname_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Pathname_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Pathname_Limit (Pathname : POSIX.Pathname)          -- obsolescent
  return POSIX.Pathname_Limit_Maxima;                        -- obsolescent
function Pathname_Limit (File : POSIX_IO.File_Descriptor)    -- obsolescent
  return POSIX.Pathname_Limit_Maxima;                        -- obsolescent
function Pathname_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Pathname_Maxima
  renames Pathname_Limit;
function Pathname_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Pathname_Maxima
  renames Pathname_Limit;
-- Pipe length limits
function Pipe_Length_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Pipe_Length_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Pipe_Length_Limit (Pathname : POSIX.Pathname)       -- obsolescent
  return POSIX.Pipe_Limit_Maxima;                            -- obsolescent
function Pipe_Length_Limit (File : POSIX_IO.File_Descriptor) -- obsolescent
  return POSIX.Pipe_Limit_Maxima;                            -- obsolescent
function Pipe_Length_Maximum (Pathname : POSIX.Pathname)
  return POSIX_Limits.Pipe_Length_Maxima
  renames Pipe_Length_Limit;
function Pipe_Length_Maximum (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Pipe_Length_Maxima
  renames Pipe_Length_Limit;
-- Socket buffer limits
function Socket_Buffer_Is_Limited (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Socket_Buffer_Is_Limited (Pathname : POSIX.Pathname)
  return Boolean;
function Socket_Buffer_Limit (File : POSIX_IO.File_Descriptor)
  return POSIX_Limits.Socket_Buffer_Maxima;
function Socket_Buffer_Limit (Pathname : POSIX.Pathname)
  return POSIX_Limits.Socket_Buffer_Maxima;
```

C

5.4.1.2 Description

There are two overloaded subprograms for each limit or option to be queried. One accepts a pathname of a file or directory. The other accepts an open file descriptor. (The parameters are named `Pathname` and `File`, respectively.)

The implementation shall support all of the limits and options identified by these subprograms and may support others.

`Links_Are_Limited` shall return `True` if there is a limit to the number of links permissible on the file and `False` otherwise.

`Link_Is_Limited`, an alternate name for `Links_Are_Limited`, is obsolescent.

`Links_Maximum` shall provide the maximum value of the link count of a file. If `Links_Are_Limited` for the given parameter returns `False`, then `Links_Maximum` shall return `POSIX_Limits.Links_Maxima'Last`.

`Link_Limit`, an alternate name for `Links_Maximum`, is obsolescent.

`Input_Line_Is_Limited` shall return `True` if there is a maximum number of bytes in a canonical input line as indicated by `File` or `Pathname` and `False` otherwise. The behavior of `Input_Line_Is_Limited` is unspecified if `File` or `Pathname` do not refer to a terminal device.

`Input_Line_Maximum` shall provide the maximum number of bytes in a terminal canonical input line identified by the `File` or `Pathname`. If `Input_Line_Is_Limited` for the given parameter returns `False`, the `Input_Line_Maximum` shall return `POSIX_Limits.Input_Line_Maxima 'Last`. It is unspecified whether an implementation supports association of the `Input_Line_Maximum` if `File` or `Pathname` does not refer to a terminal device.

`Input_Line_Limit`, an alternate name for `Input_Line_Maximum`, is obsolescent.

`Input_Queue_Is_Limited` shall return `True` if there is a maximum amount of space available in the terminal input queue associated with `File` or `Pathname` and `False` otherwise. The behavior of `Input_Queue_Is_Limited` is unspecified if `File` or `Pathname` does not refer to a terminal device.

`Input_Queue_Maximum` shall return the minimum number of bytes for which the implementation guarantees space will be available in a terminal input queue associated with `File` or `Pathname`. This *implementation* minimum is the maximum number of bytes a portable *application* may require to be typed as input before reading those bytes. If `Input_Queue_Is_Limited` returns `False` for the given parameter, then `Input_Queue_Maximum` shall return `POSIX_Limits.Input_Queue_Maxima 'Last`. It is unspecified whether an implementation supports association of the `Input_Queue_Maximum` if `File` or `Pathname` does not refer to a terminal device.

`Input_Queue_Limit`, an alternate name for `Input_Queue_Maximum`, is obsolescent.

`Filename_Is_Limited` shall return `True` if there is a limit to the length of a filename associated with `File` or `Pathname` and `False` otherwise. If `File` or `Pathname` refers to a directory, then `True` shall be returned to indicate that file names within the indicated directory are limited. If `File` or `Pathname` does not refer to a directory, then the behavior is unspecified.

`Filename_Maximum` shall return the maximum number of POSIX characters in the file name associated with `File` or `Pathname`. If `File` or `Pathname` refer to a directory, the value returned applies to the file names within the directory. If `Filename_Is_Limited` returns `False` for the given parameter, then `Filename_Maximum` shall return `POSIX_Limits.Filename_Maxima'Last`. If `File` or `Pathname` does not refer to a directory, then the behavior is unspecified.

`Filename_Limit`, an alternate name for `Filename_Maximum`, is obsolescent.

`Pathname_Is_Limited` shall return `True` if there is a limit to the length of any pathname associated with `File` or `Pathname` and `False` otherwise. If `File` or `Pathname` refers to a directory, then `True` indicates that file names relative to the named directory are limited. If `File` or `Pathname` does not refer to a directory, then the behavior is unspecified.

`Pathname_Maximum` shall return the maximum number of POSIX characters in the pathname associated with `File` or `Pathname`. If `File` or `Pathname` refers to a directory, the value returned applies to pathnames relative to the named directory. If `Pathname_Is_Limited` returns `False` for the given parameter, then `Pathname_Maximum` shall return `POSIX_Limits.Pathname_Maxima'Last`. If `File` or `Pathname` does not refer to a directory, then the behavior is unspecified.

`Pathname_Limit`, an alternate name for `Pathname_Maximum`, is obsolescent.

`Pipe_Length_Is_Limited` shall return `True` if there is a limit to the number of bytes that can be written at once to a FIFO special file named by `File` or `Pathname` and `False` otherwise. If `File` or `Pathname` refers to a directory, then this characteristic applies to any FIFO special file created within the named directory. If `File` or `Pathname` refers to any other type of file, the behavior is unspecified

`Pipe_Length_Maximum` shall return the maximum number of bytes that can be written at once to a FIFO special file named by `File` or `Pathname`. If `Filename` or `Pathname` refers to a directory, the value returned applies to any FIFO special file that exists or can be created within the named directory. If `File` or `Pathname` refers to any other type of file, the behavior is unspecified. If `Pipe_Length_Is_Limited` returns `False` for the given parameter, then `Pipe_Length_Maximum` shall return `POSIX_Limits.Pipe_Length_Maxima'Last`.

`Pipe_Length_Limit`, an alternate name for `Pipe_Length_Maximum`, is obsolescent.

`Socket_Buffer_Is_Limited` shall return `True` if there is a limit to the number of bytes that can be buffered on a socket for send and receive operations and `False` otherwise. If `File` or `Pathname` does not refer to a socket, the behavior is unspecified.

`Socket_Buffer_Limit` shall return the maximum number of bytes that can be buffered for send and receive operations on the socket named by `File` or `Pathname`. If `File` or `Pathname` does not refer to a socket, the behavior is unspecified. If `Socket_Buffer_Is_Limited` returns `False` for the given parameter, then `Socket_Buffer_Limit` shall return `POSIX_Limits.Socket_Buffer_Maxima'Last`. C

5.4.1.3 Error Handling

If the `Pathname` or `File` parameter of one of these operations is referenced to determine the value returned by the operation and any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`No_Such_File_Or_Directory`

`Pathname` does not identify an existing file.

`Not_A_Directory`

A component of the path prefix of `Pathname` is not a directory.

`Permission_Denied`

Search permission is denied for a component of the path prefix of `Pathname`.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`

`File` is not a valid file descriptor.

`Invalid_Argument`

The implementation does not define the value of the function for the specified file.

5.4.2 File Restrictions

5.4.2.1 Synopsis

```
function Change_Owner_Is_Restricted (Pathname : POSIX.Pathname)
  return POSIX_Options.Change_Owner_Restriction;
function Change_Owner_Is_Restricted (File : POSIX_IO.File_Descriptor)
  return POSIX_Options.Change_Owner_Restriction;
function Filename_Is_Truncated (Pathname : POSIX.Pathname)
  return POSIX_Options.Filename_Truncation;
function Filename_Is_Truncated (File : POSIX_IO.File_Descriptor)
  return POSIX_Options.Filename_Truncation;
```

5.4.2.2 Description

`Change_Owner_Is_Restricted` shall indicate whether the use of `Change_Owner_And_Group` (see 5.2.5) on the file named by `File` or `Pathname` is restricted. If `Change_Owner_Is_Restricted` is true, then `Change_Owner_And_Group` is restricted to a process with appropriate privileges and to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs. If `File` or `Pathname` refers to a directory, the value returned shall apply to any files defined in this standard, other than directories, that exist or can be created within the directory. If `File` or `Pathname` does not refer to a directory, then the behavior is unspecified.

`Filename_Is_Truncated` shall return `False` if pathname components longer than the limit provided by `Filename_Limit` (see 5.4.1) will cause an error and `True` otherwise. If `File` or `Pathname` refers to a directory, the value returned shall apply to file names within the named directory. If `File` or `Pathname` does not refer to a directory, then the behavior is unspecified.

5.4.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Filename_Too_Long`
: The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`No_Such_File_Or_Directory`
: `Pathname` does not identify an existing file.

`Not_A_Directory`
: A component of the path prefix of `Pathname` is not a directory.

`Permission_Denied`
: Search permission is denied for a component of the path prefix of `Pathname`.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: `File` is not a valid file descriptor.

`Invalid_Argument`
: The implementation does not define the value of the function for the specified file.

5.4.3 Pathname-Specific Options

5.4.3.1 Synopsis

```
function Synchronized_IO_Is_Supported (Pathname : POSIX.Pathname)
  return Boolean;
function Synchronized_IO_Is_Supported (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Asynchronous_IO_Is_Supported (Pathname : POSIX.Pathname)
  return Boolean;
function Asynchronous_IO_Is_Supported (File : POSIX_IO.File_Descriptor)
  return Boolean;
function Prioritized_IO_Is_Supported (Pathname : POSIX.Pathname)
  return Boolean;
function Prioritized_IO_Is_Supported (File : POSIX_IO.File_Descriptor)
  return Boolean;
```

5.4.3.2 Description

`Synchronized_IO_Is_Supported` shall return `True` if the Synchronized I/O option is supported for the file specified by `File` or `Pathname`.

`Asynchronous_IO_Is_Supported` shall return `True` if the Asynchronous I/O option is supported for the file specified by `File` or `Pathname`.

`Prioritized_IO_Is_Supported` shall return `True` if the Prioritized I/O option is supported for the file specified by `File` or `Pathname`.

If `File` or `Pathname` refers to a directory, it is unspecified whether the value of the function is defined.

5.4.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`No_Such_File_Or_Directory`

`Pathname` does not identify an existing file.

`Not_A_Directory`

A component of the path prefix of `Pathname` is not a directory.

`Permission_Denied`

Search permission is denied for a component of the path prefix of `Pathname`.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`

`File` is not a valid file descriptor.

`Invalid_Argument`

The implementation does not define the value of the function for the specified file.

Section 6: I/O Primitives

This section contains the packages POSIX_IO and POSIX_Asynchronous_IO, including all of the POSIX I/O operations, and the package POSIX_File_Locking, which provides support for file locking. The relationship between POSIX I/O and Ada I/O is defined in 8.1.

6.1 Package POSIX_IO

```
with Ada_Streams,
     POSIX,
     POSIX_Process_Identification,
     POSIX_Permissions,
     System;                                                              |c
package POSIX_IO is
   -- 6.1.1 Open or Create a File
   -- File Descriptors
   type File_Descriptor is range 0 .. POSIX.Open_Files_Maxima'Last - 1;
   Standard_Input :  constant File_Descriptor := 0;
   Standard_Output : constant File_Descriptor := 1;
   Standard_Error :  constant File_Descriptor := 2;
   type IO_Offset is implementation-defined-integer;
   -- File modes and options
   type File_Mode is (Read_Only, Write_Only, Read_Write);
   type Open_Option_Set is new POSIX.Option_Set;
   -- Empty˙Set, "+", and "-" are derived operations
   Non_Blocking : constant Open_Option_Set := implementation-defined;
   Append :       constant Open_Option_Set := implementation-defined;
   Truncate :     constant Open_Option_Set := implementation-defined;
   Exclusive :    constant Open_Option_Set := implementation-defined;
   Not_Controlling_Terminal : constant Open_Option_Set := implementation-defined;
   Signal_When_Socket_Ready : constant Open_Option_Set := implementation-defined;  |c
   File_Synchronized :
      constant POSIX_IO.Open_Option_Set := implementation-defined;
   Data_Synchronized :
      constant POSIX_IO.Open_Option_Set := implementation-defined;
   Read_Synchronized :
      constant POSIX_IO.Open_Option_Set := implementation-defined;
   -- Operations to open or close file descriptors
   function Open
      (Name :           POSIX.Pathname;
       Mode :           File_Mode;
       Options :        Open_Option_Set := Empty_Set;
       Masked_Signals : POSIX.Signal_Masking := POSIX.RTS_Signals)
      return File_Descriptor;
   function Open_Or_Create
      (Name :        POSIX.Pathname;
       Mode :        File_Mode;
       Permissions : POSIX_Permissions.Permission_Set;
       Options :     Open_Option_Set := Empty_Set;
       Masked_Signals : POSIX.Signal_Masking := POSIX.RTS_Signals)
      return File_Descriptor;
   function Is_Open (File : File_Descriptor)
      return Boolean;
   procedure Close
      (File :           in File_Descriptor;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

```
function Duplicate
   (File :   File_Descriptor;
    Target : File_Descriptor := 0)
   return File_Descriptor;
function Duplicate_And_Close
   (File :   File_Descriptor;
    Target : File_Descriptor := 0;
    Masked_Signals : POSIX.Signal_Masking := POSIX.RTS_Signals)
   return File_Descriptor;
procedure Create_Pipe
   (Read_End :  out File_Descriptor;
    Write_End : out File_Descriptor);
-- 6.1.2 I/O Buffer Type
subtype IO_Buffer is POSIX.POSIX_String; -- obsolescent
-- 6.1.3 Read from a File
procedure Read                              -- obsolescent
   (File :           in  File_Descriptor;
    Buffer :         out IO_Buffer;
    Last :           out POSIX.IO_Count;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Read
   (File :           in  File_Descriptor;
    Buffer :         out Ada_Streams.Stream_Element_Array;
    Last :           out Ada_Streams.Stream_Element_Offset;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
generic
   type T is private;
procedure Generic_Read
   (File :           in  File_Descriptor;
    Item :           out T;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
-- 6.1.4 Write to a File
procedure Write                             -- obsolescent
   (File :           in  File_Descriptor;
    Buffer :         in  IO_Buffer;
    Last :           out POSIX.IO_Count;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Write
   (File :           in  File_Descriptor;
    Buffer :         in  Ada_Streams.Stream_Element_Array;
    Last :           out Ada_Streams.Stream_Element_Offset;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
generic
   type T is private;
procedure Generic_Write
   (File :           in File_Descriptor;
    Item :           in T;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
-- 6.1.5 File Position Operations
type Position is
   (From_Beginning, From_Current_Position, From_End_Of_File);
procedure Seek
   (File :           in  File_Descriptor;
    Offset :         in  IO_Offset;
    Result :         out IO_Offset;
    Starting_Point : in  Position := From_Beginning);
function File_Size (File : File_Descriptor)
   return POSIX.IO_Count;
```

```
   function File_Position (File : File_Descriptor)
      return IO_Offset;
   -- 6.1.6 Terminal Operations
   function Is_A_Terminal (File : File_Descriptor)
      return Boolean;
   function Get_Terminal_Name (File : File_Descriptor)
      return POSIX.Pathname;
   -- 6.1.7 File Control
   procedure Get_File_Control
      (File :    in  File_Descriptor;
       Mode :    out File_Mode;
       Options : out Open_Option_Set);
   procedure Set_File_Control
      (File :    in File_Descriptor;
       Options : in Open_Option_Set);
   function Get_Close_On_Exec (File : File_Descriptor)
        return Boolean;
   procedure Set_Close_On_Exec
      (File : in File_Descriptor;
       To :   in Boolean := True);
   -- 6.1.8 Update File Status Information
   procedure Change_Permissions
      (File :       in POSIX_IO.File_Descriptor;
       Permission : in POSIX_Permissions.Permission_Set);
   -- 6.1.9 Truncate File to A Specified Length
   procedure Truncate_File
      (File :   in POSIX_IO.File_Descriptor;
       Length : in POSIX.IO_Count);
   -- 6.1.10 Synchronize the State of a File
   procedure Synchronize_File (File : in POSIX_IO.File_Descriptor);
   -- 6.1.11 Data Synchronization
   procedure Synchronize_Data (File : in POSIX_IO.File_Descriptor);
   -- 6.1.12 Socket File Ownership
   procedure Get_Owner
      (File    : in  File_Descriptor;
       Process : out POSIX_Process_Identification.Process_ID;
       Group   : out POSIX_Process_Identification.Process_Group_ID);
   procedure Set_Socket_Process_Owner
      (File    : in File_Descriptor;
       Process : in POSIX_Process_Identification.Process_ID);
   procedure Set_Socket_Group_Owner
      (File    : in File_Descriptor;
       Group   : in POSIX_Process_Identification.Process_Group_ID);
   -- 6.1.13 I/O Vector Type
   type IO_Vector is limited private;
   procedure Set_Buffer
      (Vector : in out IO_Vector;
       Buffer : in     System.Address;
       Length : in     POSIX.IO_Count);
   procedure Get_Buffer
      (Vector : in  IO_Vector;
       Buffer : out System.Address;
       Length : out POSIX.IO_Count);
private
   implementation-defined
end POSIX_IO;
```

6.1.1 Open or Create a File

6.1.1.1 Synopsis

```
-- File Descriptors
type File_Descriptor is range 0 .. POSIX.Open_Files_Maxima'Last - 1;
Standard_Input :  constant File_Descriptor := 0;
Standard_Output : constant File_Descriptor := 1;
Standard_Error :  constant File_Descriptor := 2;
type IO_Offset is implementation-defined-integer;
-- File modes and options
type File_Mode is (Read_Only, Write_Only, Read_Write);
type Open_Option_Set is new POSIX.Option_Set;
-- Empty˙Set, "+", and "-" are derived operations
Non_Blocking : constant Open_Option_Set := implementation-defined;
Append :       constant Open_Option_Set := implementation-defined;
Truncate :     constant Open_Option_Set := implementation-defined;
Exclusive :    constant Open_Option_Set := implementation-defined;
Not_Controlling_Terminal : constant Open_Option_Set := implementation-defined;
Signal_When_Socket_Ready : constant Open_Option_Set := implementation-defined;
File_Synchronized :
   constant POSIX_IO.Open_Option_Set := implementation-defined;
Data_Synchronized :
   constant POSIX_IO.Open_Option_Set := implementation-defined;
Read_Synchronized :
   constant POSIX_IO.Open_Option_Set := implementation-defined;
-- Operations to open or close file descriptors
function Open
   (Name :           POSIX.Pathname;
    Mode :           File_Mode;
    Options :        Open_Option_Set := Empty_Set;
    Masked_Signals : POSIX.Signal_Masking := POSIX.RTS_Signals)
   return File_Descriptor;
function Open_Or_Create
   (Name :        POSIX.Pathname;
    Mode :        File_Mode;
    Permissions : POSIX_Permissions.Permission_Set;
    Options :     Open_Option_Set := Empty_Set;
    Masked_Signals : POSIX.Signal_Masking := POSIX.RTS_Signals)
   return File_Descriptor;
function Is_Open (File : File_Descriptor)
   return Boolean;
procedure Close
   (File :           in File_Descriptor;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
function Duplicate
   (File :   File_Descriptor;
    Target : File_Descriptor := 0)
   return File_Descriptor;
function Duplicate_And_Close
   (File :   File_Descriptor;
    Target : File_Descriptor := 0;
    Masked_Signals : POSIX.Signal_Masking := POSIX.RTS_Signals)
   return File_Descriptor;
procedure Create_Pipe
   (Read_End :  out File_Descriptor;
    Write_End : out File_Descriptor);
```

6.1.1.2 Description

The type `File_Descriptor` shall be used to represent a file or device available for I/O operations within the system. `File_Descriptor` shall be an integer type whose range is constrained by the implementation-defined value `POSIX_Limits.Open_Files_Maxima'Last`. (See 2.6.1.) There are three predefined values of type `File_Descriptor`. These values represent files normally opened by the system before the process starts. The names of these values shall be as shown in Table 6.1:

The type `IO_Offset` shall be used to represent a location within a file, expressed in terms of bytes. This type shall contain both positive and negative values because it is possible to call `Seek`, or one of the `POSIX_File_Locking` subprograms, with a negative `IO_Offset` relative to the current file offset and because negative offsets are meaningful on some devices.

Table 6.1 – Standard File Descriptors

`Standard_Input`	`(File_Descriptor'(0))`
`Standard_Output`	`(File_Descriptor'(1))`
`Standard_Error`	`(File_Descriptor'(2))`

The type `File_Mode` shall enumerate the access mode of a file, indicating whether read and/or write operations are permitted on the file. The access modes are `Read_Only`, `Write_Only`, and `Read_Write`.

The type `Open_Option_Set` shall denote additional options on the file. The operations `"+"`, `"-"`, `">"`, `"<"`, `">="`, `"<="`, and `Empty_Set` are available on the type `Open_Option_Set` via the derived type semantics of Ada, from the operations available for `POSIX.Option_Set`. The user can create the option set by using the appropriate operations to create a set containing the options needed. In the following, the term *open operation* means one of the operations `Open` or `Open_Or_Create`.

The meanings of the options are as follows:

`Non_Blocking`

This option shall denote the behavior of the open operations on FIFO files, and on block special files or character special files that support nonblocking opens.

(1) For a FIFO whose file mode is `Read_Only` or `Write_Only`:

(a) When set, an open operation shall return without delay. An open operation with file mode of `Write_Only` shall return an error if no process currently has the file open for reading.

(b) When not set, an open operation with file mode of `Read_Only` shall block until a process opens the file for writing. An open operation with file mode of `Write_Only` shall block until a process opens the file for reading.

(2) For a block special file or character special file that supports nonblocking opens:

(a) When set, an open operation shall return without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

(b) When not set, an open operation shall wait until the device is ready or available before returning.

For cases not covered by the list above, the behavior of an open operation with the `Non_Blocking` option shall be undefined.

`Append`

This option shall denote where output shall be placed in the file. When set, the file offset in the open file description shall be reset to the end of the file prior to each write operation. When not set, the file offset shall be updated as described by the `Write`, `Generic_Write` (see 6.1.4 and `Seek` (see 6.1.5) operations.

`Exclusive`

This option shall specify that `Open_Or_Create` shall fail if the file named by the `Name` parameter exists. The check for existence and the creation of the file if it does not exist shall be a single atomic action, with respect to all other processes executing `Open_Or_Create` naming the same file within the same directory that are also using this option. If this option is specified for `Open`, the result is undefined.

`Truncate`

This option shall denote whether the file shall be truncated when opened. When set, the file shall be truncated to zero length when opened. The mode and owner are unchanged. If not set, truncation does not occur when the file is opened. This option shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation defined. The result of applying this option on a file opened with `Mode` specified as `Read_Only` is undefined.

`Not_Controlling_Terminal`

This option shall specify whether the file shall be adopted as the terminal controlling the process. (See 7.1.0.6.) If set and if the `Name` parameter identifies a terminal device, it is unspecified whether that terminal device becomes the controlling terminal for the process.

`Data_Synchronized`

Write operations on the file descriptor complete as defined by synchronized I/O data integrity completion (see 2.2.2.178).

`File_Synchronized`

Write operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see 2.2.2.179).

`Read_Synchronized`

Read operations complete at the same level of integrity as specified by `Data_Synchronized` and `File_Synchronized`. If both `Data_Synchronized` and `Read_Synchronized` are specified, all I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If both `File_Synchronized` and `Read_Synchronized` are specified, all I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

`Signal_When_Socket_Ready`

When input or output becomes possible on a socket, the process group is signaled via `Signal_IO`.

c

The `Open` operation shall open a file that already exists. The `Open_Or_Create` operation shall either open an existing file or create a new file with the given name if none exists. In both cases, the `Name` parameter specifies the name of the file to be opened or created. These operations shall create a new open file description, and shall return a value of type `File_Descriptor` referring to that open file description. The `File_Descriptor` value shall be the lowest file descriptor not currently open for that process. The file descriptor shall have the property that it is not closed by the system when one of the `Exec` family of operations is called. (See 6.1.7.) The open file description shall not be shared by any other process in the system. The file offset shall be set to the beginning of the file. The options in the open file description shall be the options provided by the `Options` parameter. The file access mode in the open file description shall be the value provided by the `Mode` parameter. The effect of calling `Open` or `Open_Or_Create` where `Name` specifies a FIFO special file and `Mode` specifies `Read_Write` is undefined. The parameter `Masked_Signals` specifies which signals shall be masked during the operation. (See 2.4.1.6.)

If `Open_Or_Create` creates a new file, the operation shall mark the Last Access Time, the Last Modification Time and the Last Status Change Time for update. If an existing file is opened and the `Truncate` option is requested, the operation shall mark the last modification time and the Last Status Change Time for update. (See 2.3.10.)

If the Sockets Detailed Network Interface option is supported: If `Name` specifies a pathname naming a socket, then it is implementation defined whether

— The operation shall fail with the error code set to `Socket_Type_Not_Supported`.

— The operation shall have the same effect as if the `POSIX_Sockets.Create` function were called with parameters appropriate to the given pathname and the `POSIX_Sockets.Connect` procedure was then called specifying the path as the destination.

If the XTI Detailed Network Interface option is supported: If the file has already been created and is now being opened; and if `Pathname` specifies a pathname naming a character special file for use with XTI calls, and the implementation supports such pathnames, then a communications endpoint shall be established as described in 17.4.19, but need not be synchronized with the data structures of the communications library or be in the Unbound state state (see 17.2). The application should call `POSIX_XTI.Synchronize_Endpoint` to synchronize such an endpoint with the data structures of the communications library and put it into the Unbound state. The application shall not designate a `File_Mode` other than `Read_Write` and shall not select an option other than `Non_Blocking` when calling the `Open` or `Open_Or_Create` operations with a pathname that names a character special file for use with XTI calls. Whether such pathnames are supported is implementation defined. Applications shall not specify a character special file for use with XTI calls when issuing an operation to create a file.

c

`Is_Open` shall indicate whether a file descriptor is open for I/O operations. This function shall return a `Boolean` value, but shall never raise an exception.

`Create_Pipe` shall create a pipe (a one-way communications channel). Upon return, the parameter `Read_End` shall be the `File_Descriptor` value associated with the pipe for `Read` operations. The parameter `Write_End` shall be the `File_Descriptor` value associated with the pipe for `Write` operations. The file descriptor values shall be the two lowest currently unopened file descriptor values for the process. The `Non_Blocking` option shall be set in both open file descriptions and both file descriptors shall have the property that it is not closed by the system when one of the `Exec` family of operations is called. (See 6.1.7.) Data can be written to the file descriptor `Read_End`, and read from the file descriptor `Write_End`. A read on the file descriptor `Read_End` shall access the data written to the file descriptor `Write_End` on a FIFO basis. A process has the pipe open for reading if it has a file descriptor open that refers to the open file description associated with the file descriptor `Read_End`. A process has the pipe open for writing if it has a file descriptor open that refers to the open file description associated wih the file descriptor `Write_End`. Upon successful completion, the operation shall mark for update the Last Access Time, Last Modification Time and Last Status Change Time for the pipe.

*If the Sockets Detailed Network Interface option is supported:*An implementation may provide `Create_Pipe` sockets. In this case, sockets-related errors may be raised to the calling application. C

`Close` shall free the file descriptor named by the parameter `File`. Upon normal return, the descriptor is closed and may be reused by the system for a subsequent open operation. When a file descriptor is closed, no further I/O operations may be performed on that file descriptor. All outstanding record locks owned by the process on the file shall be removed (that is, unlocked). (See 6.2.) If `Close` is interrupted by a signal, it shall fail, raising `POSIX_Error` with error code `Interrupted_Operation`, and the state of the file descriptor is unspecified. When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO shall be discarded. When all file descriptors associated with an open file description have been closed, the open file description shall be freed. When all file descriptors associated with the file have been closed, if the link count of the file is zero and the file is not a character special file for use with XTI calls the space occupied by the file shall be freed and the file shall no longer be accessible. (See 5.2.3.) If the link count of the file is zero and the file is a character special file for use with XTI calls, then the behavior of `Close` is unspecified. C C

If the Asynchronous I/O option is supported: When there is an outstanding cancelable AIO operation against `File` and `Close` is called, that I/O operation may be canceled. An I/O operation that is not canceled completes as if the `Close` operation had not yet occurred. All operations that are not canceled shall complete as if the `Close` blocked until the operations completed. The `Close` operation itself need not block awaiting such I/O completion. Whether any I/O operation is canceled, and which I/O operation may be canceled are implementation defined.

If the Memory Mapped Files option or the Shared Memory Objects option is supported: If a memory object remains referenced at the last close (*i.e.*, a process has it mapped), then the entire contents of the memory object shall persist until the memory object becomes unreferenced. If this close is the last close of a memory object, if the close results in the memory object becoming unreferenced, and if the memory object has been unlinked, then the memory object shall be removed.

When all descriptors associated with an open file description of a character special file for use with XTI calls have been closed, any connection that may be associated with that endpoint shall be broken. The connection may be terminated in an orderly or an abortive manner. Protocol-specific constraints on the behavior in this case shall be as described for `Close` in D.2.

When all descriptors associated with an open file description of a socket have been closed the following apply:

(a) If the socket is in the Listening state, all pending connections to the socket shall be aborted, whether or not the new connection was ready for return by the `Accept_Connection` operation. The effect of an abort in this state is protocol specific.

(b) If the socket is connection-mode, the socket is connected, and the linger option is not on, then the implementation shall initiate a normal disconnection. Whether a normal disconnection causes data that have not been delivered to be discarded is protocol-specific.

(c) If the socket is connection-mode, the socket is connected, and the linger option is on with a linger time of zero, then the implementation shall initiate an immediate disconnection. The action and consequences of an immediate disconnection are protocol-specific.

(d) If the socket is connection-mode, the socket is connected, and the linger option is on with a nonzero linger time then the following apply:

— The implementation shall initiate a normal disconnection. Whether a normal disconnection causes data that have not been delivered to be discarded is protocol specific.

— If the `Non_Blocking` option is not set for the descriptor, the `Close` operation shall block until the disconnection operation completes or until the linger time expires.

— If the `Non_Blocking` option is set for the descriptor, the operation shall not block.

(e) Whether attempts shall be made to deliver data that have not yet been delivered after the completion of the linger time or if the function does not block is protocol specific.

(f) All data in the receive queue of the socket shall be discarded.

(g) All other resources associated with the socket shall be freed.

(h) The `Close` operation shall deallocate the file descriptor. c

`Duplicate` shall return a new file descriptor that is the lowest numbered available file descriptor value not currently open by the process greater than or equal the value of the parameter `Target`. The new file descriptor shall refer to the same open file description as the file descriptor named by the parameter `File`. The returned file descriptor shall have the property that it is not closed by the system when one of the `Exec` family of operations is called. (See 6.1.7.)

`Duplicate_And_Close` shall behave as follows:

— If `File` is not open, the exception `POSIX_Error` shall be raised with error code `Bad_File_Descriptor`.

— If `File` is open and `Target` is equal to `File`, `File` shall be returned, and the file shall not be closed.

— If `File` is open, `Target` is not equal to `File`, and `Target` is open, `Target` shall be closed. Then the value that is returned shall equal `Target`. This file descriptor is opened and shall refer to the same open file description as the parameter `File`.

For example, if file descriptor 4 is open and `Duplicate_And_Close(4,7)` is called the following shall happen:

— If file descriptor 7 is open, it shall be closed.

— Then the value returned shall be 7, and file descriptors 4 and 7 are now open to the same open file description.

6.1.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

Upon `Open` or `Open_Or_Create`, search permission is denied on a component of the path prefix; or the file exists and the permissions specified by `Mode` are denied; or the file does not exist, and write permission is denied for the parent directory of the file to be created; or the `Truncate` option is set, and write permission is denied.

`Interrupted_Operation`

`Open`, `Open_Or_Create`, or `Close` was interrupted by a signal.

`Filename_Too_Long`

Upon `Open` or `Open_Or_Create`: The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`No_Such_File_Or_Directory`

Upon `Open`, the file does not exist.

`Not_A_Directory`

Upon `Open` or `Open_Or_Create`, a component of the parameter `Name` (except the last component) is not a directory.

`Is_A_Directory`

Upon `Open` or `Open_Or_Create`, the parameter `Name` denotes a directory, and the value of `Mode` is either `Read_Write` or `Write_Only`.

`Too_Many_Open_Files`

`Open`, `Open_Or_Create`, `Duplicate`, or `Create_Pipe` would cause the program to have more than Open Files Maximum open file descriptors.

`Too_Many_Open_Files_In_System`

Upon `Open`, `Open_Or_Create`, or `Create_Pipe`, a system-wide implementation-defined open file limit would be exceeded.

No_Space_Left_On_Device

Upon Open_Or_Create, the file is being created, and the device associated with the file named by the parameter Name is full.

No_Such_Device_Or_Address

Upon Open or Open_Or_Create, the requested mode is Write_Only for nonblocking I/O, and the file is a FIFO special file and no process has the FIFO open for reading.

Read_Only_File_System

Upon Open or Open_Or_Create, the file system is read-only and the value of parameter Mode is not Read_Only.

File_Exists

Upon Open_Or_Create, the value of the parameter Options has the option Exclusive set to True, and the file named by parameter Name already exists.

Bad_File_Descriptor

Upon Close, Duplicate, or Duplicate_And_Close, the file associated with the parameter File is not open.

No_Buffer_Space

The implementation attempted to create the pipe using sockets, but insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

Socket_Type_Not_Supported

The path specified is that of a Local_Protocol socket for which Open is not supported, and the implementation treats this condition as an error. c

Is_Open shall not raise any exceptions.

6.1.2 I/O Buffer Type

6.1.2.1 Synopsis

```
subtype IO_Buffer is POSIX.POSIX_String; -- obsolescent
```

6.1.2.2 Description

The subtype IO_Buffer is used to represent a sequence of POSIX_Character values for some I/O operations. An I/O operation transfers a number of POSIX_Character values to or from an IO_Buffer object, returning the number of POSIX_Character values transferred in an object of type IO_Count.

The type IO_Buffer is obsolescent. It is preserved for compatibility with POSIX.5, only. New applications should use Ada_Streams.Stream_Element_Array. (See 2.4.1.3.)

6.1.3 Read from a File

6.1.3.1 Synopsis

```
procedure Read                                  -- obsolescent
   (File :           in  File_Descriptor;
    Buffer :         out IO_Buffer;
    Last :           out POSIX.IO_Count;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Read
   (File :           in  File_Descriptor;
    Buffer :         out Ada_Streams.Stream_Element_Array;
    Last :           out Ada_Streams.Stream_Element_Offset;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
generic
   type T is private;
procedure Generic_Read
   (File :           in  File_Descriptor;
    Item :           out T;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
```

6.1.3.2 Description

The Read operation shall transfer a number of values from the file associated with the open file descriptor parameter File into the parameter Buffer. Read shall attempt to transfer enough data to completely fill the buffer, *i.e.*, Read shall attempt to read Buffer'Length values. On successful return, the parameter Last shall contain the index of the last element of Buffer supplied by the Read operation. If the parameter Buffer is a null array, then Last shall be Buffer'First - 1 and Read shall have no other result. Two overloadings of Read are defined, operating on arrays of POSIX.POSIX_Character and Ada_Streams.Stream_Element_Array.

The version of Read with argument of type IO_Buffer is obsolescent.

On a regular file or other file capable of seeking, Read shall start at the position in the file given by the file offset in the open file description associated with the file descriptor, File. Before a successful return from Read, the file offset shall be incremented by the number of bytes occupied by the values actually read. On a file not capable of seeking, Read shall start from the current position. The value of the file offset associated with such a file is undefined.

The number of values read may be less than the number of values requested (*i.e.*, the length of the Buffer parameter). The number may be less if the read request was interrupted by a signal or if the file descriptor refers to a pipe or FIFO or other special file that has less than the requested amount of data immediately available for reading. For example, a terminal device may return one typed line of data, where the number of values in the line is less than the length of the Buffer parameter. If the Read operation transfers all data, the value of Last shall be Buffer'Last.

If Read is interrupted by a signal before it reads any data, Read shall raise POSIX_Error with error code Interrupted_Operation. If Read is interrupted by a signal after it has successfully read some data, Read shall return the index of the last value transferred to the Buffer parameter. Read shall never raise POSIX_Error with error code Interrupted_Operation on a pipe or FIFO if the operation has transferred any data.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, Read shall raise IO_Exceptions.End_Error. If the parameter File refers to a device special file, the result of subsequent calls to Read is implementation defined.

When an attempt is made to read from an empty pipe or FIFO special file the following apply:

— If no process has the pipe open for writing Read shall raise IO_Exceptions.-End_Error.

— If some process has the pipe open for writing and the open file description associated with the parameter File has the Non_Blocking option, Read shall raise POSIX_Error with error code Resource_Temporarily_Unavailable.

— If some process has the pipe open for writing and the open file description associated with the parameter File does not have the Non_Blocking option, Read shall block until some data are written or the pipe is closed by all processes that had the pipe open for writing.

When an attempt is made to read a file that is other than a pipe or FIFO, supports nonblocking reads, and has no data currently available, the following apply:

— If the open file description associated with the parameter File has the Non_-Blocking option, Read shall raise POSIX_Error with error code Resource_-Temporarily_Unavailable.

— If the open file description associated with the parameter File does not have the Non_Blocking option, Read shall block until some data become available.

The Non_Blocking option shall have no effect if some data are available.

For any portion of a regular file, prior to end-of-file, that has not been written, Read shall place the value Stream_Element'(0) or POSIX_Character'val(0) into the appropriate position(s) in Buffer.

Upon successful completion, when Read successfully transfers any data, the Last Access Time shall be marked for update.

The limit Pipe Length Maximum represents the number of bytes guaranteed to be transferred during a single Read operation where the parameter File represents a pipe or a FIFO special file. An attempt to transfer quantities of values greater than this limit may be interleaved with other I/O operations on the file or device.

The Generic_Read operation is instantiated with a user-supplied type. This operation shall execute one or more Read operations to obtain sufficient bytes to complete the parameter Item. The number of values to be transferred shall be the size of the parameter Item in bytes. Generic_Read shall either return, having transferred the entire object into the parameter Item, or shall raise an exception if unable to obtain sufficient values to complete the object. If an exception is raised, the number of values actually transferred is unspecified, and there is no portable way for the application to know this number. The implementation of Generic_Read shall retry if the operation is blocked or is interrupted until either the entire object is transferred or an implementation-defined limit on retries is reached. The effect of instantiating

`Generic_Read` is unspecified if the size of the parameter `Item` is greater than Pipe Length Maximum and the external file is a pipe or FIFO special file.

For any given constrained type, instantiations of `Generic_Read` or `Generic_Write` shall be invertible in the following sense: writing an object of the type to a file and then reading the same file into a second object of the type shall result in both objects having the same value. Instantiations for unconstrained types may be refused by the implementation.

Upon successful completion, when `Generic_Read` successfully transfers any data, the Last Access Time shall be marked for update.

The following additional specifications apply if the Synchronized I/O option is supported, the following apply:

— If `Data_Synchronized` and `Read_Synchronized` have been specified, read operations on the file descriptor complete as defined by synchronized I/O data integrity completion (see 2.2.2.178).

— If `File_Synchronized` and `Read_Synchronized` have been specified, read operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see 2.2.2.179).

— If `Data_Synchronized`, `File_Synchronized`, and `Read_Synchronized` have been specified, read operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see 2.2.2.179).

If `File` refers to a shared memory object or any object other than a file, the result of `Read` or `Generic_Read` is unspecified.

The behavior of the `Read` operation for a character special file for use with XTI calls is unspecified. On a socket, the call to `Read` shall be equivalent to the call to `Receive` with `Options` equal to the `Empty_Set`. c

6.1.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`

The file descriptor `File` is not open.

The open file description associated with the file descriptor `File` is not open for reading.

`Interrupted_Operation`

The operation was interrupted by a signal.

`Input_Output_Error`

The implementation supports job control, a process in a background process group attempts to read from its controlling terminal, and either the process group of the process is orphaned or the process ignores the signal `POSIX_Signals.Signal_Terminal_Input`.

Resource_Temporarily_Unavailable

The Non_Blocking option is set and no data are immediately available to be read. In this case, the operation shall fail rather than blocking.

IO_Exceptions.End_Error shall be raised by Read when an attempt is made to read and no more values are left at the end of a file or by Generic_Read when the number of bytes in the file is smaller than the size of the Item measured in bytes.

6.1.4 Write to a File

6.1.4.1 Synopsis

```
procedure Write                              -- obsolescent
   (File :           in  File_Descriptor;
    Buffer :         in  IO_Buffer;
    Last :           out POSIX.IO_Count;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Write
   (File :           in  File_Descriptor;
    Buffer :         in  Ada_Streams.Stream_Element_Array;
    Last :           out Ada_Streams.Stream_Element_Offset;
    Masked_Signals : in  POSIX.Signal_Masking := POSIX.RTS_Signals);
generic
   type T is private;
procedure Generic_Write
   (File :           in File_Descriptor;
    Item :           in T;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

6.1.4.2 Description

The Write operation shall transfer a number of values from the parameter Buffer into the file associated with the open file descriptor parameter File. Write shall attempt to transfer the complete contents of the buffer, *i.e.*, Write shall attempt to transfer Buffer'Length values. On successful return, the parameter Last shall contain the index of the last element of Buffer transferred by the Write operation. If the Write operation transfers all data, the value of Last shall be Buffer'Last. If the parameter Buffer is a null array, then Last shall be 0 and Write shall have no result. Two overloadings of Write are defined, operating on arrays of POSIX.-POSIX_Character and Ada_Streams.Stream_Element_Array.

The version of Write with parameter of type IO_Buffer is obsolescent.

On a regular file or other file capable of seeking, Write shall start at the position in the file given by the file offset in the open file description associated with the file descriptor given by parameter File. Before a successful return from Write, the file offset shall be incremented by the number of bytes occupied by the values actually written. On a regular file, if the resulting file offset value is greater than the length of the file in bytes, the length of the file shall be set to this file offset value. On a file not capable of seeking, Write shall start from the current position. The value of a file offset associated with such a file is undefined.

If the Append option is set in the open file description associated with the file descriptor File, then the file offset shall be set to the end of the file prior to each write,

and no intervening file modification operation shall be allowed between the change to the file offset and the associated write operation.

If the `Write` operation requests that more values be written than there is room to accommodate (for example, past the physical end of the medium), the `Write` operation shall transfer only as many values as can be written. In this case, the `Write` operation shall return the index of the last element of `Buffer` written in the `Last` parameter. For instance, if `Buffer'Size` is 100 (and `Buffer'First` is 1) and there is room on the device for only 20 values, then `Write` shall transfer 20 values, and return the value 20 for the parameter `Last`. The next write request (for a nonzero number of values) would then fail (except as noted before).

If `Write` is interrupted by a signal before it writes any data, `Write` shall raise `POSIX_Error` with error code `Interrupted_Operation`. If `Write` is interrupted by a signal after it has successfully written some data, either it shall raise `POSIX_Error` with error code `Interrupted_Operation` or it shall return the index of the last value transferred from the `Buffer` parameter. `Write` shall never raise `POSIX_Error` with error code `Interrupted_Operation` on a pipe or FIFO if the operation has transferred any data.

After the `Write` operation to a regular file has returned,

— Any successful `Read` operation from each position in the file that was modified by that `Write` operation shall return the data specified by the `Write` operation for that position until such positions are again modified by another `Write` operation.

— Any subsequent successful `Write` to the same position in the file shall overwrite the previous file data. (The phrase "subsequent successful `Write` operation" in the previous sentence is intended to be viewed from a system perspective, *i.e.*, a `Read` operation followed by a system-wide subsequent `Write` operation.)

`Write` operations to a pipe or FIFO special file shall be handled in the same manner as `Write` operations on a regular file, with the following exceptions:

(1) There is no file offset associated with a pipe; hence each `Write` operation shall append to the end of the pipe.

(2) `Write` operations that transfer amounts of data less than Pipe Length Maximum bytes shall not be interleaved with other processes performing `Write` operations on the same pipe. `Write` operations that transfer more than Pipe Length Maximum bytes may have data interleaved, on arbitrary boundaries, with `Write` operations by other processes, whether or not the open file description for the pipe has the `Non_Blocking` option.

(3) If the open file description associated with the pipe does not have the `Non_Blocking` option, then the `Write` operation may block. On normal completion of the `Write` operation, it shall set the parameter `Last` to be `Buffer'Last`.

(4) If the open file description associated with the pipe has the `Non_Blocking` option, then `Write` operations shall be handled differently, as described below:

  (a) The `Write` operation shall not block.

  (b) A `Write` operation where `Buffer'Length` is less than or equal to Pipe Length Maximum shall do one of the following:

(i) If there is sufficient space available in the pipe, the operation shall transfer all of the data and return the value `Buffer'Last` in the parameter `Last`.

(ii) If there is not sufficient space available in the pipe, the operation shall transfer no data and raise `POSIX_Error` with error code `Resource_Temporarily_Unavailable`.

(c) A `Write` operation where `Buffer'Length` is greater than Pipe Length Maximum shall do one of the following:

(i) If at least one value can be transferred, the `Write` operation shall transfer as many values as possible and return the index of the last element of `Buffer` transferred in the parameter `Last`. When all data previously written to the pipe have been read, it shall transfer at least Pipe Length Maximum bytes.

(ii) If no data can be written, the `Write` operation shall transfer no data and shall raise `POSIX_Error` with error code `Resource_Temporarily_Unavailable`.

When an attempt is made to write to a file or device that is other than a pipe or FIFO, is associated with the file descriptor `File`, and supports nonblocking writes, but cannot accept the data immediately, the following apply:

— If the open file description associated with the file descriptor `File` does not have the `Non_Blocking` option, the `Write` operation shall block until the data can be accepted or written.

— If the open file description associated with the file descriptor `File` has the `Non_Blocking` option, the `Write` operation shall not block. If some data can be written without blocking, the `Write` operation shall transfer as much data as it can without blocking, and the index of the last value from `Buffer` that was transferred shall be returned in the parameter `Last`. Otherwise, the `Write` operation shall not transfer any data and shall raise `POSIX_Error`, with error code `Resource_Temporarily_Unavailable`.

Upon successful completion, when `Write` successfully transfers any data, the Last Access Time and Last Modification Time shall be marked for update.

The `Generic_Write` operation is instantiated with a user-supplied type. This operation shall execute one or more `Write` operations to transfer sufficient bytes to transfer the parameter `Item`. The number of values to be transferred shall be the size of the parameter `Item` in bytes. `Generic_Write` shall either return, having transferred the entire object from the parameter `Item`, or shall raise an exception if unable to transfer sufficient values to complete the object. If an exception is raised, the number of values actually transferred is unspecified, and there is no portable way for the application to know this number. The implementation of `Generic_Write` shall retry if the operation is blocked or is interrupted until either the entire object is transferred or an implementation-defined limit on retries is reached. The effect of instantiating `Generic_Write` is unspecified if the size in bytes of the parameter `Item` is greater than the limit returned by Pipe Length Maximum and the external file is a pipe or FIFO special file.

For any given constrained type, instantiations of `Generic_Read` or `Generic_Write` shall be invertible in the following sense: writing an object of the type to a file and

then reading the same file into a second object of the type shall result in both objects having the same value. Instantiations for unconstrained types may be refused by the implementation.

Upon successful completion, when `Generic_Write` successfully transfers any data, the Last Access Time and Last Modification Time shall be marked for update.

The following additional specifications apply if the Synchronized I/O option is supported, the following apply:

— If `Data_Synchronized` has been specified, write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion (see 2.2.2.178).

— If `File_Synchronized` has been specified, write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see 2.2.2.179).

— If `Data_Synchronized` and `File_Synchronized` have been specified, write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see 2.2.2.179).

— If `Data_Synchronized` and `Read_Synchronized` have been specified, write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion (see 2.2.2.178).

— If `File_Synchronized` and `Read_Synchronized` have been specified, write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see 2.2.2.179).

— If `Data_Synchronized`, `File_Synchronized`, and `Read_Synchronized` have been specified, write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see 2.2.2.179).

If `File` refers to a shared memory object or any object other than a file, the result of `Write` or `Generic_Write` is unspecified.

The behavior of the `Write` operation for a character special file for use with XTI calls is unspecified. On a socket, the call to `Write` shall be equivalent to the call to `Send` with `Options` equal to the `Empty_Set`. C

6.1.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`

The file descriptor `File` is not open.

The open file description associated with the file descriptor `File` is not open for writing.

`Input_Output_Error`

The implementation supports job control, a process in a background process group attempts to write to its controlling terminal, and either the process

group of the process is orphaned or the process ignores the signal POSIX_-Signals.Signal_Terminal_Output.

A write was attempted to a terminal device and a modem disconnect was detected by the terminal interface for that device. (See 7.1.0.13.)

Interrupted_Operation

The operation was interrupted by a signal.

Resource_Temporarily_Unavailable

If the Non_Blocking option is set in the open file description associated with the file descriptor File and the task would be delayed in completing the operation, the operation shall fail rather than blocking.

Broken_Pipe

The parameter File is associated with a pipe or FIFO special file whose read end is not open.

File_Too_Large

The file size exceeds a system limit.

No_Space_Left_On_Device

No space remains on the device associated with the parameter File.

6.1.5 File Position Operations

6.1.5.1 Synopsis

```
type Position is
   (From_Beginning, From_Current_Position, From_End_Of_File);
procedure Seek
   (File :           in  File_Descriptor;
    Offset :         in  IO_Offset;
    Result :         out IO_Offset;
    Starting_Point : in  Position := From_Beginning);
function File_Size (File : File_Descriptor)
   return POSIX.IO_Count;
function File_Position (File : File_Descriptor)
   return IO_Offset;
```

6.1.5.2 Description

For files and devices where a file offset exists, the system shall maintain the current file position in the open file description associated with the file or device. The current file position can be changed by calls to Read or Write operations or can be directly manipulated by Seek.

Seek shall modify the current file position in the open file description associated with the file descriptor File. The interpretation of the parameter Offset shall depend on the value of the parameter Starting_Point as follows:

— If the value of Starting_Point is From_Beginning, the file position shall be set to the value of Offset bytes from the beginning of the file.

— If the value of Starting_Point is From_Current_Position, the file position shall be set to the arithmetic sum of the current value of the file position and the value of Offset. A negative value of Offset has the effect of adjusting the file position backwards from its current position.

— If the value of `Starting_Point` is `From_End_Of_File`, the file position shall be set to the arithmetic sum of the size of the file in bytes plus the value of `Offset`. A negative value of `Offset` has the effect of adjusting the read/write pointer backwards from the end of the file.

It is not an error to seek past the end of a file. If a program seeks past the end of the file, the next `Read` or `Generic_Read` operation shall find end-of-file, and the next `Write` or `Generic_Write` operation shall extend the file.

`File_Size` shall return the size of the file associated with the file descriptor `File` in terms of bytes. `File_Position` shall return the current file position, in terms of bytes, from the open file descriptor associated with the file descriptor `File`.

Some devices are incapable of seeking. The value of the file offset in the open file description associated with the file descriptor `File` is undefined. The behavior of `Seek`, `File_Position`, or `File_Size` operations on such devices is implementation defined.

The result of `Seek`, `File_Position`, or `File_Size` on shared memory objects, or on a character special file for use with XTI calls, or on a socket, or on any object other than a file is unspecified. c

6.1.5.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: The parameter `File` is not open.

`Invalid_Seek`
: The file associated with the parameter `File` is a pipe or a FIFO special file.

6.1.6 Terminal Operations

6.1.6.1 Synopsis

```
function Is_A_Terminal (File : File_Descriptor)
   return Boolean;
function Get_Terminal_Name (File : File_Descriptor)
   return POSIX.Pathname;
```

6.1.6.2 Description

`Is_A_Terminal` shall return `True` if the external file associated with the parameter `File` is a terminal device and `False` otherwise.

`Get_Terminal_Name` shall return a value of type `Pathname` that represents the name of the external terminal device associated with the parameter `File`.

The null string shall be returned if the value of the parameter `File` is not a valid file descriptor associated with a terminal or if the pathname cannot be determined.

6.1.6.3 Error Handling

This standard does not specify any error conditions that are required to be detected for these operations. Errors may be reported under conditions that are unspecified by this standard.

6.1.7 File Control

6.1.7.1 Synopsis

```
procedure Get_File_Control
   (File :    in  File_Descriptor;
    Mode :    out File_Mode;
    Options : out Open_Option_Set);
procedure Set_File_Control
   (File :    in File_Descriptor;
    Options : in Open_Option_Set);
function Get_Close_On_Exec (File : File_Descriptor)
     return Boolean;
procedure Set_Close_On_Exec
   (File : in File_Descriptor;
    To :   in Boolean := True);
```

6.1.7.2 Description

The type `Open_Option_Set` shall be used to contain file options, as described in 6.1.1. Only the options `Append` and `Non_Blocking` are defined for use with the operations `Get_File_Control` and `Set_File_Control`.

`Get_File_Control` shall obtain the file mode and file option set from the open file description associated with parameter `File`. The value of the file mode shall be returned in parameter `Mode`, and the value of the file option set shall be placed into parameter `Options`.

The values of file options other than `Append` or `Non_Blocking` in the parameter `Options` are implementation defined.

`Set_File_Control` shall update the option set in the open file description associated with the file descriptor `File`. File options defined by this standard for use with `Set_File_Control` shall be updated. Other file options defined by this standard that are set in the parameter `Options` shall be ignored. The effect of options not defined by this standard is unspecified.

If the Shared Memory Objects option is supported: If `File` refers to a shared memory object or any object that other than a file, the effect of `Set_File_Control` is unspecified.

When a program executes one of the `Exec` family of operations, files may be closed across the call or may remain open. Each open file descriptor shall have associated with it information indicating whether the file descriptor should be closed by the `Exec`-family operation.

The value of this information shall be queried by `Get_Close_On_Exec` for the given file descriptor `File`, and `Set_Close_On_Exec` shall set this value for the given file descriptor `File`.

The application shall not designate a File_Mode other than Read_Write and shall not select an option other than Non_Blocking when calling the Get_File_Control and Set_File_Control operations with a pathname that names a character special file for use with XTI calls. c

6.1.7.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor

Upon Get_File_Control, Set_File_Control, Get_Close_On_Exec, or Set_Close_On_Exec, the parameter File is not open.

Invalid_Argument

This implementation does not support synchronized I/O for this file.

6.1.8 Update File Status Information

6.1.8.1 Synopsis

```
procedure Change_Permissions
   (File :       in POSIX_IO.File_Descriptor;
    Permission : in POSIX_Permissions.Permission_Set);
```

6.1.8.2 Description

The functionality described in this subclause is optional. If neither the Memory Mapped Files option nor the Shared Memory Objects option is supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

NOTE: It is expected that a future revision will make this interface mandatory for conforming implementations and regular files. The dependence of this entire subclause on the Memory Mapped Files option and the Shared Memory Objects option will be removed at the time the contemplated revision is approved. Since the existence of shared memory will remain optional, the effects of this operation on shared memory objects will remain conditional.

Change_Permissions shall change the permission set associated with the object referred to by File. If the effective user ID of the calling process matches the file owner or the calling process has appropriate privileges, Change_Permissions shall change the permissions of the object specified by File to those specified in Permission. Additional implementation-defined restrictions may cause Set_Group_ID and Set_User_ID in Permission to be ignored.

If the Shared Memory Objects option is supported: If File is a shared memory object, Change_Permissions need only affect the Owner_Read, Owner_Write, Group_Read, Group_Write, Other_Read, and Other_Write permissions; other permissions specified in the Permissions parameter may be ignored.

If the calling process does not have appropriate privileges, if the group ID of the file does not match the effective group ID or one of the supplementary group IDs, if one or more of Owner_Execute, Group_Execute, and if Others_Execute are specified

in Permission, and the file is a regular file, Set_Group_ID in the permission set of the object specified by File shall be cleared upon successful return from Change_-Permissions.

The effect on file descriptions for files open at the time of the Change_Permissions is implementation defined.

The Last Status Change Time of the file shall be marked for update upon completion of this operation. (See 2.3.10.)

6.1.8.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor
: The File parameter is invalid.

Operation_Not_Implemented
: This implementation does not support the Change_Permissions procedure.

Operation_Not_Permitted
: The effective user ID does not match the owner of the file, and the calling process does not have the appropriate privileges.

Read_Only_File_System
: The file resides on a read-only file system.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: File refers to a pipe, and the implementation does not support this operation for pipes.

6.1.9 Truncate File to A Specified Length

6.1.9.1 Synopsis

```
procedure Truncate_File
   (File :   in POSIX_IO.File_Descriptor;
    Length : in POSIX.IO_Count);
```

6.1.9.2 Description

The functionality described in this subclause is optional. If neither the Memory Mapped Files option nor the Shared Memory Objects option is supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

NOTE: It is expected that a future revision will make this interface mandatory for conforming implementations and regular files. The dependence of this entire subclause on the Memory Mapped Files option and the Shared Memory Objects option will be removed at the time the contemplated revision is approved. Since the existence of shared memory will remain optional, the effects of this operation on shared memory objects will remain conditional.

If `File` refers to a regular file open for writing, `Truncate_File` shall cause the file to be truncated to `Length` bytes. If the length of the file previously exceeded `Length` bytes, the extra data shall be discarded. If the file previously was smaller than this length, it is unspecified whether the file is changed or its length increased. If the file is extended, the extended area shall appear as if it were zero-filled.

If `File` refers to a shared memory object, `Truncate_File` sets the length of the shared memory object to `Length` bytes.

If the file does not refer to a regular file or a shared memory object, the result is unspecified.

If the effect of `Truncate_File` is to decrease the length of a file or shared memory object and if whole pages beyond the new end were previously mapped, then the whole pages beyond the new end shall be discarded, and references to them shall result in generation of the signal `Signal_Bus_Error`, which the Ada language implementation shall convert to the exception `Program_Error`.

The value of the seek pointer shall not be modified by a call to `Truncate_File`.

Upon successful completion, the `Truncate_File` procedure shall update the Last Status Change Time and Last Modification Time of the file. If the `Truncate_File` procedure is unsuccessful, the file is unaffected.

6.1.9.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: The `File` parameter is not a valid file descriptor open for writing.

`Invalid_Argument`
: `File` does not refer to a file on which this operation is possible.

`Read_Only_File_System`
: The file resides on a read-only file system.

`Operation_Not_Implemented`
: `Truncate_File` is not supported by this implementation.

NOTE: The last error code is not specified by POSIX.1.

6.1.10 Synchronize the State of a File

6.1.10.1 Synopsis

```
procedure Synchronize_File (File : in POSIX_IO.File_Descriptor);
```

6.1.10.2 Description

The functionality described in this subclause is optional. If the File Synchronization option is not supported, the implementation may cause all calls to the explicitly declared

operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

NOTE: It is expected that a future revision will make this interface mandatory for conforming implementations and regular files. The text of the support condition will be removed at the time the contemplated revision is approved. Since the Synchronized I/O option will remain optional, the semantics relating to it will remain conditional.

`Synchronize_File` can be used by the application to indicate that all data for the open file descriptor `File` is to be transferred to the storage device associated with the file specified by `File` in an implementation defined manner. `Synchronize_File` shall not return until the system has completed that action or until an error is detected.

The conformance document shall include sufficient information for the user to determine whether it is possible to configure an application and installation to ensure that the data is stored with the degree of required stability for the intended use. The hardware characteristics upon which the implementation relies to assure that data are successfully transferred are implementation defined.

If the Synchronized I/O option is supported: `Synchronize_File` forces all currently queued I/O operations associated with the file indicated by `File` to the synchronized I/O completion state (see 2.2.2.177). All I/O operations shall be completed as defined for synchronized I/O file integrity completion (see 2.2.2.179).

6.1.10.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: `File` is not a valid file descriptor.

`Invalid_Argument`
: The implementation does not support synchronized I/O for this file.

`Operation_Not_Implemented`
: `Synchronize_File` is not supported by this implementation.

If any of the queued I/O operations fails `Synchronize_File` shall return the error conditions defined for `POSIX_IO.Read` and `POSIX_IO.Write`.

6.1.11 Data Synchronization

6.1.11.1 Synopsis

```
procedure Synchronize_Data (File : in POSIX_IO.File_Descriptor);
```

6.1.11.2 Description

The functionality described in this subclause is optional. If the Synchronized I/O option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

`Synchronize_Data` forces all currently queued I/O operations associated with the file indicated by `File` to the synchronized I/O completion state (see 2.2.2.177).

The functionality is as described for `Synchronize_File` (See 6.1.10) with the exception that all I/O operations shall be completed as defined for synchronized I/O data integrity completion (see 2.2.2.178).

The hardware characteristics upon which the implementation relies to assure that data are successfully transferred are implementation defined.

6.1.11.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: `File` is not a valid file descriptor open for writing.

`Invalid_Argument`
: This implementation does not support synchronized I/O for this file.

`Operation_Not_Implemented`
: `Synchronize_Data` is not supported by this implementation.

If any of the queued I/O operations fails, `Synchronize_Data` shall return the error conditions defined for `POSIX_IO.Read` and `POSIX_IO.Write`.

6.1.12 Socket File Ownership

6.1.12.1 Synopsis

```
procedure Get_Owner
   (File    : in  File_Descriptor;
    Process : out POSIX_Process_Identification.Process_ID;
    Group   : out POSIX_Process_Identification.Process_Group_ID);
procedure Set_Socket_Process_Owner
   (File    : in File_Descriptor;
    Process : in POSIX_Process_Identification.Process_ID);
procedure Set_Socket_Group_Owner
   (File    : in File_Descriptor;
    Group   : in POSIX_Process_Identification.Process_Group_ID);
```

c

6.1.12.2 Description

`Get_Owner` gets the process or the process group owner for a socket. The semantics for all other types of file descriptors are undefined.

`Set_Socket_Process_Owner` sets the process owner for a socket. `Set_Socket_Group_Owner` sets the process group owner for a socket. It is implementation defined whether a process group ID may be specified. These operations apply only to socket type file descriptors. The semantics for all other types of file descriptors are undefined.

c

6.1.12.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: The `File` parameter is not a valid file descriptor.

`Invalid_Argument`
: `File` does not refer to a file on which this operation is possible.

`Operation_Not_Implemented`
: The operation is not supported by this implementation.

c

6.1.13 I/O Vector Type

6.1.13.1 Synopsis

```
type IO_Vector is limited private;
procedure Set_Buffer
   (Vector : in out IO_Vector;
    Buffer : in     System.Address;
    Length : in     POSIX.IO_Count);
procedure Get_Buffer
   (Vector : in  IO_Vector;
    Buffer : out System.Address;
    Length : out POSIX.IO_Count);
```

c

6.1.13.2 Description

The type `IO_Vector` is used to represent a buffer address and length for use with network I/O operations (see 18.4.1.3 and 17.4.1.9). Other operations that may use this object are implementation defined. An I/O vector has (at least) the following attributes:

Buffer
: The address of the buffer into which the data are to be read, or from which the data are to be written.

  NOTE: Buffer is a persistent reference to storage that may persist beyond an I/O operation. The effect of deallocating or modifying the contents of the buffers in an I/O vector before the I/O operation has completed is undefined.

  NOTE: Applications may use pointers to the `POSIX.Octet_Array` type to ensure proper data width for network I/O operations. When using pointers to other data types, byte width and ordering issues (i.e., big endian, little endian) are the responsibility of the application.

Length
: The length of the data to be transferred, as a count of octets.

The `Set_Buffer` procedure shall set the Buffer attribute of the `IO_Vector` to point to the storage indicated by the `Buffer` parameter and the Length attribute of the `IO_Vector` to `Length` (in octets). An `IO_Vector` object may be reset to indicate no data by either a `Length` of zero or the `Buffer` parameter set to `System.Null_Address`. The `Length` attribute may be set to less than the size of the storage allocated for Buffer. The `Get_Buffer` procedure shall return the Buffer and Length attributes of the `IO_Vector`.

c

6.2 Package POSIX_File_Locking

This package contains advisory file record locking operations. File locks are advisory, in that locking a file does not prevent other operations on the file. To determine whether a file is locked before performing another I/O operation, the appropriate POSIX_File_Locking operation should be called.

File locks are a property of the external file.

```
with POSIX,
     POSIX_IO,
     POSIX_Process_Identification;
package POSIX_File_Locking is
    -- Lock and Unlock a Region of a File
   type Lock_Kind is (Read_Lock, Write_Lock, Unlock);
   type File_Lock (Whole_File : Boolean := True) is
     record
       Lock : Lock_Kind;
       case Whole_File is
         when True => null;
         when False =>
           Starting_Point : POSIX_IO.Position;
           Start :          POSIX_IO.IO_Offset;
           Length :         POSIX.IO_Count;
       end case;
     end record;
   procedure Get_Lock
      (File :    in  POSIX_IO.File_descriptor;
       Lock :    in  File_Lock;
       Result :  out File_Lock;
       Process : out POSIX_Process_Identification.Process_ID);
   procedure Set_Lock
      (File : in POSIX_IO.File_Descriptor;
       Lock : in File_Lock);
   procedure Wait_To_Set_Lock
      (File :           in POSIX_IO.File_Descriptor;
       Lock :           in File_Lock;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);

end POSIX_File_Locking;
```

6.2.1 Lock and Unlock a Region of a File

6.2.1.1 Description

The type Lock_Kind shall be used to indicate the kind of lock or lock operation requested. Values for this type are

Read_Lock
: The file is locked for multiple (shared) readers. Write locks are prohibited.

Write_Lock
: The file is locked for a single (exclusive) writer. Other read and write locks are prohibited.

Unlock
: The file is unlocked.

The type File_Lock shall represent the lock obtained on a file. Objects of this type shall include information to show the kind of lock, the part of the file locked (whole file or a range of bytes), and the Process_ID of the process holding the lock.

A lock shall extend for the whole file (if Whole_File is True) or it shall start at offset Start in the file and continue for Length bytes (if Whole_File is False). Start is interpreted based on the value of Starting_Point. If Starting_Point has the value From_Beginning, Start identifies a location measured from the beginning of the file. If Starting_Point has the value From_Current_Position, Start identifies a location measured from the current file position associated with the file. If Starting_Point has the value From_End_Of_File, Start identifies a location measured from the end of the file.

Get_Lock shall obtain information on the first lock associated with the parameter File that overlaps the parameter Lock. If no process holds an overlapping lock on the file, the value of the parameter Process shall be POSIX_Process_Identification.Null_Process_ID; otherwise, the value of the parameter Process shall be the Process_ID of the process holding the lock. Upon return, if an overlapping lock was found, its value is copied onto Result, and Starting_Point is From_Beginning.

Set_Lock shall obtain a lock on the file associated with the parameter File, using the information provided in the parameter Lock to identify the part of the file to be locked. When Set_Lock returns, the file has been locked. If the lock cannot be immediately granted, Set_Lock shall raise POSIX_Error. (See 6.2.1.2.)

Set_Lock with Lock_Kind equal to Unlock unlocks a file or region. Unlocking a region leaves the specified region unlocked, but leaves lock(s) on the other part(s) of the file that were previously locked. Unlocking a region that was not previously locked is not an error.

Wait_To_Set_Lock shall obtain a file lock on the file associated with the parameter File, using the information provided in the parameter Lock to identify the part of file to be locked. When Wait_To_Set_Lock returns, the file has been locked. This procedure shall block the caller until the lock can be granted. The caller that will be blocked may be the whole Ada program or the calling Ada task, depending on the value of POSIX.File_Lock_Blocking_Behavior. The parameter Masked_Signals specifies which signals shall be masked during the operation. (See 2.4.1.6.)

6.2.1.2 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor

> Upon Get_Lock, Set_Lock, or Wait_To_Set_Lock, the parameter File is not open. This error code shall also be set by Set_Lock or Wait_To_Set_Lock if the parameter Lock is Read_Lock and the parameter File is not open for a mode that supports reading, or if the parameter Lock is Write_Lock and the parameter File is not open for a mode that supports writing.

`Interrupted_Operation`

`Wait_To_Set_Lock` was interrupted by a signal.

`Invalid_Argument`

Upon `Get_Lock`, `Set_Lock`, or `Wait_To_Set_Lock`, the file associated with the parameter `File` does not support locking (*e.g.*, the file is a FIFO special file).

`Resource_Deadlock_Avoided`

Upon `Set_Lock` or `Wait_To_Set_Lock`, the system detects that a deadlock would occur.

`Resource_Temporarily_Unavailable`

Upon `Set_Lock`, the portion of the file associated with the parameter `File` has a conflicting lock.

`No_Locks_Available`

Upon `Set_Lock` or `Wait_To_Set_Lock`, a system limit on the number of locks is exceeded.

6.3 Package POSIX_Asynchronous_IO

This package provides access to services for asynchronous input and output.

The functionality described in this clause is optional. If the Asynchronous I/O option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this clause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this clause.

```
with Ada_Streams,
     POSIX,
     POSIX_IO,
     POSIX_Signals;
package POSIX_Asynchronous_IO is
    -- 6.3.1 AIO Descriptor Type
   type AIO_Descriptor is private;
   function Create_AIO_Control_Block
     return AIO_Descriptor;
   procedure Destroy_AIO_Control_Block
      (AD : in out AIO_Descriptor);
    -- 6.3.2 Attributes of AIO Control Blocks
   type List_IO_Operations is
      (No_Op,
       Read,
       Write);
   type IO_Array_Pointer is access Ada_Streams.Stream_Element_Array;
   function Get_File (AD : AIO_Descriptor)
     return POSIX_IO.File_Descriptor;
   procedure Set_File
      (AD :   in AIO_Descriptor;
       File : in POSIX_IO.File_Descriptor);
   function Get_Offset (AD : AIO_Descriptor) return POSIX_IO.IO_Offset;
   procedure Set_Offset
      (AD :     in AIO_Descriptor;
       Offset : in POSIX_IO.IO_Offset);
```

```
function Get_Buffer (AD : AIO_Descriptor) return IO_Array_Pointer;
procedure Set_Buffer
   (AD :     in AIO_Descriptor;
    Buffer : in IO_Array_Pointer);
function Get_Length (AD : AIO_Descriptor)
  return Ada_Streams.Stream_Element_Count;
procedure Set_Length
   (AD :     in AIO_Descriptor;
    Length : in Ada_Streams.Stream_Element_Count);
function Get_Priority_Reduction (AD : AIO_Descriptor) return Natural;
procedure Set_Priority_Reduction
   (AD :                 in AIO_Descriptor;
    Priority_Reduction : in Natural);
function Get_Event (AD : AIO_Descriptor)
  return POSIX_Signals.Signal_Event;
procedure Set_Event
   (AD :    in AIO_Descriptor;
    Event : in POSIX_Signals.Signal_Event);
function Get_Operation (AD : AIO_Descriptor)
  return List_IO_Operations;
procedure Set_Operation
   (AD :        in AIO_Descriptor;
    Operation : in List_IO_Operations);
-- 6.3.3 Asynchronous Read
procedure Read (AD : in AIO_Descriptor);
-- 6.3.4 Asynchronous Write
procedure Write (AD : in AIO_Descriptor);
-- 6.3.5 List Directed I/O
type AIO_Descriptor_List is
   array (Positive range <>) of AIO_Descriptor;
procedure List_IO_No_Wait
   (List :  in out AIO_Descriptor_List;
    Event : in POSIX_Signals.Signal_Event);
procedure List_IO_Wait
   (List :           in out AIO_Descriptor_List;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
-- 6.3.6 Retrieve Status of AIO Request
type AIO_Status is
   (In_Progress,
    Completed_Successfully,
    Canceled);
function Get_AIO_Status
   (AD : AIO_Descriptor)
  return AIO_Status;
function Get_AIO_Error_Code
   (AD : AIO_Descriptor)
  return POSIX.Error_Code;
-- 6.3.7 Retrieve Bytes Transferred by AIO Request
function Get_Bytes_Transferred
   (AD : AIO_Descriptor) return Ada_Streams.Stream_Element_Count;
-- 6.3.8 Cancel AIO Request
type Cancelation_Status is
   (Canceled,
    Not_Canceled,
    All_Done);
function Cancel
   (AD : AIO_Descriptor)
  return Cancelation_Status;
```

```
   function Cancel
      (File : POSIX_IO.File_Descriptor)
     return Cancelation_Status;
   -- 6.3.9 Wait for AIO Request to Complete
   procedure Await_IO_Or_Timeout
      (AD :             in AIO_Descriptor;
       Timeout :        in POSIX.Timespec;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
   procedure Await_IO
      (AD :             in AIO_Descriptor;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
   procedure Await_IO_Or_Timeout
      (List :           in AIO_Descriptor_List;
       Timeout :        in POSIX.Timespec;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
   procedure Await_IO
      (List :           in AIO_Descriptor_List;
       Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
   -- 6.3.10 Asynchronous File and Data Synchronization
   procedure Synchronize_File (AD : in AIO_Descriptor);
   procedure Synchronize_Data (AD : in AIO_Descriptor);
private
   implementation-defined
end POSIX_Asynchronous_IO;
```

6.3.1 AIO Descriptor Type

6.3.1.1 Synopsis

```
type AIO_Descriptor is private;
function Create_AIO_Control_Block
  return AIO_Descriptor;
procedure Destroy_AIO_Control_Block
   (AD : in out AIO_Descriptor);
```

6.3.1.2 Description

Values of the type `AIO_Descriptor` are used as AIO descriptors. A *valid AIO descriptor* is a reference to an AIO control block, which is not visible. A value of type `AIO_Descriptor` is a valid AIO descriptor if and only if it was returned by a call to `Create_AIO_Control_Block` and has not been subsequently invalidated by a call to `Destroy_AIO_Control_Block`.

The effect of assigning the value of one variable of type `AIO_Descriptor` to another variable of type `AIO_Descriptor` is to make a copy of the reference.

The application is responsible for the creation of the reference and the allocation of the control block using `Create_AIO_Control_Block`. The application is responsible for the destruction of the reference and the deallocation of the control block using `Destroy_AIO_Control_Block`.

`Create_AIO_Control_Block` shall allocate a control block, and return a value of type `AIO_Descriptor` that refers to it. Since `Create_AIO_Control_Block` is a function, the implementation shall be such that the value returned can be assigned to a variable of type `AIO_Descriptor`.

A call to Destroy_AIO_Control_Block shall deallocate the control block to which the AD parameter refers and destroy the reference, unless the object corresponds to an AIO request that is still being processed.

While an AIO control block is in use by the system for an AIO operation, the application is responsible for preserving the control block and the AIO buffer to which it refers, unchanged. In general, the effect of performing an operation that modifies or deallocates an AIO control block or AIO buffer while it is being used by the system, is undefined.

If Destroy_AIO_Control_Block is called for an AIO control block that corresponds to an AIO request that is still being processed, and if the system detects that the request is still being processed, the call shall raise POSIX_Error with error code Operation_Not_Permitted.

NOTE: An application can discover when the system is no longer using an AIO control block via the functions Get_AIO_Status and Get_AIO_Error_Code, which return In_Progress or Operation_In_Progress so long as the control block is in use.

6.3.1.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

No_Space_Left_On_Device
: Upon Create_AIO_Control_Block, a resource other than memory required to initialize the AIO control block has been exhausted.

Not_Enough_Space
: Upon Create_AIO_Control_Block, insufficient memory is available.

Operation_Not_Implemented
: Create_AIO_Control_Block and Destroy_AIO_Control_Block are not supported by this implementation.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: Upon Destroy_AIO_Control_Block, the value of AD is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise Constraint_Error.

Operation_Not_Permitted
: Destroy_AIO_Control_Block was invoked when the corresponding AIO operation was in progress.

6.3.2 Attributes of AIO Control Blocks

6.3.2.1 Synopsis

```
type List_IO_Operations is
   (No_Op,
    Read,
    Write);
```

```
type IO_Array_Pointer is access Ada_Streams.Stream_Element_Array;
function Get_File (AD : AIO_Descriptor)
  return POSIX_IO.File_Descriptor;
procedure Set_File
   (AD :   in AIO_Descriptor;
    File : in POSIX_IO.File_Descriptor);
function Get_Offset (AD : AIO_Descriptor) return POSIX_IO.IO_Offset;
procedure Set_Offset
   (AD :     in AIO_Descriptor;
    Offset : in POSIX_IO.IO_Offset);
function Get_Buffer (AD : AIO_Descriptor) return IO_Array_Pointer;
procedure Set_Buffer
   (AD :     in AIO_Descriptor;
    Buffer : in IO_Array_Pointer);
function Get_Length (AD : AIO_Descriptor)
  return Ada_Streams.Stream_Element_Count;
procedure Set_Length
   (AD :     in AIO_Descriptor;
    Length : in Ada_Streams.Stream_Element_Count);
function Get_Priority_Reduction (AD : AIO_Descriptor) return Natural;
procedure Set_Priority_Reduction
   (AD :                 in AIO_Descriptor;
    Priority_Reduction : in Natural);
function Get_Event (AD : AIO_Descriptor)
  return POSIX_Signals.Signal_Event;
procedure Set_Event
   (AD :    in AIO_Descriptor;
    Event : in POSIX_Signals.Signal_Event);
function Get_Operation (AD : AIO_Descriptor)
  return List_IO_Operations;
procedure Set_Operation
   (AD :        in AIO_Descriptor;
    Operation : in List_IO_Operations);
```

6.3.2.2 Description

An AIO control block has (at least) the following attributes:

File

The file descriptor on which the AIO operation is to be performed.

Offset

The offset within the file, at which the I/O operation is to be performed, specified as a count of bytes. For a read operation, and for a write operation if Append is not set for the File attribute, the requested operation takes place as if Seek were called immediately prior to the operation with an `Offset` equal to the Offset attribute and a `Starting_Point` equal to `From_Beginning`. For a write operation, if Append is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an AIO operation, the value of the file offset for the file is unspecified.

Buffer

A pointer to the buffer into which the data are to be read, or from which the data are to be written. The value is required to designate an object of type `Streams.Stream_Element_Array`. For read and write operations, the

designated object shall be interpreted in the same fashion as the `Buffer` arguments of `POSIX_IO.Read` and `POSIX_IO.Write`, respectively.

The application is responsible for the allocation and deallocation of I/O buffers. The effect of deallocating or modifying the contents of an I/O buffer before the I/O operation has completed and the Bytes Transferred attribute has been retrieved is undefined.

Length

The length of the data to be transferred, as a count of bytes.

Priority_Reduction

The difference between the process priority and the priority to be assigned to the I/O operation.

If the Prioritized I/O option and the Priority Process Scheduling option is supported: AIO operations are queued in priority order, with the priority of each AIO operation based on the current process priority of the calling process. The attribute Priority_Reduction can be used to lower (but not raise) the AIO priority of the operation and shall be within the range `0 .. Asynchronous_IO_Priority_Delta_Maximum`, inclusive. The order of processing of requests submitted by processes whose scheduling policies are not `FIFO_Within_Priorities` or `Round_Robin_Within_Priorities` is unspecified. The *AIO priority* of an AIO request is computed as the process priority minus the Priority_Reduction attribute of the AIO control block.

The AIO priority assigned to each AIO request is an indication of its desired order of execution relative to other AIO requests for this file. Requests issued with the same priority to a character special file shall be processed by the underlying device in FIFO order; the order of processing of requests of the same priority issued to files that are not character special files is unspecified. Numerically higher priority values indicate requests of higher priority. The value of the Priority_Reduction attribute of the AIO control block shall have no effect on process or task scheduling priority. When prioritized AIO requests to the same file are blocked waiting for a resource required for the I/O operations and the resource subsequently becomes available, the higher-priority requests shall be granted the resource before lower priority requests are granted the resource. The relative priority of AIO reuests and synchronous I/O requests is implementation defined.

If the Prioritized I/O option is supported: The implementation shall define for which files I/O prioritization is supported.

Event

A specification of how the calling process is to be notified of completion of the I/O operation. The value of the Event attribute shall be interpreted as specified in 3.3.12. If the Notification attribute of Event equals `No_Notification`, then no signal shall be generated on I/O completion, but the status of the request shall be set appropriately.

Operation

The I/O operation to be performed. The Operaton attribute is used only by `List_IO_No_Wait` and `List_IO_Wait` (see 6.3.5), which allow multiple AIO operations to be submitted at a single time.

Implementations may add other attributes, as permitted in 1.3.1.1.
NOTE: Adding attributes that may change the behavior of applications with respect to this standard when those attributes are uninitialized also requires that the extension be activated as described in 1.3.1.1.

Values of the type `List_IO_Operations` are used to specify an I/O operation to be performed, for use with the list directed I/O operations (see 6.3.5). The interpretations of values of this type, when they appear as the value of the Operation attribute of the AIO control block associated with an entry in an AIO descriptor list, are

`No_Op`
: The list entry shall be ignored.

`Read`
: An I/O operation shall be submitted as if by a call to `Read` with `AD` equal to the list entry.

`Write`
: An I/O operation shall be submitted as if by a call to `Write` with `AD` equal to the list entry.

The function whose name is of the form `Get_X`, where `X` is the name of one of the attributes of an AIO control block, shall return the value of the `X` attribute of the argument.

The procedure whose name is of the form `Set_X` shall set the `X` attribute of the object specified by the first argument to the value specified by the second argument.

Implementations may add other attributes, as permitted in 1.3.1.1.
NOTE: Adding attributes that may change the behavior of applications with respect to this standard when those attributes are uninitialized also requires that the extension be activated as described in 1.3.1.1.

6.3.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Operation_Not_Implemented`
: The operation is not supported by this implementation.

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: The value of `AD` is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise `Constraint_Error`.

6.3.3 Asynchronous Read

6.3.3.1 Synopsis

```
procedure Read (AD : in AIO_Descriptor);
```

6.3.3.2 Description

The Read operation allows the calling process to read the number of bytes specified by the Length attribute of AD from the file specified by the File attribute, into the buffer designated by the Buffer attribute. The procedure call returns when the read operation has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

If the Prioritized I/O option and the Priority Process Scheduling option is supported: If Prioritized I/O option is supported for this file the request is submitted with AIO priority as specified in 6.3.2.2.

Provided that the Read request is successfully queued, AD can be used as an argument to Get_AIO_Status, Get_AIO_Error_Code, and Get_Bytes_Transferred in order to retrieve the status of the AIO request during its *lifetime* (see 6.3.6).

The attributes of AD shall be interpreted as defined in 6.3.2. The Operation attribute of the AIO control block shall be ignored.

Concurrent AIO requests using the same value of the AD parameter produce undefined behavior.

If synchronized I/O is enabled for the File attribute of AD, the behavior of this procedure shall be according to the definitions of file synchronization (see 6.1.10) and data synchronization (see 6.1.11).

For any system action that changes the process memory space while an AIO operation is outstanding to the address range being changed, the result of that action is undefined.

6.3.3.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Temporarily_Unavailable
: The requested AIO operation was not queued due to system resource limitations.

Operation_Not_Implemented
: The Read procedure is not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: The value of AD is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise Constraint_Error.

Certain conditions may be detected synchronously at the time of the call to Read, or asynchronously. If any of the following conditions is detected synchronously, the exception POSIX_Error shall be raised, with the corresponding error code:

`Bad_File_Descriptor`

The File attribute of AD is not a valid file descriptor open for reading.

`Invalid_Argument`

The value of one or more of the Offset, Priority_Reduction, and Length attributes of AD is invalid.

If the operation successfully queues the I/O request, but it subsequently encounters an error, the error shall be reported via the Status Code attribute of the request, as specified in 6.3.6.

6.3.4 Asynchronous Write

6.3.4.1 Synopsis

```
procedure Write (AD : in AIO_Descriptor);
```

6.3.4.2 Description

The `Write` procedure allows the calling process to write Length bytes to the file associated with File from the buffer pointed to by Buffer (see 6.3.4), where Length, File, and Buffer are attributes of AD. The procedure call shall return when the write request has been initiated or, at a minimum, queued to the file or device.

If the Prioritized I/O option and the Priority Process Scheduling option is supported: If the Prioritized I/O option is supported for this file the request is submitted with AIO priority as specified in 6.3.2.2.

Provided that the `Write` request is successfully queued, AD can be used as an argument to `Get_AIO_Status`, `Get_AIO_Error_Code`, and `Get_Bytes_Transferred` in order to retrieve the status of the AIO request during its lifetime (see 6.3.6).

The attributes of AD shall be interpreted as defined in 6.3.2. The Operation attribute of the AIO control block shall be ignored.

Concurrent AIO requests using the same value of the AD parameter produce undefined behavior.

*If the Synchronized I/O option and the I option is supported:*f the Synchronized I/O option is supported on the file specified by the File attribute of AD, the behavior of this procedure shall be according to the definitions of file synchronization (see 6.1.10) and data synchronization (see 6.1.11).

For any system action that changes the process memory space while an AIO operation is outstanding to the address range being changed, the result of that action is undefined.

6.3.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Resource_Temporarily_Unavailable`

The requested AIO request was not queued due to system resource limitations.

Operation_Not_Implemented

The Write procedure is not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The value of AD is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise Constraint_Error.

Certain conditions may be detected synchronously at the time of the call to Write, or asynchronously. If any of the following conditions is detected synchronously, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor

The File attribute of AD is not a valid file descriptor open for writing.

Invalid_Argument

The value of one or more of the Offset, Priority_Reduction, and Length attributes of AD is invalid.

If the operation successfully queues the I/O request, but it subsequently encounters an error, the error shall be reported via the Status Code attribute of the request, as specified in 6.3.6.

6.3.5 List Directed I/O

6.3.5.1 Synopsis

```
type AIO_Descriptor_List is
   array (Positive range <>) of AIO_Descriptor;
procedure List_IO_No_Wait
   (List :  in out AIO_Descriptor_List;
    Event : in POSIX_Signals.Signal_Event);
procedure List_IO_Wait
   (List :           in out AIO_Descriptor_List;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

6.3.5.2 Description

The List_IO_No_Wait and List_IO_Wait procedures allow the calling process to initiate a list of I/O requests with a single procedure call. The effects of the two procedures differ only in whether the selected procedure returns when the I/O operations have completed or as soon as the requests have been queued. List_IO_Wait waits until all I/O is complete.

List_IO_Wait may be interruptible by the delivery of a signal. Masked_Signals specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

List_IO_No_Wait returns as soon as the requests have been queued, and signal generation shall occur according to the Event argument when all the requested I/O operations have completed.

Each component of List is required to be currently associated with an AIO control block, which specifies a requested I/O operation. The I/O requests enumerated by List shall be submitted in an unspecified order.

The Operation attribute of each List component specifies the operation to be performed, as described for type List_IO_Operations in 6.3.2. If the Operation attribute is Read or Write, the other attributes shall be interpreted in the same manner as for the AD parameter of the asynchronous Read or Write calls, respectively.

The Event argument, used with List_IO_No_Wait, specifies a signal event structure that defines the signal that shall be generated when the entire list of I/O requests have been completed. If the Notification attribute of Event is No_Notification, then no asynchronous notification occurs. If the Notification attribute of Event equals Signal_Notification, asynchronous notification as specified in 3.3.12 shall occur when all the requests in List have completed.

The behavior of this function is altered for each component of List according to the definitions of file synchronization (see 6.1.10) and data synchronization (see 6.1.11) if synchronized I/O is enabled for the File attribute.

For both List_IO_No_Wait and List_IO_Wait, if the I/O requests contained in the list are initiated or queued, one or more of the requests may fail. Failure of an individual request does not prevent completion of any other individual request. To discover the outcome of each I/O request, the application must examine the Status Code attribute (see 6.3.6) of each AIO descriptor in List, using Get_AIO_Status, Get_AIO_Error_Code, and Get_Bytes_Transferred during the *lifetime* of the request.

6.3.5.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Temporarily_Unavailable

The resources necessary to queue all the I/O requests were not available. The application can check the Status Code for each value of type AIO_Descriptor in the List to discover the individual request(s) that failed.

The number of I/O operations indicated by List'Length would cause the system-wide limit Asynchronous I/O Maximum to be exceeded.

Invalid_Argument

The value of List'Length is greater than List_IO_Maximum.

Interrupted_Operation

A signal was delivered during the wait for all I/O requests to complete, in a List_IO_Wait call.

NOTE: Since each I/O operation invoked by List_IO_Wait may possibly generate a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application must examine each list component to discover whether the request was initiated, canceled, or completed.

Input_Output_Error

One or more of the individual I/O operations failed during a List_IO_Wait operation. The application can check the Status Code for each value of type AIO_Descriptor in the List to discover the individual request(s) that failed.

Operation_Not_Implemented

List_IO_No_Wait and List_IO_Wait are not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The value of any of the components of List is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise Constraint_Error.

In addition to the errors returned by the List_IO_No_Wait and List_IO_Wait, if List_IO_No_Wait or List_IO_Wait succeeds or fails with error codes of Resource_Temporarily_Unavailable, Interrupted_Operation, or Input_Output_Error, then some of the I/O specified by the list may have been initiated. The I/O request indicated by each list component can encounter errors specific to the individual read or write function being performed. In this event, the Status Code attribute (see 6.3.6) associated with each list component carries information about the specific error (if any) encountered by the corresponding AIO operation. For each value of type AIO_Descriptor in the List corresponding to an I/O operation that encountered an error, the associated Status Code shall be as specified in 6.3.6.

6.3.6 Retrieve Status of AIO Request

6.3.6.1 Synopsis

```
type AIO_Status is
   (In_Progress,
    Completed_Successfully,
    Canceled);
function Get_AIO_Status
   (AD : AIO_Descriptor)
  return AIO_Status;
function Get_AIO_Error_Code
   (AD : AIO_Descriptor)
  return POSIX.Error_Code;
```

6.3.6.2 Description

The *lifetime* of an AIO request begins when the request is queued, or initiated (if it is initiated immediately without queueing), and ends when the corresponding AIO descriptor is used in a call to Get_Bytes_Transferred.

During its lifetime an AIO request has the following attributes, which are collectively termed the *status* of the request.

Status Code

This value of type `POSIX.Error_Code` indicates the stage of completion of the operation and, if the operation failed, indicates the nature of the failure. If the operation failed, the Status Code shall be the error code value that would be returned by `Get_Error_Code` if the corresponding synchronous operation (`POSIX_IO.Read` or `POSIX_IO.Write`) had failed for the same reason or one of the following values:

`Bad_File_Descriptor`

The request was for a `Read` operation, and the File attribute of the specified AIO descriptor is not a valid file descriptor open for reading; or the request was for a `Write` operation, and the File attribute of the specified AIO descriptor is not a valid file descriptor open for writing.

`Invalid_Argument`

The file offset value implied by the Offset attribute of the specified AIO descriptor would be invalid.

`Resource_Temporarily_Unavailable`

The request was part of a call to `List_IO_Wait`, and it was not queued due to resource limitations.

Otherwise, if the operation has not failed, the Status Code shall be as follows:

`In_Progress`

The request was successfully queued, but has not yet completed.

`Completed_Successfully`

The AIO request has completed successfully.

`Canceled`

The request was successfully queued, but was canceled before the I/O completed due to an explicit `Cancel` request.

Bytes Transferred

If the requested operation completed successfully or was canceled, it has an associated count of the number of bytes actually transferred by the I/O operation. This value shall be the same as would be returned in the parameter `Last` by the corresponding synchronous operation (the `POSIX_IO.Read` or `POSIX_IO.Write`).

NOTE: The status attributes of an AIO request defined here are not to be confused with and are not part of the attributes of an AIO control block defined in 6.3.2.

`Get_AIO_Status` shall raise `POSIX_Error`, with the Status Code of the request as error code, if the request has failed and `Get_Bytes_Transferred` has not yet been called for `AD`. If the request has not failed and `Get_Bytes_Transferred` has not yet been called for `AD`, the function shall return the value of type `AIO_Status` corresponding to the Status Code according to Table 6.2:

`Get_AIO_Error_Code` shall return the Status Code of the AIO request specified by `AD` if `Get_Bytes_Transferred` has not yet been called for `AD`.

The AIO descriptor originally specified in the call that requests initiation of an AIO operation can be used, thereafter, as a handle for retrieving the status of the I/O request. It shall remain valid for this purpose during its lifetime.

Table 6.2 – Error Codes and AIO Status Values

Status Code Value	AIO_Status Value
Operation_In_Progress	In_Progress
No_Error	Completed_Successfully
Operation_Canceled	Canceled

NOTE: An application may get status for an AIO descriptor repeatedly, but may not do so after a call to Get_Bytes_Transferred for the AIO descriptor.

6.3.6.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Implemented

Get_AIO_Status and Get_AIO_Error_Code are not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

AD does not refer to an AIO request whose Bytes Transferred attribute has not yet been retrieved. In other wordsshall, either no AIO operations have been requested via AD, or Get_Bytes_Transferred has already been called for AD since the last AIO request made via AD.

The value of AD is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise Constraint_Error.

In addition, for Get_AIO_Status, if the requested AIO operation has failed with an error code as described for POSIX_IO.Read, POSIX_IO.Write, POSIX_IO.Synchronize_File, or POSIX_IO.Synchronize_Data, then POSIX_Error shall be raised with the error code associated with the I/O request.

6.3.7 Retrieve Bytes Transferred by AIO Request

6.3.7.1 Synopsis

```
function Get_Bytes_Transferred
   (AD : AIO_Descriptor) return Ada_Streams.Stream_Element_Count;
```

6.3.7.2 Description

Get_Bytes_Transferred returns the Bytes Transferred attribute of the AIO request specified by AD, as defined in 6.3.6. If the I/O request is still being processed, then the value returned is undefined. The Get_Bytes_Transferred function shall be callable exactly once to retrieve the Bytes Transferred attribute for a given AIO request. Thereafter, if the same AIO descriptor is used in a call to Get_Bytes_Transferred, Get_AIO_Status, or Get_AIO_Error_Code, an error may be returned.

NOTE: When the same AIO descriptor is used to make another AIO request later, then Get_Bytes_Transferred can again be called to retrieve the status of that request.

NOTE: It is not clear in POSIX.1 whether `Get_Bytes_Transferred` *is required to return the number of bytes transferred if the operation is canceled. This standard recommends that implementations follow this practice, if possible.*

6.3.7.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

AD does not refer to an AIO request whose Bytes Transferred attribute has not yet been retrieved. In other words, either no AIO request has been made on AD, or `Get_Bytes_Transferred` has already been called for AD since the last request on AD.

`Operation_Not_Implemented`

The `Get_Bytes_Transferred` function is not supported by this implementation.

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The value of AD is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise `Constraint_Error`.

6.3.8 Cancel AIO Request

6.3.8.1 Synopsis

```
type Cancelation_Status is
   (Canceled,
    Not_Canceled,
    All_Done);
function Cancel
   (AD : AIO_Descriptor)
  return Cancelation_Status;
function Cancel
   (File : POSIX_IO.File_Descriptor)
  return Cancelation_Status;
```

6.3.8.2 Description

`Cancel` attempts to cancel one or more AIO requests. There are two forms of the function. The form with a AD parameter attempts to cancel the particular request associated with AD. The form with the `File` parameter attempts to cancel all outstanding cancelable AIO requests against the file descriptor `File`.

For AIO requests that are successfully canceled, the Status Code shall be set to `Canceled`, and the appropriate notification signal shall be generated if notification via signal was specified in the AIO request.

For AIO requests that are not successfully canceled, the attributes of the request are not modified by `Cancel`, and notification shall take place when the request completes.

It is implementation defined which requests are cancelable.

`Cancel` shall return one of the following according to the status of the I/O request.

`Canceled`
: The AIO requests were canceled.

`Not_Canceled`
: At least one of the requests cannot be canceled because the requested I/O operation is in progress. In this case, the state of the other requests, if any, referenced in the call to `Cancel` is not indicated by the return value of `Cancel`. The application can discover the state of affairs for these requests by using `Get_AIO_Status` or `Get_AIO_Error_Code`.

`All_Done`
: All of the requests have already completed.

6.3.8.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: The `File` parameter is not a valid file descriptor.

`Operation_Not_Implemented`
: `Cancel` is not supported by this implementation.

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: The value of `AD` is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise `Constraint_Error`.

6.3.9 Wait for AIO Request to Complete

6.3.9.1 Synopsis

```
procedure Await_IO_Or_Timeout
   (AD :             in AIO_Descriptor;
    Timeout :        in POSIX.Timespec;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Await_IO
   (AD :             in AIO_Descriptor;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Await_IO_Or_Timeout
   (List :           in AIO_Descriptor_List;
    Timeout :        in POSIX.Timespec;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
procedure Await_IO
   (List :           in AIO_Descriptor_List;
    Masked_Signals : in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

6.3.9.2 Description

`Await_IO` and `Await_IO_Or_Timeout` shall block the caller until one of the following occurs:

(1) The AIO request referenced by the `AD` argument has completed.

(2) At least one of the AIO requests referenced by the `List` argument has completed.

(3) The procedure is interrupted by a signal.

(4) For `Await_IO_Or_Timeout`, the time interval specified by `Timeout` has passed.

`Await_IO` and `Await_IO_Or_Timeout` may be interruptible by the delivery of a signal. `Masked_Signals` specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

The AIO descriptor specified by `AD` shall have been used in initiating an AIO request via `Read`, `Write`, or `List_IO_No_Wait`. If it is associated with a completed AIO request (*i.e.*, the Status Code for the request is not equal to `In_Progress` at the time of the call) the function shall return without blocking.

If any of the `AIO_Descriptor` values in the `List` correspond to completed AIO requests (*i.e.*, the Status Code of the request is not equal to `In_Progress` at the time of the call), the function shall return without blocking the caller. The `List` argument is an array of values of type `AIO_Descriptor`. Each value of type `AIO_Descriptor` in the `List` shall have been used in initiating an AIO request via `Read`, `Write`, or `List_IO_No_Wait`. If this array contains values of type `AIO_Descriptor` that have not been used in submitting AIO requests, the effect is undefined.

NOTE: For operations with a `List` parameter, an application may determine which AIO I/O request(s) completed by examining the associated Status Code using `Get_AIO_Status` or `Get_AIO_Error_Code` for each member of the `List`.

If the `Await_IO_Or_Timeout` procedure, if the time interval specified by the `Timeout` parameter passes before either the I/O request specified by `AD` is completed or any of the I/O requests specified by `List` are completed, then `Await_IO_Or_Timeout` shall raise `POSIX_Error`.

6.3.9.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Resource_Temporarily_Unavailable`

Upon `Await_IO_Or_Timeout`, no specified AIO request completed in the time interval indicated by `Timeout`.

`Interrupted_Operation`

A signal interrupted the `Await_IO` or `Await_IO_Or_Timeout` procedure.

NOTE: Since each requested AIO operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O requests being awaited.

Operation_Not_Implemented

The Await_IO and Await_IO_Or_Timeout procedures are not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The value of AD or the value of any of the components of List is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise Constraint_Error.

6.3.10 Asynchronous File and Data Synchronization

6.3.10.1 Synopsis

```
procedure Synchronize_File (AD : in AIO_Descriptor);
procedure Synchronize_Data (AD : in AIO_Descriptor);
```

6.3.10.2 Description

The functionality described in this subclause is optional. If the Synchronized I/O option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

Synchronize_File and Synchronize_Data asynchronously force all I/O requests associated with the file specified by the AD parameter and queued at the time of the call to the synchronized completion state. The procedure call to Synchronize_File or Synchronize_Data shall return when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

For a call to Synchronize_Data, all currently queued I/O requests are completed as if by a call to Synchronize_Data with the associated file descriptor as the File parameter. (See 6.1.11.) For a call to Synchronize_File, all currently queued I/O requests are completed as if by a call to Synchronize_File with the associated file descriptor as the File parameter. (See 6.1.10.) If the Synchronize_File or Synchronize_Data procedure fails, or if the request queued by Synchronize_File or Synchronize_Data fails, then, as for the forms of Synchronize_File or Synchronize_Data with the File parameter, outstanding I/O requests are not guaranteed to have been completed.

If Synchronize_File or Synchronize_Data succeeds, then only the I/O operations that were queued at the time of the call to Synchronize_File or Synchronize_Data are guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

During the *lifetime* of the AIO request (see 6.3.6), the AD value can be used as an argument to Get_AIO_Status, Get_AIO_Error_Code, and Get_Bytes_Transferred in order to obtain information about the *status* of the AIO request.

When the request is queued, the Status Code of the request shall be set to `In_Progress`. When all data have and been successfully transferred, the status shall be reset to reflect the success or failure of the request. If the request completes successfully, the Status Code of the request shall be set to `No_Error`. If the request does not complete successfully because it is canceled by `Cancel`, the Status Code of the request shall be set to `Canceled`. If the requested operation does not complete successfully for another reason, the Status Code of the request shall be set to indicate the error.

The Event attribute of AD shall determine the asynchronous notification to occur as specified in 3.3.12 when all requested I/O operations have achieved the synchronized I/O completion state (see 2.2.2.177).

All other attributes of the AIO control block referred to by AD shall be ignored.

If the `Synchronize_File` or `Synchronize_Data` procedure fails or there is an error condition associated with AD, data are not guaranteed to have been successfully transferred.

6.3.10.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Resource_Temporarily_Unavailable`
: The AIO request was not queued due to temporary resource limitations.

`Bad_File_Descriptor`
: The File attribute of AD is not a valid file descriptor open for writing.

`Invalid_Argument`
: This implementation does not support synchronized I/O for this file.

`Operation_Not_Implemented`
: `Synchronize_File` and `Synchronize_Data` are not supported by this implementation.

If any of the queued I/O requests fails, `Synchronize_File` and `Synchronize_Data` shall set the Status Code attribute of the request associated with AD to the error condition defined for `POSIX_IO.Read` and `POSIX_IO.Write`.

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: The value of AD is not a valid AIO descriptor. For this error, the implementation under some circumstances may instead raise `Constraint_Error`.

Section 7: Device- and Class-Specific Functions

This section contains a general terminal interface for communications ports on the system and contains operations on the controlling terminal of a process. The terminal interface includes a description of its interaction with the Job Control option, canonical input mode processing and input timer operations, and terminal special characters. The package `POSIX_Terminal_Functions` provides operations to get and set terminal characteristics and related operations on the serial line.

7.1 General Terminal Interface

This terminal interface shall be supported on any asynchronous communications ports provided by the implementation. It is file,implementation defined whether this interface supports network connections or synchronous ports or both. Some parts of this interface apply when an implementation supports the Job Control option. Certain subprograms in this section apply only to the controlling terminal of a process. Where either of these cases applyb is so noted.

7.1.0.4 Opening a Terminal Device File

When a terminal file is opened, the system causes the process to wait until a connection is established. In practice, application programs seldom open these files; they are opened by special programs and become the standard input, output, and error files of an application.

As noted in the description of `POSIX_IO.Open` (see 6.1.1), opening a terminal device file with the option `Non_Blocking` set to `False` shall cause the process to block until the terminal device is ready and available. The `Enable_Receiver` element of the `Control_Modes` subrange of the `Terminal_Modes_Set` array in package `POSIX_Terminal_Functions` (see 7.2.4) may also affect the opening of a terminal device file.

7.1.0.5 Process Groups

A terminal may have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters, as discussed in 7.1.0.12. If the implementation supports the Job Control option, a command interpreter process supporting the Job Control option can allocate the terminal to different jobs, or process groups, by placing related processes in a single process group and associating this process group with the terminal. The foreground process group of a terminal may be set or examined by a process, assuming the process has appropriate privileges; see subprograms `POSIX_Terminal_Functions.Get_Process_Group_ID` and `POSIX_Terminal_Functions.Set_Process_Group_ID` in 7.2.11. The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the foreground process group. (See 7.1.0.7.)

7.1.0.6 The Controlling Terminal

A terminal may belong to a process as the controlling terminal of that process. Each process of a session that has a controlling terminal shall have the same controlling terminal. A terminal may be the controlling terminal for at most one session.

The controlling terminal for a session shall be allocated by the session leader in an implementation-defined manner. If a session leader has no controlling terminal and opens a terminal device file that is not already associated with a session without using the `POSIX_IO.Not_Controlling_Terminal` option (see 6.1.1), it is implementation defined whether the terminal becomes the controlling terminal of the session leader. If a process that is not a session leader opens a terminal file or if the `POSIX_IO.Not_Controlling_Terminal` option is used on a call to either `POSIX_IO.Open` or `POSIX_IO.Open_Or_Create`, that terminal shall not become the controlling terminal of the calling process. When a controlling terminal becomes associated with a session, its foreground process group shall be set to the process group of the session leader.

The controlling terminal shall be inherited by a child process during `POSIX_Process_Primitives.Start_Process`, `POSIX_Process_Primitives.Start_Process_Search`, and `POSIX_Unsafe_Process_Primitives.Fork` subprogram invocations (see 3.1 and 3.2). A process shall relinquish its controlling terminal when it creates a new session with the `POSIX_Process_Identification.Create_Session` subprogram (see 4.1) or when all file descriptors associated with the controlling terminal have been closed.

When a controlling process terminates, the controlling terminal shall be disassociated from the current session, allowing it to be acquired by a new session leader. Subsequent access to the terminal by other processes in the earlier session may be denied, with attempts to access the terminal treated as if modem disconnect had been sensed. (See 7.1.0.13.)

7.1.0.7 Terminal Access Control

If a process is in the foreground process group of its controlling terminal, read operations shall be allowed as described in 7.1.0.8. For implementations that support the Job Control option, any attempts by a process in a background process group to read from its controlling terminal shall cause its process group to be sent `POSIX_Signals.Signal_Terminal_Input` unless one of the following cases applies:

— If the reading process is ignoring or blocking `Signal_Terminal_Input` or if the process group of the reading process is orphaned, the read shall raise `POSIX_Error` with error code `Input_Output_Error`, and no signal shall be sent. The default action of `Signal_Terminal_Input` shall be to stop the process to which it is sent. (See 3.3.4.)

— If a process is in the foreground process group of its controlling terminal, write operations shall be allowed as described in 7.1.0.11. Attempts by a process in a background process group to write to its controlling terminal shall cause the process group to be sent `POSIX_Signals.Signal_Terminal_Output` unless one of the following special cases applies:

  — If `Send_Signal_For_BG_Output` is disabled, or if `Send_Signal_For_BG_Output` is enabled, if the process is ignoring or blocking `Signal_Terminal_Output`, and if the process shall be allowed to write to the terminal and no signal shall be sent.

  — If `Send_Signal_For_BG_Output` is enabled and the process group of the writing process is orphaned, and the writing process is not ignoring or block-

ing `Signal_Terminal_Output`, the write operation shall raise `POSIX_Error` with error code `Input_Output_Error`, and no signal shall be sent.

Certain calls that set terminal parameters shall be treated in the same fashion as write operations, except that `Send_Signal_For_BG_Output` shall be ignored. In other words, the effect shall be identical to that for terminal write operations when `Send_Signal_For_BG_Output` is enabled. (See 7.2.)

7.1.0.8 Input Processing and Reading Data

A terminal device associated with a terminal device file may operate in full-duplex mode, so that data may arrive while output is occurring. Each terminal device file shall be associated with an input queue into which incoming data shall be stored by the system before being read by a process. If `POSIX_Configurable_File_Limits.Input_Queue_Is_Limited` returns `True`, the system shall impose a limit on the number of bytes that may be stored in the input queue. This limit is Input Queue Maximum (see 5.4). The behavior of the system when this limit is exceeded is implementation defined.

Two general kinds of input processing shall be available: canonical and noncanonical. These modes are described in 7.1.0.9 and 7.1.0.10. Additionally, input characters shall be processed according to the `Input_Modes` (see 7.2.2) and `Local_Modes` (see 7.2.5) characteristics. Such processing may include *echoing*, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. Echoing is useful for terminals that can operate in full-duplex mode.

The manner in which data are provided to a process reading from a terminal device file shall be dependent on whether the terminal device file is in canonical or noncanonical mode.

The `Non_Blocking` option for `POSIX_IO.Open` (see 6.1.1 and `POSIX_IO.Set_File_Control` in 6.1.7) shall also affect the manner in which data are provided to a process. If the option `Non_Blocking` was set to `False`, then any read request on the specified file shall be blocked until data are available or a signal has been received. If the option `Non_Blocking` was set to `True`, then the read request shall be completed, without blocking, in one of three ways:

(1) If enough data are available to satisfy the entire request, the read operation shall complete successfully and return the index of the last byte read.

(2) If not enough data are available to satisfy the entire request, the read operation shall complete successfully, having read as much data as possible, and return the index of the last byte it was able to read.

(3) If no data are available, the read operation shall instead return immediately with `Last` set to `Buffer'First - 1`.

When data are available depends on whether the input processing mode is canonical or noncanonical. Subclauses 7.1.0.9 and 7.1.0.10 describe each of these input processing modes.

7.1.0.9 Canonical Mode Input Processing

In canonical mode input processing, terminal input shall be processed in units of lines. A line is delimited by a newline character (NL), an end-of-file character (EOF), or an end-of-line character (EOL). See 7.1.0.12 for more information on NL, EOF, and EOL.

In other words, a read request shall not return until an entire line has been typed (*i.e.*, at least an NL, EOF, or EOL is entered) or a signal has been received. Also, no matter how many POSIX characters are requested in the read call, at most one line shall be returned. It shall not be necessary to read a whole line at once; any number of POSIX characters, even one, may be requested in a read without losing information.

The limit on the number of POSIX characters in a line is Input Line Maximum. The effect if this limit is exceeded is implementation defined. If `POSIX_Configurable_File_Limits.Input_Line_Is_Limited` returns `False`, there shall be no such limit. (See 5.4.) Erase and kill processing shall occur when either of two special characters Erase and Kill (see 7.1.0.12), is received. This processing shall affect data in the input queue that have not yet been delimited by an NL, EOF, or EOL character. This undelimited data shall make up the current line. The Erase character shall delete the last character in the current line, if there is any. The Kill character shall delete all data in the current line, if there is any. The Erase and Kill characters shall have no effect if no data are in the current line. The Erase and Kill characters themselves shall not be placed in the input queue.

7.1.0.10 Noncanonical Mode Input Processing

In noncanonical mode input processing, input POSIX characters shall not be assembled into lines, and erase and kill processing shall not occur. The values of the `Minimum_Input_Count` and `Input_Time` components of the type `Terminal_Characteristics` shall be used to determine how to process the POSIX characters received.

`Minimum_Input_Count` shall represent the minimum number of POSIX characters that should be received when the read operation function successfully returns. `Input_Time` shall be a timer that is used to time out burst and short-term data transmissions. If `Minimum_Input_Count` is greater than the value `POSIX_Limits.Input_Queue_Maxima'Last` or greater than Input Queue Maximum for the file (see 2.4 and 5.4), the response to the request is implementation defined. The four sets of values for `Minimum_Input_Count` and `Input_Time` and their interactions are described as follows:

— `Minimum_Input_Count > 0`, `Input_Time > 0.0`

In this case, `Input_Time` serves as an intercharacter timer and shall be activated after the first byte is received. Since it is an intercharacter timer, it shall be reset after a byte is received. The interaction between `Minimum_Input_Count` and `Input_Time` shall be as follows: as soon as one byte is received, the intercharacter timer shall be started. If `Minimum_Input_Count` bytes are received before the intercharacter timer expires (a situation that can occur because the timer is reset upon receipt of each byte), the read request shall be satisfied. If the timer expires before `Minimum_Input_Count`

bytes are received, the characters received to that point shall be returned to the user. If `Input_Time` expires, at least one byte value shall be returned because the timer would not have been enabled unless a byte was received. In this case (`Minimum_Input_Count > 0`, `Input_Time > 0.0`), the read operation shall block until the `Minimum_Input_Count` and `Input_Time` mechanisms are activated by the receipt of the first byte or until a signal is received. A process that uses this case to read record-based terminal I/O may block indefinitely in the read operation.

— `Minimum_Input_Count > 0`, `Input_Time = 0.0`

In this case, since the value of `Input_Time` is zero, the timer shall play no role and only `Minimum_Input_Count` shall be significant. A pending read request shall not be satisfied until `Minimum_Input_Count` bytes have been received (*i.e.*, the pending read request shall block until `Minimum_Input_Count` bytes have been received) or until a signal is received. A process that uses this case to read record-based terminal I/O may block indefinitely in the read operation.

— `Minimum_Input_Count = 0`, `Input_Time > 0.0`

In this case, since `Minimum_Input_Count = 0`, `Input_Time` shall no longer represent an intercharacter timer. It shall now serve as a read timer that shall be activated as soon as the read operation is processed. A read request shall be satisfied as soon as a single byte is received or when the read request timer expires. In this case, if the timer expires, no bytes shall be returned. If the timer does not expire, the only way the read request can be satisfied is if a byte is received. In this case, the read shall not block indefinitely waiting for a byte, if no byte is received within `Input_Time` seconds after the read is initiated, the read operation shall return a value of `Buffer'First - ` in the `Last` parameter, and the contents of the `Buffer` are undefined since no data shall have been read.

— `Minimum_Input_Count = 0`, `Input_Time = 0.0`

The minimum of either the number of bytes requested or the number of bytes currently available shall be returned without waiting for more bytes to be input. If no bytes are available, the read operation shall return a value of `Buffer'First -1` in the `Last` parameter, and the contents of the `Buffer` will be undefined since no data shall have been read. The task shall not block.

7.1.0.11 Writing Data and Output Processing

When a process writes one or more bytes to a terminal device file, they shall be processed according to the values of the associated `Output_Modes`. The implementation may provide a buffering mechanism. Therefore, when a call to a write operation completes, all of the bytes written shall have been scheduled for transmission to the device, but the transmission will not necessarily have completed. See also 6.1.1 and 6.1.4 for the effects of the `Non_Blocking` option for `POSIX_IO.Open` and `POSIX_IO.Open_Or_Create` functions on `POSIX_IO.Write`.

7.1.0.12 Special Characters

Certain characters shall have special functions on input or output or both when operating in canonical mode. These functions are summarized as follows:

Interrupt: The Interrupt special character shall be recognized on input if the `Enable_Signals` mode is enabled as described in 7.2.5. The Interrupt special character shall cause `POSIX_Signals.Signal_Interrupt` to be sent to all processes in the foreground process group for which the terminal is the controlling terminal.

Quit: The Quit special character shall be recognized on input if the `Enable_Signals` mode is enabled as described in 7.2.5. The Quit special character shall cause `POSIX_Signals.Signal_Quit` to be sent to all processes in the foreground process group for which the terminal is the controlling terminal.

Erase: The Erase special character shall be recognized on input if `Canonical_Input` is enabled as described in 7.2.5. The Erase special character shall erase the last character in the current line. (See 7.1.0.9.) It shall not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character.

Kill: The Kill special character shall be recognized on input if `Canonical_Input` is enabled as described in 7.2.5. It shall delete the entire line, as delimited by an NL, EOF, or EOL character.

EOF: The end-of-file special character shall be recognized on input if `Canonical_Input` is enabled as described in 7.2.5. When EOF is received, all the bytes available to be read shall immediately be passed to the process, without waiting for a newline, and the EOF shall be discarded. If there are no bytes available (that is, the EOF occurred at the beginning of a line), the read operation shall return a value of zero in the `Last` parameter, and the contents of the `Buffer` will be undefined. Any subsequent read operation shall raise the `IO_Exceptions.End_Error` exception.

NL: The newline special character shall be recognized on input if `Canonical_Input` is enabled as described in 7.2.5. It shall be the line delimiter. The NL character should be the `POSIX_Character` translation of `ASCII.LF`.

EOL: The end-of-line special character shall be recognized on input if `Canonical_Input` is enabled as described in 7.2.5. It shall be an additional line delimiter, like NL.

Suspend: *If the Job Control option is supported* (see 7.2.8), the Suspend special character shall be recognized on input. If the `Enable_Signals` mode is enabled as described in 7.2.5, receipt of the Suspend character shall cause `POSIX_Signals.Signal_Terminal_Stop` to be sent to all processes in the foreground process group for which the terminal is the controlling terminal.

Stop: The Stop special character shall be recognized on both input and output if the `Enable_Start_Stop_Input` or `Enable_Start_Stop_Output`

mode is enabled. The Stop character shall be used to temporarily suspend output. It is useful with cathode-ray-tube (video) terminals terminals to prevent output from disappearing before it can be read.

Start: The Start special character shall be recognized on both input and output if the `Enable_Start_Stop_Input` or `Enable_Start_Stop_Output` mode is enabled. The Start character shall be used to resume output that has been suspended by a Stop character. The effect of a Start character in the absence of a previous Stop character is implementation defined.

CR: The CR special character shall be recognized on input if the `Canonical_Input` mode flag is enabled. When `Canonical_Input` and `Map_CR_To_LF` are enabled and `Ignore_CR` is disabled, this character shall be translated into an NL and shall have the same effect as an NL character.

If a special character (except NL, EOL, and CR) is recognized as described in the preceding paragraphs, then the character shall be removed from the input stream and discarded when it is processed. If the special character is not recognized, then it shall be passed through the input buffer as a normal character.

The NL and CR characters cannot be changed. It is implementation defined whether the Start and Stop characters can be changed. The values for Interrupt, Quit, Erase, Kill, EOF, EOL, and Suspend (only if the Job Control option is supported) shall be changeable to suit individual tastes using the procedure `Define_Special_Control_Character`. (See 7.2.8.)

If two or more special characters have the same value, the function performed when that character is received is undefined.

A special character shall be recognized not only by its value, but also by its context; for example, an implementation may define a sequence of multiple `POSIX_Character` values that have a meaning different from the meaning of the `POSIX_Character` value when considered individually. Implementations may also define additional single `POSIX_Character` functions. These implementation-defined sequences of multiple `POSIX_Character` values or single `POSIX_Character` functions shall be recognized only if the `Extended_Functions` mode is enabled; otherwise, data shall be interpreted as normal characters or as the special characters defined in this section.

7.1.0.13 Modem Disconnect

If a modem disconnect is detected by the terminal interface for a controlling terminal and if `Ignore_Modem_Status` mode is disabled for the terminal (see 7.2.4), `POSIX_Signals.Signal_Hangup` shall be sent to the controlling process associated with the terminal. Unless other arrangements have been made (see 3.3.4) delivery of the `Signal_Hangup` shall cause the controlling process to terminate. Any subsequent read from the terminal device shall return with an EOF indication until the device is closed. Thus, processes that read a terminal file and test for EOF can terminate appropriately after a disconnect. Any subsequent write to the terminal device shall raise `POSIX_Error` with error code `Input_Output_Error` until the device is closed.

7.1.0.14 Closing a Terminal Device File

The last process to close a terminal device file shall cause any output to be sent to the device and any input to be discarded. Then, if `Hang_Up_On_Last_Close` is enabled in the control modes and the communications port supports a disconnect function, the terminal device shall perform a disconnect.

7.2 Package `POSIX_Terminal_Functions`

This clause describes the subprograms that are used to control the general terminal function. If the implementation supports the Job Control option, unless otherwise noted for a specific command, these subprograms shall be restricted from use by background processes. Attempts to perform these operations shall cause the process group to be sent `POSIX_Signals.Signal_Terminal_Output`. If the calling process is blocking or ignoring `POSIX_Signals.Signal_Terminal_Output`, the process is allowed to perform the operation, and `Signal_Terminal_Output` is not sent.

In all the subprograms, `File` is an open file descriptor. However, the subprograms affect the underlying terminal device file, not just the open file description associated with the file descriptor.

```
with POSIX,
     POSIX_IO,
     POSIX_Process_Identification;
package POSIX_Terminal_Functions is
   Null_POSIX_Character: constant POSIX.POSIX_Character
     := POSIX.POSIX_Character'VAL(implementation-defined);
   Flag_POSIX_Character: constant POSIX.POSIX_CHARACTER
     := POSIX.POSIX_Character'VAL(implementation-defined);
    -- 7.2.1 Terminal Characteristics
   type Terminal_Characteristics is private;
   Invalid_Terminal_Characteristics: constant Terminal_Characteristics;
   function Get_Terminal_Characteristics
      (File : POSIX_IO.File_Descriptor)
     return Terminal_Characteristics;
   type Terminal_Action_Times is
      (Immediately, After_Output, After_Output_And_Input);
   procedure Set_Terminal_Characteristics
      (File:            in POSIX_IO.File_Descriptor;
       Characteristics: in Terminal_Characteristics;
       Apply:           in Terminal_Action_Times := Immediately;
       Masked_Signals:  in POSIX.Signal_Masking := POSIX.RTS_Signals);
   type Terminal_Modes is
      -- Subtype Input˙Modes:
      (Interrupt_On_Break, Map_CR_To_LF, Ignore_Break,
       Ignore_CR, Ignore_Parity_Errors, Map_LF_To_CR,
       Enable_Parity_Check, Strip_Character, Enable_Start_Stop_Input,
       Enable_Start_Stop_Output, Mark_Parity_Errors,
       -- Subtype Output˙Modes:
       Perform_Output_Processing,
       -- Subtype Control˙Modes:
       Ignore_Modem_Status, Enable_Receiver, Send_Two_Stop_Bits,
       Hang_Up_On_Last_Close, Parity_Enable, Odd_Parity,
```

```
      -- Subtype Local_Modes:
      Echo, Echo_Erase, Echo_Kill, Echo_LF, Canonical_Input,
      Extended_Functions, Enable_Signals, No_Flush,
      Send_Signal_For_BG_Output);
   -- 7.2.2 Input Modes
   subtype Input_Modes is Terminal_Modes
      range Interrupt_On_Break .. Mark_Parity_Errors;
   -- 7.2.3 Output Modes
   subtype Output_Modes is Terminal_Modes
      range Perform_Output_Processing .. Perform_Output_Processing;
   -- 7.2.4 Control Modes
   subtype Control_Modes is Terminal_Modes
      range Ignore_Modem_Status .. Odd_Parity;
   -- 7.2.5 Local Modes
   subtype Local_Modes is Terminal_Modes
      range Echo .. Send_Signal_For_BG_Output;
   -- 7.2.6 Retrieve and Define Terminal Modes and Bits per Character
   type Terminal_Modes_Set is array (Terminal_Modes) of Boolean;
   subtype Bits_Per_Character is Positive range implementation-defined;
   function Terminal_Modes_Of
      (Characteristics: Terminal_Characteristics)
     return Terminal_Modes_Set;
   procedure Define_Terminal_Modes
      (Characteristics: in out Terminal_Characteristics;
       Modes:           in     Terminal_Modes_Set);
   function Bits_Per_Character_Of
      (Characteristics: Terminal_Characteristics)
     return Bits_Per_Character;
   procedure Define_Bits_Per_Character
      (Characteristics: in out Terminal_Characteristics;
       Bits:            in     Bits_Per_Character);
   -- 7.2.7 Baud Rate Subprograms
    type Baud_Rate is
        (B0, B50, B75, B110, B134, B150, B200, B300, B600,
         B1200, B1800, B2400, B4800, B9600, B19200, B38400);
    function Input_Baud_Rate_Of
       (Characteristics: Terminal_Characteristics)
      return Baud_Rate;
    procedure Define_Input_Baud_Rate
       (Characteristics: in out Terminal_Characteristics;
        Input_Baud_Rate: in     Baud_Rate);
    function Output_Baud_Rate_Of
       (Characteristics: Terminal_Characteristics)
      return Baud_Rate;
    procedure Define_Output_Baud_Rate
       (Characteristics:  in out Terminal_Characteristics;
        Output_Baud_Rate: in     Baud_Rate);
   -- 7.2.8 Special Control Characters
   type Control_Character_Selector is
      (EOF_Char, EOL_Char, Erase_Char, Interrupt_Char,
       Kill_Char, Quit_Char, Suspend_Char, Start_Char, Stop_Char);
   function Special_Control_Character_Of
      (Characteristics: Terminal_Characteristics;
       Selector: Control_Character_Selector)
     return POSIX.POSIX_Character;
```

```
   procedure Define_Special_Control_Character
      (Characteristics: in out Terminal_Characteristics;
       Selector:        in     Control_Character_Selector;
       Char:            in     POSIX.POSIX_Character);
   procedure Disable_Control_Character
      (Characteristics: in out Terminal_Characteristics;
       Selector:        in     Control_Character_Selector);
   -- 7.2.9 Noncanonical Controls
   function Input_Time_Of
      (Characteristics: Terminal_Characteristics)
     return Duration;
   procedure Define_Input_Time
      (Characteristics: in out Terminal_Characteristics;
       Input_Time:      in     Duration);
   function Minimum_Input_Count_Of
      (Characteristics: Terminal_Characteristics)
     return Natural;
   procedure Define_Minimum_Input_Count
      (Characteristics:     in out Terminal_Characteristics;
       Minimum_Input_Count: in     Natural);
   -- 7.2.10 Line Control Operations
   procedure Send_Break
      (File:         in POSIX_IO.File_Descriptor;
       The_Duration: in Duration := 0.0);
   procedure Drain
      (File:           in POSIX_IO.File_Descriptor;
       Masked_Signals: in POSIX.Signal_Masking := POSIX.RTS_Signals);
   type Queue_Selector is
      (Received_But_Not_Read, Written_But_Not_Transmitted, Both);
   procedure Discard_Data
      (File:     in POSIX_IO.File_Descriptor;
       Selector: in Queue_Selector);
   type Flow_Action is
      (Suspend_Output, Restart_Output, Transmit_Stop, Transmit_Start);
   procedure Flow
      (File:   in POSIX_IO.File_Descriptor;
       Action: in Flow_Action);
   -- 7.2.11 Foreground Process Group ID
   function Get_Process_Group_ID
      (File: POSIX_IO.File_Descriptor)
     return POSIX_Process_Identification.Process_Group_ID;
   procedure Set_Process_Group_ID
      (File:     in POSIX_IO.File_Descriptor;
       Group_ID: in POSIX_Process_Identification.Process_Group_ID);
   -- 7.2.12 Generate Terminal Pathname
   function Get_Controlling_Terminal_Name return POSIX.Pathname;

private
   implementation-defined
end POSIX_Terminal_Functions;
```

7.2.1 Terminal Characteristics

7.2.1.1 Synopsis

```
   type Terminal_Characteristics is private;
   Invalid_Terminal_Characteristics: constant Terminal_Characteristics;
   function Get_Terminal_Characteristics
```

```
   (File : POSIX_IO.File_Descriptor)
  return Terminal_Characteristics;
type Terminal_Action_Times is
   (Immediately, After_Output, After_Output_And_Input);
procedure Set_Terminal_Characteristics
   (File:            in POSIX_IO.File_Descriptor;
    Characteristics: in Terminal_Characteristics;
    Apply:           in Terminal_Action_Times := Immediately;
    Masked_Signals:  in POSIX.Signal_Masking := POSIX.RTS_Signals);
type Terminal_Modes is
   -- Subtype Input˙Modes:
   (Interrupt_On_Break, Map_CR_To_LF, Ignore_Break,
    Ignore_CR, Ignore_Parity_Errors, Map_LF_To_CR,
    Enable_Parity_Check, Strip_Character, Enable_Start_Stop_Input,
    Enable_Start_Stop_Output, Mark_Parity_Errors,
    -- Subtype Output˙Modes:
    Perform_Output_Processing,
    -- Subtype Control˙Modes:
    Ignore_Modem_Status, Enable_Receiver, Send_Two_Stop_Bits,
    Hang_Up_On_Last_Close, Parity_Enable, Odd_Parity,
    -- Subtype Local˙Modes:
    Echo, Echo_Erase, Echo_Kill, Echo_LF, Canonical_Input,
    Extended_Functions, Enable_Signals, No_Flush,
    Send_Signal_For_BG_Output);
```

7.2.1.2 Description

Programs that need to control certain terminal I/O characteristics shall do so by using the `Terminal_Characteristics` private type as defined in the package `POSIX_Terminal_Functions` and the operations `Get_Terminal_Characteristics` and `Set_Terminal_Characteristics` on that type. All objects of type `Terminal_Characteristics` shall be initialized to the value `Invalid_Terminal_Characteristics`. This value denotes that valid data have not been stored into the object via a call to `Get_Terminal_Characteristics`. All objects of type `Terminal_Characteristics` should be initialized by a call to `Get_Terminal_Characteristics` before they are used. Characteristics denoted by this type include (but are not limited to) those shown in Table 7.1.

Table 7.1 – `Terminal_Characteristics` Components

Component Name	Used For
`Terminal_Modes_Set`	Subdivided into Input modes, Output modes, Control modes, and Local modes. The modes control the functions of the terminal.
`Input_Baud_Rate`	Terminal input data rate.
`Output_Baud_Rate`	Terminal output data rate.
`Control_Characters`	Array of terminal interface Control Characters.
`Input_Time`	Time value for timer used in noncanonical input.
`Minimum_Input_Count`	`POSIX_Character` count value used for noncanonical input.

The characteristics defined in this package shall not take effect until they have been set via an invocation of the procedure `Set_Terminal_Characteristics`. This procedure should be used to set the state specified by the parameter `Characteristics`. The parameter `Apply` shall be used to determine when the characteristics are to be updated, as follows:

— `Immediately` shall instruct the system to change the values immediately.

— `After_Output` shall instruct the system to wait until all output written to the parameter `File` has been transmitted. This mode should be used when the change affects output.

— `After_Output_And_Input` shall instruct the system to wait until all output has been transmitted. All input received and not read shall be discarded before the change is made.

The current state of the terminal characteristics shall be obtained via the function `Get_Terminal_Characteristics`. Components of the terminal characteristics shall be modified by subprograms defined in the following subclauses. Modifications to the terminal characteristics then shall be applied to the terminal via the `Set_Terminal_Characteristics` procedure. The parameter `Masked_Signals` in `Set_Terminal_Characteristics` specifies which signals shall be masked during the operation. (See 2.4.1.6.)

The zero baud rate `B0` shall be used to terminate the connection. If `B0` is specified as the output baud rate in the parameter `Characteristics` when `Set_Terminal_Characteristics` is called, the modem control lines no longer shall be asserted. Normally, no longer asserting the modem control lines will disconnect the line.

If the input baud rate is equal to `B0` in the parameter `Characteristics` when `Set_Terminal_Characteristics` is called, the input baud rate will be changed by `Set_Terminal_Characteristics` to the same value as that specified by the value of the output baud rate, exactly as if the input rate had been set to the output rate by a call to `Define_Input_Baud_Rate`. This usage of `B0` is obsolescent.

`Set_Terminal_Characteristics` shall return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It shall set all the attributes that the implementation does support as requested and leave all the attributes not supported by the hardware unchanged. If no part of the request can be honored, it shall raise `POSIX_Error` with error code `Invalid_Argument`. If the input and output baud rates differ and are a combination that is not supported, neither baud rate shall be changed. A subsequent call to `Get_Terminal_Characteristics` shall return the actual state of the terminal device (reflecting both the changes made and not made in the previous call to `Set_Terminal_Characteristics`).

No actions defined by this standard, other than a call to `Set_Terminal_Characteristics` or a call to `POSIX_IO.Close` on the last file descriptor in the system associated with this terminal device, shall cause any of the terminal attributes defined by this standard to change.

The type `Terminal_Modes` is divided into subranges `Input_Modes`, `Output_Modes`, `Control_Modes`, and `Local_Modes`.

7.2.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

Bad_File_Descriptor

The parameter File is not a valid POSIX_IO.File_Descriptor.

Inappropriate_IO_Control_Operation

The file associated with the parameter File is not a terminal.

Interrupted_Operation

A signal interrupted the Set_Terminal_Characteristics procedure.

Invalid_Argument

Either the value of Characteristics was not initialized via a call to Get_Terminal_Characteristics, or none of the terminal characteristics requested by Characteristics is supported for the terminal device associated with the parameter File.

7.2.2 Input Modes

7.2.2.1 Synopsis

```
subtype Input_Modes is Terminal_Modes
   range Interrupt_On_Break .. Mark_Parity_Errors;
```

7.2.2.2 Description

Values of the subtype Input_Modes, shown in Table 7.2, shall describe the basic terminal input control.

Table 7.2 – Terminal_Modes Values for Input Control

Value	Meaning
Enable_Parity_Check	Enable input parity check
Enable_Start_Stop_Input	Enable start/stop input control
Enable_Start_Stop_Output	Enable start/stop output control
Ignore_Break	Ignore break condition
Ignore_CR	Ignore CR
Ignore_Parity_Errors	Ignore characters with parity errors
Interrupt_On_Break	Signal interrupt on break
Map_CR_To_LF	Map CRto NL on input
Map_LF_To_CR	Map NL to CR on input
Mark_Parity_Errors	Mark parity errors
Strip_Character	Strip character

In the context of asynchronous serial data transmission, a break condition is defined as a sequence of zero-valued bits that continues for more than the time to send one byte. The entire sequence of zero-valued bits shall be interpreted as a single break condition, even if it continues for a time equivalent to more than one byte. In contexts other than asynchronous serial data transmission, the definition of a break condition is implementation defined.

If Ignore_Break mode is enabled, a break condition detected on input shall be ignored, that is, not put on the input queue and, therefore, not read by any process. If Ignore_Break mode is disabled and Interrupt_On_Break is enabled, the break condition shall flush the input and output queues. Furthermore, if the terminal is the

controlling terminal of a foreground process group, the break condition shall generate a single `POSIX_Signals.Signal_Interrupt` signal to that foreground process group. If neither `Ignore_Break` nor `Interrupt_On_Break` mode is enabled, a break condition shall be read as a single `Null_POSIX_Character` or, if `Mark_Parity_Errors` mode is enabled, as the three-character sequence `Flag_POSIX_Character`, `Null_POSIX_Character`, `Null_POSIX_Character`.

If `Ignore_Parity_Errors` mode is disabled, a byte with a framing or parity error (other than break) shall be ignored.

If `Mark_Parity_Errors` mode is enabled and `Ignore_Parity_Errors` mode is disabled, a byte with a framing or parity error (other than break) shall be given to the application as the three-character sequence `Flag_POSIX_Character`, `Null_POSIX_Character`, *X*, where *X* is the data of the byte received in error. To avoid ambiguity in this case, a valid character of `Flag_POSIX_Character` shall be given to the application as the two-character sequence `Flag_POSIX_Character`, `Flag_POSIX_Character`. If neither `Mark_Parity_Errors` nor `Ignore_Parity_Errors` mode is enabled, a framing or parity error (other than break) shall be given to the application as a single character `Null_POSIX_Character`.

If `Enable_Parity_Check` mode is enabled, input parity checking shall be enabled. If `Enable_Parity_Check` mode is disabled, input parity checking shall be disabled, allowing output parity generation without input parity errors.

NOTE: Whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled. (See 7.2.4.)

If parity detection is enabled but input parity checking is disabled, the hardware to which the terminal is connected shall recognize the parity bit, but the terminal special file shall not check whether this bit is set correctly.

If `Strip_Character` mode is enabled, valid input bytes first shall be stripped to seven bits; otherwise, valid input bytes shall be processed in their entirety.

If `Map_LF_To_CR` mode is enabled, a received NL character shall be translated into the CR character. If `Ignore_CR` mode is enabled, a received CR character shall be ignored (discarded). If `Ignore_CR` mode is disabled and `Map_CR_To_LF` mode is enabled, a received CR character shall be translated into the NL character.

If `Enable_Start_Stop_Output` mode is enabled, *start/stop output control* shall be enabled. A received Stop character shall suspend output, and a received Start character shall restart output. When `Enable_Start_Stop_Output` mode is enabled, Start and Stop characters shall be removed from the input stream, but merely perform flow control functions. When `Enable_Start_Stop_Output` mode is disabled, the Start and Stop characters shall be ignored by the operating system and shall be passed to the applications program when read.

If `Enable_Start_Stop_Input` mode is enabled, *start/stop input control* shall be enabled. The system shall transmit one or more Stop characters, which are intended to cause the terminal device to stop transmitting data, as needed to prevent the number of characters in the input queue from exceeding Input Queue Maximum. The system shall also transmit one or more Start characters, which are intended to cause the terminal device to resume transmitting data, as soon as the device can continue

transmitting data without risk of overflowing the input queue. The precise conditions under which Stop and Start characters are transmitted are implementation defined.

The initial input control value after a call to `POSIX_IO.Open` is implementation defined.

7.2.3 Output Modes

7.2.3.1 Synopsis

```
subtype Output_Modes is Terminal_Modes
   range Perform_Output_Processing .. Perform_Output_Processing;
```

7.2.3.2 Description

Values of the subtype `Output_Modes`, shown in Table 7.3, shall describe the basic terminal output control.

Table 7.3 – `Terminal_Modes` Values for Output Control

Value	Meaning
`Perform_Output_Processing`	Perform output processing

If `Perform_Output_Processing` mode is enabled, output data shall be processed in an implementation-defined fashion so that lines of text are modified to appear appropriately on the terminal device; otherwise, characters shall be transmitted without change.

The initial input control value after a call to `POSIX_IO.Open` is implementation defined.

7.2.4 Control Modes

7.2.4.1 Synopsis

```
subtype Control_Modes is Terminal_Modes
   range Ignore_Modem_Status .. Odd_Parity;
```

7.2.4.2 Description

Values of the subtype `Control_Modes`, shown in Table 7.4, shall describe the basic terminal hardware control modes.

Table 7.4 – `Terminal_Modes` Values for Hardware Control

Value	Meaning
`Enable_Receiver`	Enable receiver
`Hang_Up_On_Last_Close`	Hang up on last close
`Ignore_Modem_Status`	Ignore modem status lines
`Odd_Parity`	Odd parity, else even
`Parity_Enable`	Parity enable
`Send_Two_Stop_Bits`	Send two stop bits, else one

The Bits_Per_Character type shall specify the character size in bits for both transmission and reception. This size shall not include the parity bit, if any.

If Send_Two_Stop_Bits mode is enabled, two stop bits shall be used; otherwise, one stop bit shall be used. For example, at 110 baud, two stop bits are used by convention.

If Enable_Receiver mode is enabled, the receiver shall be enabled. Otherwise, no characters shall be received.

If Parity_Enable mode is enabled, parity generation and detection shall be enabled and a parity bit shall be added to each character. If parity is enabled, Odd_Parity shall specify odd parity, if enabled; otherwise, even parity shall be assumed.

If Hang_Up_On_Last_Close mode is enabled, the modem control lines for the port shall be lowered when the last process with the port open closes the port or the process terminates. The modem connection shall be broken.

If Ignore_Modem_Status mode is enabled, a connection shall not depend on the state of the modem status lines. If Ignore_Modem_Status mode is disabled, the modem status lines shall be monitored.

By default, a call to POSIX_IO.Open or POSIX_IO.Open_Or_Create shall wait for the modem connection to complete before returning. However, if the Non_Blocking option is True or if Ignore_Modem_Status has been enabled, POSIX_IO.Open and POSIX_IO.Open_Or_Create shall return immediately without waiting for the connection.

If the object for which the terminal characteristics are set is not an asynchronous serial connection, some of the characteristics may be ignored. For example, if an attempt is made to set the baud rate on a network connection to a terminal on another host, the baud rate may or may not be set on the connection between that terminal and the machine to which it is directly connected.

The initial hardware control value after POSIX_IO.Open is implementation defined.

7.2.5 Local Modes

7.2.5.1 Synopsis

```
subtype Local_Modes is Terminal_Modes
   range Echo .. Send_Signal_For_BG_Output;
```

7.2.5.2 Description

Values of the subtype Local_Modes, shown in Table 7.5, shall describe the basic terminal local control modes.

If Echo mode is enabled, input characters shall be echoed back to the terminal. If Echo mode is disabled, input characters shall not be echoed.

If Echo_Erase and Canonical_Input modes are enabled, the Erase character shall cause the terminal to erase the last character in the current line from the display, if possible. If there is no character to erase, an implementation may echo an indication that there was no character to erase, or do nothing.

Table 7.5 – `Terminal_Modes` Values for Local Control Modes

Value	Meaning
`Canonical_Input`	Canonical input (erase and kill processing)
`Echo`	Enable echo
`Echo_Erase`	Echo Erase as an error-correcting backspace
`Echo_Kill`	Echo Kill
`Echo_LF`	Echo the `POSIX_Character` translation of `ASCII.LF`
`Enable_Signals`	Enable signals
`Extended_Functions`	Enable extended implementation-defined functions
`No_Flush`	Disable flush after interrupt, quit, or suspend
`Send_Signal_For_BG_Output`	Send `POSIX_Signals.Signal_Terminal_Output` for background output

If `Echo_Kill` and `Canonical_Input` modes are enabled, the Kill character shall either cause the terminal to erase the line from the display or shall echo the `POSIX_Character` translation of `ASCII.LF` character after the Kill character.

If `Echo_LF` and `Canonical_Input` modes are enabled, the `POSIX_Character` translation of `ASCII.LF` character shall be echoed even if `Echo` mode is disabled.

If `Canonical_Input` mode is enabled, canonical processing shall be enabled. Canonical processing enables the erase and kill edit functions and the assembly of input characters into lines delimited by NL, EOF, and EOL, as described in 7.1.0.9. If `Canonical_Input` mode is disabled, read requests shall be satisfied directly from the input queue. A read shall not be satisfied until at least `Minimum_Input_Count` `POSIX_Character`s have been received or the timeout value `Input_Time` has expired between `POSIX_Character`s. (See 7.1.0.10.)

If `Enable_Signals` mode is enabled, each input character shall be checked against the special control characters Interrupt, Quit, and Suspend (only if the Job Control option is supported). If an input character matches one of these control characters, the function associated with that character shall be performed. If `Enable_Signals` mode is disabled, no checking shall be done, and the characters shall be processed normally. Thus, these special input functions shall be possible only if `Enable_Signals` mode is enabled.

If `Extended_Functions` mode is enabled, implementation-defined functions shall be recognized from the input data. It is implementation defined how an enabled `Extended_Functions` mode interacts with `Canonical_Input`, `Enable_Signals`, `Enable_Start_Stop_Output`, or `Enable_Start_Stop_Input` modes. If `Extended_Functions` mode is disabled, then implementation-defined functions shall not be recognized, and the corresponding input characters shall be processed as described for `Canonical_Input`, `Enable_Signals`, `Enable_Start_Stop_Output`, and `Enable_Start_Stop_Input` modes.

If `No_Flush` mode is enabled, the normal flush of the input and output queues associated with the Interrupt, Quit, and Suspend (only if the Job Control option is supported) characters shall not be done.

If the Job Control option is supported: If `Send_Signal_For_BG_Output` mode is en-

abled POSIX_Signals.Signal_Terminal_Output shall be sent to the process group of a process that tries to write to its controlling terminal if it is not in the foreground process group for that terminal. This signal, by default, shall stop the members of the process group. If the process group is not stopped, the output generated by that process shall be sent to the current output stream. Processes that are blocking or ignoring POSIX_Signals.Signal_Terminal_Output are excepted; output produced by these processes shall be sent to the output stream, and the POSIX_Signals.-Signal_Terminal_Output signal shall not be sent.

The initial local control value after POSIX_IO.Open is implementation defined.

7.2.6 Retrieve and Define Terminal Modes and Bits per Character

7.2.6.1 Synopsis

```
type Terminal_Modes_Set is array (Terminal_Modes) of Boolean;
subtype Bits_Per_Character is Positive range implementation-defined;
function Terminal_Modes_Of
   (Characteristics: Terminal_Characteristics)
  return Terminal_Modes_Set;
procedure Define_Terminal_Modes
   (Characteristics: in out Terminal_Characteristics;
    Modes:           in      Terminal_Modes_Set);
function Bits_Per_Character_Of
   (Characteristics: Terminal_Characteristics)
  return Bits_Per_Character;
procedure Define_Bits_Per_Character
   (Characteristics: in out Terminal_Characteristics;
    Bits:            in      Bits_Per_Character);
```

7.2.6.2 Description

The type Terminal_Modes_Set shall be used to establish the terminal modes that will be set in a corresponding value of the type Terminal_Characteristics. For an object of type Terminal_Modes_Set, if the value in the parameter Modes indexed by a value of type Terminal_Modes is set to True, Define_Terminal_Modes shall enable that mode in the parameter Characteristics. It shall be disabled if the value is set to False. For a value of type Terminal_Modes_Set returned by Terminal_Modes_Of, the terminal mode shall be enabled in the parameter Characteristics if the value indexed by a value of type Terminal_Modes is set to True, and shall be disabled if the value is set to False.

The function Bits_Per_Character_Of shall be used to retrieve the number of bits per character set in the parameter Characteristics. The procedure Define_-Bits_Per_Character shall be used to set the number of bits per character in the parameter Characteristics.

7.2.6.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Upon Define_Terminal_Modes, Terminal_Modes_Of, Define_Bits_Per_Character, or Bits_Per_Character_Of, the object corresponding to the parameter Characteristics was not initialized via a call to Get_Terminal_Characteristics.

Define_Terminal_Modes or Terminal_Modes_Of may raise POSIX_Error for implementation-defined conditions.

Bits_Per_Character_Of may raise Constraint_Error if the parameter Characteristics contains a value that is out of range of the subtype Bits_Per_Character.

7.2.7 Baud Rate Subprograms

7.2.7.1 Synopsis

```
type Baud_Rate is
    (B0, B50, B75, B110, B134, B150, B200, B300, B600,
     B1200, B1800, B2400, B4800, B9600, B19200, B38400);
function Input_Baud_Rate_Of
   (Characteristics: Terminal_Characteristics)
  return Baud_Rate;
procedure Define_Input_Baud_Rate
   (Characteristics: in out Terminal_Characteristics;
    Input_Baud_Rate: in     Baud_Rate);
function Output_Baud_Rate_Of
   (Characteristics: Terminal_Characteristics)
  return Baud_Rate;
procedure Define_Output_Baud_Rate
   (Characteristics:  in out Terminal_Characteristics;
    Output_Baud_Rate: in     Baud_Rate);
```

7.2.7.2 Description

The functions Input_Baud_Rate_Of and Output_Baud_Rate_Of shall return the input and output data rate contained in the Terminal_Characteristics specified in the function call. The procedures Define_Input_Baud_Rate and Define_Output_Baud_Rate shall change the data rate values in the Terminal_Characteristics specified in the procedure call.

7.2.7.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The value contained in the parameter Characteristics was not initialized via a call to Get_Terminal_Characteristics.

Input_Baud_Rate_Of, Output_Baud_Rate_Of, Define_Input_Baud_Rate, and Define_Output_Baud_Rate may raise POSIX_Error for implementation-defined conditions.

7.2.8 Special Control Characters

7.2.8.1 Synopsis

```
type Control_Character_Selector is
   (EOF_Char, EOL_Char, Erase_Char, Interrupt_Char,
    Kill_Char, Quit_Char, Suspend_Char, Start_Char, Stop_Char);
function Special_Control_Character_Of
   (Characteristics: Terminal_Characteristics;
    Selector: Control_Character_Selector)
  return POSIX.POSIX_Character;
procedure Define_Special_Control_Character
   (Characteristics: in out Terminal_Characteristics;
    Selector:        in     Control_Character_Selector;
    Char:            in     POSIX.POSIX_Character);
procedure Disable_Control_Character
   (Characteristics: in out Terminal_Characteristics;
    Selector:        in     Control_Character_Selector);
```

7.2.8.2 Description

The special control character name and description in both canonical and noncanonical modes are shown in Table 7.6.

Table 7.6 – Special Control Character Usage

Selector	Canonical Mode	Noncanonical Mode	Description
EOF_Char	EOF		End-of-file character
EOL_Char	EOL		End-of-line character
Erase_Char	Erase		Erase character
Interrupt_Char	Interrupt	Interrupt	Interrupt character
Kill_Char	Kill		Kill character
Quit_Char	Quit	Quit	Quit character
Suspend_Char	Suspend	Suspend	Suspend character
Start_Char	Start	Start	Start character
Stop_Char	Stop	Stop	Stop character

The type Control_Character_Selector shall be used to designate control characters. Define_Special_Control_Character shall define the control character designated by the parameter Selector to the value of the parameter Char in the parameter Characteristics. Special_Control_Character_Of shall return the character associated with the control character designated by the parameter Selector in the parameter Characteristics. Special encodings of POSIX_Character shall be used to represent control characters only when the control characters do not exist in the POSIX_Character set.

Implementations that do not support the Job Control option may ignore the Suspend control character value and deliver the value to the program.

Implementations that do not support changing the Start and Stop characters may ignore these character values but shall return the value used when called.

The Disable_Control_Character procedure shall disable the control character. In other words, no input data shall be recognized as the disabled special character.

The initial values of all control characters are implementation defined.

7.2.8.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Inappropriate_IO_Control_Operation

Upon Special_Control_Character_Of, the special control character is disabled.

Operation_Not_Implemented

The operation is not supported on this device or by this system.

Invalid_Argument

The value contained in the parameter Characteristics was not initialized via a call to Get_Terminal_Characteristics.

No exceptions are specified by this standard for Define_Special_Control_Character.

Define_Special_Control_Character may raise POSIX_Error for implementation-defined conditions.

7.2.9 Noncanonical Controls

7.2.9.1 Synopsis

```
function Input_Time_Of
   (Characteristics: Terminal_Characteristics)
  return Duration;
procedure Define_Input_Time
   (Characteristics: in out Terminal_Characteristics;
    Input_Time:      in     Duration);
function Minimum_Input_Count_Of
   (Characteristics: Terminal_Characteristics)
  return Natural;
procedure Define_Minimum_Input_Count
   (Characteristics:     in out Terminal_Characteristics;
    Minimum_Input_Count: in     Natural);
```

7.2.9.2 Description

Define_Input_Time shall set the input time value used in noncanonical mode. (See 7.1.0.10.) Input_Time_Of shall retrieve the input time value from the control structure.

Define_Minimum_Input_Count shall set the input count value used in noncanonical mode. (See 7.1.0.10.) Minimum_Input_Count_Of shall retrieve the input count value from the control structure.

7.2.9.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The value contained in the parameter `Characteristics` was not initialized via a call to `Get_Terminal_Characteristics`.

`Input_Time_Of`, `Define_Input_Time`, `Minimum_Input_Count_Of`, and `Define_Minimum_Input_Count` may raise `POSIX_Error` for implementation-defined conditions.

7.2.10 Line Control Operations

7.2.10.1 Synopsis

```
procedure Send_Break
   (File:         in POSIX_IO.File_Descriptor;
    The_Duration: in Duration := 0.0);
procedure Drain
   (File:           in POSIX_IO.File_Descriptor;
    Masked_Signals: in POSIX.Signal_Masking := POSIX.RTS_Signals);
type Queue_Selector is
   (Received_But_Not_Read, Written_But_Not_Transmitted, Both);
procedure Discard_Data
   (File:     in POSIX_IO.File_Descriptor;
    Selector: in Queue_Selector);
type Flow_Action is
   (Suspend_Output, Restart_Output, Transmit_Stop, Transmit_Start);
procedure Flow
   (File:   in POSIX_IO.File_Descriptor;
    Action: in Flow_Action);
```

7.2.10.2 Description

If the terminal is using asynchronous serial data transmission, the procedure `Send_Break` shall cause the transmission of a continuous stream of zero-valued bits for the parameter `The_Duration`. If `The_Duration` is zero, it shall cause transmission of zero-valued bits for at least `0.25` seconds and not more than `0.5` seconds.

If the terminal is not using asynchronous serial data transmission, it is implementation defined whether the `Send_Break` procedure sends data to generate a break condition or returns without taking any action.

The `Drain` procedure shall wait until all output written to the object referred to by the parameter `File` has been transmitted.

The `Discard_Data` procedure shall discard data written to the object referred to by the parameter `File`, based on the value of the parameter `Selector`:

— If `Written_But_Not_Transmitted`, data written to the object designated by the parameter `File` but not transmitted shall be discarded.

— If `Received_But_Not_Read`, data received from the object designated by the parameter `File` but not read shall be discarded.

— If Both, data both transmitted to and received from the object designated by the parameter File shall be discarded.

The Flow procedure shall suspend transmission or reception of data on the object referred to by the parameter File, depending on the value of parameter Action:

— If Suspend_Output, output shall be suspended. In other words, output characters will not be sent to the output object when requested. They shall be held in an internal buffer, and/or the write will block.

— If Restart_Output, suspended output will be restarted.

— If Transmit_Stop, the system shall transmit a Stop character, which is intended to cause the terminal device to stop transmitting data to the system.

— If Transmit_Start, the system shall transmit a Start character, which is intended to cause the terminal device to start transmitting data to the system.

The default upon open of a terminal file is that neither its input nor its output is suspended.

The parameter Masked_Signals in the operation Drain specifies which signals shall be masked during the operation. (See 2.4.1.6.)

7.2.10.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor

Upon Send_Break, Drain, Discard_Data, or Flow, the parameter File is not a valid POSIX_IO.File_Descriptor.

Inappropriate_IO_Control_Operation

Upon Send_Break, Drain, Discard_Data, or Flow, the object designated by the parameter File is not a terminal.

Interrupted_Operation

Upon Drain, the operation was interrupted by a signal.

7.2.11 Foreground Process Group ID

7.2.11.1 Synopsis

```
function Get_Process_Group_ID
   (File: POSIX_IO.File_Descriptor)
  return POSIX_Process_Identification.Process_Group_ID;
procedure Set_Process_Group_ID
   (File:     in POSIX_IO.File_Descriptor;
    Group_ID: in POSIX_Process_Identification.Process_Group_ID);
```

7.2.11.2 Description

If the Job Control option is supported: `Get_Process_Group_ID` shall return the value of the process group ID of the foreground process group associated with the terminal. The function is allowed for a process that is in a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

If the Job Control option is not supported for this implementation, the function either shall be supported as defined or shall raise `POSIX_Error` with error code `Operation_Not_Implemented`.

If the Job Control option is supported: If the process has a controlling terminal, the `Set_Process_Group_ID` shall set the foreground process group ID associated with the terminal to the parameter `Group_ID`. The parameter `File` shall be the controlling terminal of the calling process, and the controlling terminal shall be currently associated with the session of the calling process. The value of the parameter `Group_ID` has to match a process group ID of a process in the same session as the calling process.

7.2.11.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Operation_Not_Implemented`

The implementation does not support the Job Control option.

`Bad_File_Descriptor`

The parameter `File` is not a valid `POSIX_IO.File_Descriptor`.

`Inappropriate_IO_Control_Operation`

(1) Upon `Get_Process_Group_ID`, the calling process does not have a controlling terminal, or the object designated by the parameter `File` is not the controlling terminal.

(2) Upon `Set_Process_Group_ID`, the calling process does not have a controlling terminal, or the object designated by the parameter `File` is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

`Invalid_Argument`

Upon `Set_Process_Group_ID`, the value of the parameter `Group_ID` is not a valid process group ID.

7.2.12 Generate Terminal Pathname

7.2.12.1 Synopsis

```
function Get_Controlling_Terminal_Name return POSIX.Pathname;
```

7.2.12.2 Description

The function `Get_Controlling_Terminal_Name` shall return a value of type `POSIX.Pathname` that shall refer to the current controlling terminal of the current process. `Get_Controlling_Terminal_Name` shall return an empty `Pathname` if the pathname that would refer to the controlling terminal cannot be determined or if the function is unsuccessful.

Access to the file is not guaranteed, even if `Get_Controlling_Terminal_Name` returns a valid pathname.

7.2.12.3 Error Handling

No exceptions are specified by this standard for `Get_Controlling_Terminal_Name`.

`Get_Controlling_Terminal_Name` may raise `POSIX_Error` for implementation-defined conditions.

Section 8: Language-Specific Services for Ada

This section provides additional interpretation within a POSIX context for certain features of the Ada language. Additional interpretation is provided only in cases where the Ada RM {1} permits implementation definition.

8.1 Interoperable Ada I/O Services

This section specifies certain aspects of I/O in the Ada binding to POSIX. Previous sections of this standard have provided an Ada binding to the I/O facilities of the POSIX interface. This section will provide a POSIX interpretation of the Ada I/O facilities required by Annex A of the Ada RM {1}. This interpretation does not attempt to duplicate the capabilities of the POSIX I/O model. Instead, it interprets the Ada model within the constraints of the POSIX model. A secondary objective is to provide interoperability (the ability to interchange data) between the Ada and POSIX I/O models for a useful subset of I/O operations.

To accomplish these objectives, this standard requires that a POSIX-conforming Ada compiler, runtime system, and libraries conform to additional requirements beyond those of Annex A of the Ada RM {1}.

8.1.1 Requirements

8.1.1.1 Character Set

The implementation shall provide a correspondence between the Ada type `Standard.Character` and `POSIX.POSIX_Character`. This correspondence shall have the characteristics described in 2.4.2.

8.1.1.2 Form Parameter

The `Form` parameter for Ada Text I/O operations shall be a string with the following syntax, written in the notation used within the Ada RM {1}:

form_image::=
[*field_name*_identifier => field_value {, *field_name*_identifier => field_value}]

field_value::= *field_value*_identifier | integer

The required values of *field_name*_identifier and field_value are described in the paragraphs that follow. The implementation is permitted to add alternatives to the syntax of field_value in order to support implementation-defined fields. All field names and field values shall be insensitive to case.

Implementations are permitted to define additional fields of the `Form` parameter as long as they conform to the syntax given above. The exception `IO_Exceptions.Use_Error` shall be raised if the `Form` parameter includes a field name or a field value that is not defined by the implementation or by this standard. The `Use_Error` exception shall be raised if a particular field name appears more than once within a `Form` parameter.

File permissions

The field names are `Owner`, `Group`, and `Other`. The allowed field values are `None`, `Read`, `Write`, or `Execute`, or any combination of the three latter

names separated by underscores (for example, `Read_Write`). These fields may be used only with the `Create` operation; use with the `Open` operations shall raise `Use_Error`. The fields shall set the access permissions for the created file. If a field is specified, the resulting set of permissions shall be the logical intersection of those specified in the parameter and those granted by `POSIX_Permissions.Get_Allowed_Process_Permissions`. If a field is not specified, the default value shall be `Read_Write_Execute`, which results in the set of permissions specified by the current value of `POSIX_-Permissions.Get_Allowed_Process_Permissions`.

Appending to the file

The field name is `Append`. The field value is either `True` or `False`. If not specified, the default value is `False`. This field may be used only with the `Open` operation. Use with the `Create` operation shall raise `Use_Error`. Use of `Append => True` means that any output shall be written to the end of the named external file.

Blocking or nonblocking I/O

The field name is `Blocking`. Potential field values are the images of the enumeration literals of type `POSIX.Blocking_Behavior`, *i.e.*, `Tasks` `Program` or `Special`. (See 2.4.1.5.) An implementation, however, is not required to support all of the enumerated behaviors. The supported behaviors are those enumerated in the subtype `POSIX.Text_IO_Blocking_Behavior`. The default value is the image of the constant `POSIX.IO_Blocking_-Behavior`. Use of `Blocking => Tasks`, when supported, means that the calling task, but not others, shall wait for the completion of any I/O operation on the file. Use of `Blocking => Program`, when supported, means that all tasks within the active partition shall wait for the completion of an I/O operation on the file. Use of `Blocking => Special` shall have an implementation-defined effect. Use of an unsupported value shall raise `Use_Error`. The file objects returned by `Text_IO.Standard_Input` and `Text_IO.Standard_Output` shall behave as if they were opened with the default value of `Blocking`.

How characters are read from the keyboard

The field name is `Terminal_Input`. The field value is either `Lines` or `Characters`. If not specified, the default value is `Lines`. If the value `Lines` is selected, the input terminal shall be processed in canonical mode. If the value `Characters` is selected, noncanonical terminal input shall be performed with `Minimum_Input_Count = 0` and `Input_Time = 0.0`. `Terminal_Input` shall have no effect if the file is not opened for input or if the file is not opened on a terminal. (See 7.1.0.10.)

Creation of FIFO files

The field name is `File_Structure`. The field value is either `Regular` or `FIFO`. If not specified, the default value is `Regular`. Use of this field is permitted only with the `Create` operation. Use with the `Open` operation shall cause the exception `Use_Error` to be raised. Use of `File_Structure => FIFO` means that the file to be created shall be a named FIFO file.

Access to open POSIX files

The field name is `File_Descriptor`. The field value is a sequence of characters that would denote a valid file descriptor value if used as the param-

eter for `POSIX_IO.File_Descriptor'Value`. Use of this option means that the opened Ada file object becomes associated with the external file denoted by the given file descriptor. The association continues until the Ada file object is closed. The operation shall not change the state of the file. If the integer number does not correspond to an open file descriptor, or if the modes are incompatible, then `Use_Error` shall be raised. During the period that the Ada file is open, the execution of any POSIX I/O or file operations, except for file locking (see 6.2), on the file descriptor will have unspecified results on the contents of the external file. Subsequent use of the `Text_IO.Name` function on this file shall raise `Use_Error`. This field may be used only with `Open`; use with `Create` shall raise `Use_Error`.

NOTE: This mechanism permits access to the standard error file, `POSIX_IO.Standard_Error`.

Treatment of terminators

The field name is `Page_Terminators`. The field value is either `True` or `False`. The default value is `True`. Use of `Page_Terminators => True` means that the external representation of the terminators defined in A.10 (7-8) of the Ada RM {1} is implementation defined. Use of `Page_Terminators => False` means that the external file shall contain no page terminators. In this case, upon output, the line terminator shall be represented in the external file by the character mapped to `ASCII.LF`; the page terminator shall be omitted from the external file; and the file terminator shall be represented by physical end of file. Upon input of a file with `Page_Terminators => False`, any occurrence of the character mapped to `ASCII.FF` shall not be interpreted as a page terminator, but shall instead result in the input of the `ASCII.FF` character. Upon output of a file with `Page_Terminators => False`, an explicit call to `Text_IO.New_Page` shall raise `Use_Error`, and an explicit call to `Text_IO.Set_Line` when the current line number exceeds the value specified by the `To` parameter shall raise `Use_Error`.

NOTE: The accompanying rationale describes the effects upon other subprograms of `Text_IO`. They are not specified here because they are derived from interpretation of the Ada standard.

The file objects returned by `Text_IO.Standard_Input` and `Text_IO.Standard_Output` shall operate as if they were opened with `Page_Terminators => False`.

8.1.1.3 Buffer Flushing

The implementation of Ada I/O operations may provide buffering in addition to that provided by the underlying operating system. The package `POSIX_Supplement_to_Ada_IO` provides operations that the application may invoke in order to flush output buffers to the operating system.

8.1.1.4 Additional Interpretation of the Ada Standard

Implementations shall conform to the following interpretations of the referenced sections of Annex A of the Ada RM {1}. In some cases, these interpretations require specific implementations of items that otherwise would be permitted by the the Ada RM {1} to be implementation dependent.

A.7 (1) of the Ada RM {1}

The identical set of external files shall be accessible via the POSIX I/O operations and by the Ada I/O operations, although the results of such access may be implementation defined except as noted elsewhere in this section. The file names used by Ada I/O operations and by POSIX I/O operations shall be interconvertible by applying the character mapping discussed in 2.4.2. The Ada `Form` parameter shall be interpreted as described elsewhere in this section.

A.7 (2) of the Ada RM {1}

The term "external file" shall be interpreted as referring to a POSIX file (defined in 2.2.2.64).

A.7 (6) of the Ada RM {1}

Upon the termination of an Ada active partition, all Ada file objects that were opened by the active partition and that remain open shall be closed, and any output buffers that the implementation has associated with the file objects shall be flushed. This action shall occur after the completion of the Ada main subprogram (if any) and after the termination of any tasks dependent upon library units but prior to the termination of the POSIX process that corresponds to the Ada active partition. Ada programs that are abandoned due to the propagation of an unhandled exception are subject to this interpretation.

A.8.2 (22) of the Ada RM {1}

In general, the `Name` function shall return the absolute pathname of a file. If `Text_IO.Standard_Input` or `Text_IO.Standard_Output` is directed to a terminal device, then the `Text_IO.Name` function shall return a value that is identical to the result of applying the character set mapping to the value returned by `POSIX_IO.Get_Terminal_Name`. Use of the `Name` function under circumstances where the name cannot be determined shall raise `Use_Error`. These circumstances include applying `Name` to a file opened using the `File_Descriptor` field of the `Form` parameter and applying `Name` to a file that has been redirected to a pipe or other facility so that a pathname or terminal device name cannot be determined.

A.10 (2) of the Ada RM {1}

The characters appearing at the Ada I/O interface shall correspond to the characters appearing in the external file according to the character set mapping described in 2.4.2.

A.10 (6) of the Ada RM {1}

The standard input file, the standard output file, and the standard error file referenced in this paragraph shall be interpreted as the POSIX standard input, standard output, and standard error files, respectively.

A.13 (13) of the Ada RM {1}

The phrases "input character sequence" and "value input" shall be interpreted as referring to the characters and values resulting from applying the character mapping described in 2.4.2 to the data appearing in the external file.

8.1.1.5 Error Handling

The implementation shall raise all exceptions required by Annex A of the Ada RM {1}. In addition, the implementation shall raise the exceptions listed in the following paragraphs.

`IO_Exceptions.Use_Error` shall be raised under the following circumstances:

— `Text_IO.Open` or `Text_IO.Create` is called when the `Form` parameter contains a field name or field value that is not defined by the implementation or by the standard.

— `Text_IO.Open` or `Text_IO.Create` is called when a particular field name appears more than once in the `Form` parameter.

— `Text_IO.Open` or `Text_IO.Create` is called when an unsupported value for `Blocking` appears in the `Form` parameter.

— `Text_IO.Open` is called when the `Form` parameter contains an `Owner` field, a `Group` field, an `Other` field, or a `FIFO` field.

— `Text_IO.Open` is called when the value given for the `File_Descriptor` field does not correspond to an open file descriptor or if the modes are incompatible.

— `Text_IO.Create` is called when the `Form` parameter contains an `Append` field.

— `Text_IO.Create` is called when the `Form` parameter contains a `File_Descriptor` field.

— `Text_IO.Name` is attempted upon an Ada file object that was opened with the `File_Descriptor` field specified within the `Form` parameter, or the operation is attempted upon a file that is redirected to a pipe or other facility so that a pathname or terminal device name cannot be determined.

— `Text_IO.New_Page` is performed on an output file opened with `Page_Terminators => False`.

— `Text_IO.Set_Line` is performed on an output file opened with `Page_Terminators => False`, and the value specified by the `To` parameter is less than the current line number.

8.1.2 Additional Interpretation of the POSIX.1 Standard

Execution of the `Text_IO.Open` and `Text_IO.Create` operations shall never result in the Ada file object becoming associated with the controlling terminal of the POSIX process associated with the active partition. The application may associate a file descriptor for a controlling terminal with an object of type `Text_IO.File_Type` by using the `File_Descriptor` field of the `Text_IO.Open` `Form` parameter. The functionality (see 7.1.0.6) obtained in this manner is implementation defined.

8.2 Package `POSIX_Supplement_to_Ada_IO`

The package `POSIX_Supplement_to_Ada_IO` provides supplementary facilities for applications programs within a POSIX context that perform input and output operations using the Ada model of I/O provided by Annex A of the Ada RM {1}.

```
with POSIX,
     POSIX_IO,
     POSIX_Permissions,
     IO_Exceptions,
     Text_IO;
package POSIX_Supplement_to_Ada_IO is
   -- 8.2.1 Parse Form Values
   type File_Structure_Values is (Regular, FIFO);
   type Terminal_Input_Values is (Lines, Characters);
   type Possible_File_Descriptor (Valid:  Boolean := False) is
      record
        case Valid is
          when True =>
            Descriptor:  POSIX_IO.File_Descriptor;
          when False => null;
        end case;
      end record;
   type Form_Values_for_Open is
      record
        Append:           Boolean := False;
        Blocking:         POSIX.Text_IO_Blocking_Behavior
          := POSIX.IO_Blocking_Behavior;
        Terminal_Input:   Terminal_Input_Values := Lines;
        Page_Terminators: Boolean := True;
        File_Descriptor:  Possible_File_Descriptor;
      end record;
   type Form_Values_for_Create is
      record
        Permission_Mask:  POSIX_Permissions.Permission_Set
          := POSIX_Permissions.Access_Permission_Set;
        Blocking:         POSIX.Text_IO_Blocking_Behavior
          := POSIX.IO_Blocking_Behavior;
        Terminal_Input:   Terminal_Input_Values := Lines;
        File_Structure:   File_Structure_Values := Regular;
        Page_Terminators: Boolean := True;
      end record;
   function Form_String (Val: Form_Values_for_Open) return String;
   function Form_Value (Str:  String) return Form_Values_for_Open;
   function Form_String (Val: Form_Values_for_Create) return String;
   function Form_Value (Str:  String) return Form_Values_for_Create;
   -- 8.2.2 Flush Files
   procedure Flush_All;
   procedure Flush_Text_IO (File: in Text_IO.File_Type);
   generic
      type File_Type is limited private;
   procedure Flush_Sequential_IO (File: File_Type);
   generic
      type File_Type is limited private;
   procedure Flush_Direct_IO (File: File_Type);

   Use_Error:  exception renames IO_Exceptions.Use_Error;
end POSIX_Supplement_to_Ada_IO;
```

8.2.1 Parse Form Values

8.2.1.1 Synopsis

```
   type File_Structure_Values is (Regular, FIFO);
```

```
type Terminal_Input_Values is (Lines, Characters);
type Possible_File_Descriptor (Valid:  Boolean := False) is
   record
     case Valid is
       when True =>
         Descriptor:  POSIX_IO.File_Descriptor;
       when False => null;
     end case;
   end record;
type Form_Values_for_Open is
   record
     Append:           Boolean := False;
     Blocking:         POSIX.Text_IO_Blocking_Behavior
       := POSIX.IO_Blocking_Behavior;
     Terminal_Input:   Terminal_Input_Values := Lines;
     Page_Terminators: Boolean := True;
     File_Descriptor:  Possible_File_Descriptor;
   end record;
type Form_Values_for_Create is
   record
     Permission_Mask:  POSIX_Permissions.Permission_Set
       := POSIX_Permissions.Access_Permission_Set;
     Blocking:         POSIX.Text_IO_Blocking_Behavior
       := POSIX.IO_Blocking_Behavior;
     Terminal_Input:   Terminal_Input_Values := Lines;
     File_Structure:   File_Structure_Values := Regular;
     Page_Terminators: Boolean := True;
   end record;
function Form_String (Val: Form_Values_for_Open) return String;
function Form_Value (Str:  String) return Form_Values_for_Open;
function Form_String (Val: Form_Values_for_Create) return String;
function Form_Value (Str:  String) return Form_Values_for_Create;
```

8.2.1.2 Description

The overloaded `Form_String` and `Form_Value` functions provide capability for constructing and parsing a `Form` parameter. The two record types `Form_Values_for_Open` and `Form_Values_for_Create` describe the allowed form values for `Open` and `Create` operations, respectively. The function `Form_String` shall return a value of type `String` representing a legal value for the `Form` parameter, as specified in 8.1.1.2. Values for the `Form` parameter shall be obtained from the parameter `Val`. The function `Form_Value` shall parse the string in the parameter `Str` and return a value of type `Form_Values_for_Open` or `Form_Values_for_Create`.

The `Form_Value` function shall ignore fields within the `Form` string that begin with field names that are not specified by this standard.

Implementors may supply a package containing identically named functions to construct and parse implementation-defined fields within the `Form` parameter.

8.2.1.3 Error Handling

`POSIX_Error` shall be raised, with error code `Invalid_Argument`, by `Form_Value` if a field does not conform to the specified syntax, if a nonstandard field value is associated with a standard field name, or if a particular field is specified more than once.

No exceptions are specified by this standard for `Form_String`.

8.2.2 Flush Files

8.2.2.1 Synopsis

```
procedure Flush_All;
procedure Flush_Text_IO (File: in Text_IO.File_Type);
generic
   type File_Type is limited private;
procedure Flush_Sequential_IO (File: File_Type);
generic
   type File_Type is limited private;
procedure Flush_Direct_IO (File: File_Type);
```

8.2.2.2 Description

Execution of `Flush_Text_IO` upon a value of type `File_Type` with mode `Out_File` shall result in the output buffers being flushed to the operating system.

Execution of `Flush_Text_IO` upon a value of type `File_Type` with mode `In_File` shall have no effect.

The generic procedure `Flush_Sequential_IO` should be instantiated with the generic formal type `File_Type` being matched by the type `File_Type` produced via an instantiation of the standard package `Sequential_IO`. When the instantiation is called with a value for the parameter `File` whose file mode is `Out_File`, the implementation shall flush all buffers associated with the given parameter to the operating system.

The generic procedure `Flush_Direct_IO` should be instantiated with the generic formal type `File_Type` being matched by the type `File_Type` produced via an instantiation of the standard package `Direct_IO`. When the instantiation is called with a value for the parameter `File` whose file mode is `Out_File`, the implementation shall flush all buffers associated with the given parameter to the operating system.

Execution of an instance of `Flush_Sequential_IO` or `Flush_Direct_IO` that was instantiated upon a file type with mode `In_File` shall have no effect.

Execution of instances of `Flush_Sequential_IO` or `Flush_Direct_IO` that are not as specified in the preceding paragraphs are implementation dependent.

Execution of `Flush_All` shall have the same effect as executing the appropriate individual flush operation upon each open Ada file.

8.2.2.3 Error Handling

No exceptions are specified by this standard for any of these operations.

Section 9: System Databases

This section contains two packages that provide access to the POSIX.1 system databases. The Group Database contains for each group a listing of group name, associated group ID, and a list of group members. The User Database contains a listing of users, including login name, user and group IDs, initial working directory, and initial user program.

9.1 Package POSIX_User_Database

This package provides the types and operations on the POSIX User Database. The User Database is the POSIX facility for storing information about users of the system, including the user ID, the user name, the primary group ID associated with the user, and the initial working directory and initial program. If the initial program in the User Database is null, the system default is used as the initial program. The interpretation of a null initial working directory is unspecified.

There are two sets of operations in POSIX_User_Database. One set retrieves information from a given value of type User_Database_Item, and the other set retrieves values of the type User_Database_Item.

```
with POSIX,
     POSIX_Process_Identification;
package POSIX_User_Database is
    -- 9.1.1 Access Contents of a User Database Item
    type User_Database_Item is private;
    function User_Name_Of (DB_Item: User_Database_Item)
      return POSIX.POSIX_String;
    function User_ID_Of (DB_Item: User_Database_Item)
      return POSIX_Process_Identification.User_ID;
    function Group_ID_Of (DB_Item: User_Database_Item)
      return POSIX_Process_Identification.Group_ID;
    function Initial_Directory_Of (DB_Item: User_Database_Item)
      return POSIX.POSIX_String;
    function Initial_Program_Of (DB_Item: User_Database_Item)
      return POSIX.POSIX_String;
    -- 9.1.2 Access User Database Items
    function Get_User_Database_Item
       (UID: POSIX_Process_Identification.User_ID)
      return User_Database_Item;
    function Get_User_Database_Item
       (Name: POSIX.POSIX_String)
      return User_Database_Item;  private

    implementation-defined
end POSIX_User_Database;
```

9.1.1 Access Contents of a User Database Item

9.1.1.1 Synopsis

```
    type User_Database_Item is private;
    function User_Name_Of (DB_Item: User_Database_Item)
      return POSIX.POSIX_String;
    function User_ID_Of (DB_Item: User_Database_Item)
```

```
    return POSIX_Process_Identification.User_ID;
  function Group_ID_Of (DB_Item: User_Database_Item)
    return POSIX_Process_Identification.Group_ID;
  function Initial_Directory_Of (DB_Item: User_Database_Item)
    return POSIX.POSIX_String;
  function Initial_Program_Of (DB_Item: User_Database_Item)
    return POSIX.POSIX_String;
```

9.1.1.2 Description

The User Database contains the name of the user, the user ID, the group ID, and pathnames for initial default directory and default program.

`User_Name_Of` shall return the textual name of the user associated with the parameter `DB_Item`.

`User_ID_Of` shall return the user ID associated with the value of the parameter `DB_Item`.

`Group_ID_Of` shall return the primary group membership of the user that is associated with the value of the parameter `DB_Item`.

`Initial_Directory_Of` shall return the pathname to the home directory of the user that is associated with the value of the parameter `DB_Item`. If this operation returns a null `POSIX.POSIX_String`, the meaning is implementation defined.

`Initial_Program_Of` shall return the pathname of the initial program of the user that is associated with the value of the parameter `DB_Item`. If the operation returns a null `POSIX.POSIX_String`, the system-default user program shall be used.

9.1.1.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The `DB_Item` parameter is not valid. The caller has to obtain a User Database item from one of the access operations described in 9.1.2 before calling any of these operations.

9.1.2 Access User Database Items

9.1.2.1 Synopsis

```
  function Get_User_Database_Item
     (UID: POSIX_Process_Identification.User_ID)
    return User_Database_Item;
  function Get_User_Database_Item
     (Name: POSIX.POSIX_String)
    return User_Database_Item;
```

9.1.2.2 Description

These operations define how to obtain a `User_Database_Item`. One form of `Get_User_Database_Item` shall return the value associated with the parameter `UID`. The other form of `Get_User_Database_Item` shall return the value associated with the parameter `Name`.

9.1.2.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

There is no User Database item corresponding to the given `Name` or `UID` parameter.

9.2 Package `POSIX_Group_Database`

This package provides the types and operations on the POSIX Group Database. The Group Database is the POSIX facility for storing information about the groups on the system, including group ID, the group name, and a list of user names who are members of the group. Every user is a member of at least one group.

There are three sets of operations in `POSIX_Group_Database`. One set retrieves information from an object of type `Group_Database_Item`, the second supports iterating through the list of user names, and the third retrieves values of the type `Group_Database_Item`.

```
with POSIX,
     POSIX_Process_Identification;
package POSIX_Group_Database is
    -- 9.2.1 Access Contents of a Group Database Item
    type Group_Database_Item is private;
    type Group_ID_List is private;
    function Group_Name_Of (DB_Item: Group_Database_Item)
      return POSIX.POSIX_String;
    function Group_ID_Of (DB_Item: Group_Database_Item)
      return POSIX_Process_Identification.Group_ID;
    function Group_ID_List_Of (DB_Item: Group_Database_Item)
      return Group_ID_List;
    -- 9.2.2 Access Elements of the Group Item List of Members
    generic
       with procedure Action
          (ID:   in POSIX.POSIX_String;
           Quit: in out Boolean);
    procedure For_Every_Member (List: in Group_ID_List);
    function Length (Member_List: Group_ID_List)
      return Natural;
    -- 9.2.3 Access Group Database Items
    function Get_Group_Database_Item
       (GID: POSIX_Process_Identification.Group_ID)
      return Group_Database_Item;
    function Get_Group_Database_Item
       (Name : POSIX.POSIX_String)
      return Group_Database_Item;

private
    implementation-defined
end POSIX_Group_Database;
```

9.2.1 Access Contents of a Group Database Item

9.2.1.1 Synopsis

```
type Group_Database_Item is private;
type Group_ID_List is private;
function Group_Name_Of (DB_Item: Group_Database_Item)
  return POSIX.POSIX_String;
function Group_ID_Of (DB_Item: Group_Database_Item)
  return POSIX_Process_Identification.Group_ID;
function Group_ID_List_Of (DB_Item: Group_Database_Item)
  return Group_ID_List;
```

9.2.1.2 Description

The Group Database contains the following information for each group: the group name, the group ID, and a list of group members. The type `Group_ID_List` is a private type with an operation to iterate over the members of the given group, as described in 9.2.3.

`Group_Name_Of` shall return the group name associated with the value of parameter `DB_Item`

`Group_ID_Of` shall return the group ID associated with the value of parameter `DB_Item`.

`Group_ID_List_Of` shall return the list of group members associated with the value of parameter `DB_Item`.

9.2.1.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The `DB_Item` parameter is not valid. The caller has to obtain a Group Database item from one of the access operations described in 9.2.3 before calling any of these operations.

9.2.2 Access Elements of the Group Item List of Members

9.2.2.1 Synopsis

```
generic
   with procedure Action
      (ID:   in POSIX.POSIX_String;
       Quit: in out Boolean);
procedure For_Every_Member (List: in Group_ID_List);
function Length (Member_List: Group_ID_List)
  return Natural;
```

9.2.2.2 Description

`Length` shall return the number of users that are members of the parameter `Member_List`. `Length` shall return zero if no users are members of the parameter `Member_List`. The generic procedure `For_Every_Member` provides a facility for iterating over the contents of a `Group_ID_List`. For each member of the parameter `List`, the actual procedure associated with `Action` in the instantiation shall be called, with a value of type `POSIX.POSIX_String` representing the name of the group member. The exact format of the string passed as the value of the parameter `ID` is not specified by this standard. It is acceptable for this string to be the string image of the group ID. If the parameter `List` contains no member (*i.e.*, `Length` returns zero), then the actual procedure associated with the formal parameter `Action` shall not be called.

9.2.2.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The `List` parameter is not valid.

9.2.3 Access Group Database Items

9.2.3.1 Synopsis

```
function Get_Group_Database_Item
   (GID: POSIX_Process_Identification.Group_ID)
  return Group_Database_Item;
function Get_Group_Database_Item
   (Name : POSIX.POSIX_String)
  return Group_Database_Item;
```

9.2.3.2 Description

These operations define how to obtain a value of the type `Group_Database_Item`. One form of `Get_Group_Database_Item` shall return the value of type `Group_Database_Item` associated with the parameter `GID`. The other form of `Get_Group_Database_Item` shall return the value associated with the parameter `Name`.

9.2.3.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

There is no Group Database item corresponding to the given `GID` or `Name` parameter.

Section 10: Data Interchange Format

This section position is reserved to preserve the correspondence of section numbers between this standard and POSIX.1, and to allow for the possibilit of a future Ada binding to the section on Data Interchange Format of POSIX.1.

Section 11: Synchronization

This section describes the services that this standard provides for process and task synchronization via counting semaphores, mutexes, and condition variables.

11.1 Package POSIX_Semaphores

This package provides access to services related to counting semaphores. Counting semaphores are a type of synchronization object that can be used for interprocess and intertask synchronization.

NOTE: Semaphores are provided in this standard primarily for interprocess communication. The preferred mechanisms for intertask synchronization are those provided by the Ada language, namely, protected objects and task entries.

The functionality described in this clause is optional. If the Semaphores option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this clause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this clause.

```
with POSIX,
     POSIX_IO,
     POSIX_Permissions;
package POSIX_Semaphores is
   -- 11.1.1 Semaphore and Semaphore Descriptor Types
   type Semaphore is limited private;
   type Semaphore_Descriptor is private;
   -- 11.1.2 Initialize an Unnamed Semaphore
   procedure Initialize
      (Sem:       in out Semaphore;
       Value:     in  Natural;
       Is_Shared: in  Boolean := False);
   function Descriptor_Of
      (Sem:       Semaphore)
     return Semaphore_Descriptor;
   -- 11.1.3 Finalize an Unnamed Semaphore
   procedure Finalize (Sem: in out Semaphore);
   -- 11.1.4 Create/Open a Named Semaphore
   function Open
      (Name:           POSIX.POSIX_String;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return Semaphore_Descriptor;
   function Open_Or_Create
      (Name:           POSIX.POSIX_String;
       Permissions:    POSIX_Permissions.Permission_Set;
       Value:          Natural;
       Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
      return Semaphore_Descriptor;
   -- 11.1.5 Close a Named Semaphore
   procedure Close (Sem: in out Semaphore_Descriptor);
   -- 11.1.6 Remove a Named Semaphore
   procedure Unlink_Semaphore (Name: in POSIX.POSIX_String);
```

```
   -- 11.1.7 Decrement a Semaphore
   procedure Wait
      (Sem:            in Semaphore_Descriptor;
       Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
   function Try_Wait
      (Sem: Semaphore_Descriptor)
     return Boolean;
   -- 11.1.8 Increment a Semaphore
   procedure Post (Sem: in Semaphore_Descriptor);
   -- 11.1.9 Get the Value of a Semaphore
   function Get_Value (Sem:  Semaphore_Descriptor)
     return Integer;
private
   implementation-defined
end POSIX_Semaphores;
```

11.1.1 Semaphore and Semaphore Descriptor Types

11.1.1.1 Synopsis

```
type Semaphore is limited private;
type Semaphore_Descriptor is private;
```

11.1.1.2 Description

The type Semaphore is used to represent semaphore objects. Semaphores may, but need not, be implemented within the file system, and open semaphores may use file descriptors. Thus, the use of semaphores may reduce the number of file descriptors available for other uses (see Open Files Maximum in 2.6.1).

Operating on copies of Semaphore objects (which may be created by returning function results of this type or by using the type as an **in out** parameter) results in undefined behavior.

This standard defines two kinds of semaphores: unnamed semaphores (see 11.1.2) and named semaphores (see 11.1.4).

Objects of type Semaphore_Descriptor are used as semaphore descriptors, that is, handles for semaphore objects. A semaphore descriptor is *valid* if it is associated with an open named semaphore or an initialized unnamed semaphore.

Copies of Semaphore_Descriptor objects shall always specify the same semaphore object. Operating on an object of type Semaphore_Descriptor after the corresponding semaphore object no longer exists results in undefined behavior, which might affect other processes sharing the same semaphore.

This standard specifies two attributes of semaphore objects:

Value

The Value attribute of a semaphore is an integer representing the state of the semaphore. This attribute is initialized when the semaphore is initialized (unnamed semaphore) or created (named semaphore). Three operations (Wait, Try_Wait, and Post) may be applied to semaphores and operate on the Value attribute. (See 11.1.7 and 11.1.8.)

Process Shared

The Process Shared attribute of a semaphore is a `Boolean` that indicates whether the semaphore is sharable between tasks in different processes. A task in any process that can access a sharable semaphore can safely operate upon it using the operations defined in this section. For unnamed semaphores the Process Shared attribute is initialized by the parameter `Is_Shared` (see 11.1.2). For named semaphores the Process Shared attribute is always set to `True` when the semaphore is created (see 11.1.4), that is, named semaphores are always sharable.

11.1.2 Initialize an Unnamed Semaphore

11.1.2.1 Synopsis

```
procedure Initialize
   (Sem:       in out Semaphore;
    Value:     in  Natural;
    Is_Shared: in  Boolean := False);
function Descriptor_Of
   (Sem:       Semaphore)
  return Semaphore_Descriptor;
```

11.1.2.2 Description

`Initialize` shall initialize the unnamed semaphore `Sem`, setting the Value attribute to `Value` and the Process Shared attribute to `Is_Shared`. Following a normal return from `Initialize`, the semaphore is *initialized*, meaning it can be used in subsequent calls to the operations defined in this section. This semaphore remains initialized until it is finalized. If `Initialize` is called on a semaphore that is already initialized (without an intervening `Finalize`), the effect is undefined.

NOTE: In order to provide the semantics specified above it will be necessary for implementations to ensure that the parameter `Sem` of `Initialize` is passed by reference.

If `Is_Shared` is `False`, then `Sem` is shared only among tasks of the same process; any task in this process can use `Sem` for performing the operations defined in this section. The use of `Sem` by tasks other than those created in the same process is undefined.

NOTE: Several processes may access a sharable unnamed semaphore, if it is allocated in shared memory.

`Descriptor_Of` shall return a descriptor for the semaphore object `Sem` that can later be used in semaphore operations. The semaphore corresponding to `Sem` is required to already be initialized, otherwise the effect is undefined. If `Descriptor_Of` is called again, by the same task or by another task within the same process, and `Sem` has not since been finalized, the same descriptor value shall be returned.

11.1.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The `value` parameter exceeds Semaphores Value Maximum.

No_Space_Left_On_Device
: A resource required to initialize the semaphore has been exhausted. The limit on semaphores (Semaphores Maximum) has been reached.

Operation_Not_Implemented
: Initialize is not supported by this implementation.

Operation_Not_Permitted
: The process lacks the appropriate privilege to initialize the semaphore.

11.1.3 Finalize an Unnamed Semaphore

11.1.3.1 Synopsis

```
procedure Finalize (Sem: in out Semaphore);
```

11.1.3.2 Description

Finalize is used to finalize the unnamed semaphore Sem. The effect of subsequent use of Sem is undefined until the semaphore is reinitialized by another call to Initialize.

NOTE: In order to provide the semantics specified above, it will be necessary for implementations to ensure that the parameter Sem of Finalize is passed by reference.

The effect of finalizing a semaphore on which other tasks are currently blocked is undefined.

11.1.3.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: Sem is not a properly initialized semaphore object.

Operation_Not_Implemented
: Finalize is not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Busy
: There are currently tasks blocked on Sem.

11.1.4 Create/Open a Named Semaphore

11.1.4.1 Synopsis

```
function Open
   (Name:           POSIX.POSIX_String;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
  return Semaphore_Descriptor;
function Open_Or_Create
   (Name:           POSIX.POSIX_String;
    Permissions:    POSIX_Permissions.Permission_Set;
    Value:          Natural;
    Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
   return Semaphore_Descriptor;
```

11.1.4.2 Description

`Open` and `Open_Or_Create` establish a connection between a named semaphore and a process. They return a semaphore descriptor that any task within the process can use to refer to the semaphore associated with the specified name. Following a normal return from `Open` or `Open_Or_Create`, the semaphore can be used in subsequent calls to the operations defined in this section. This semaphore remains usable by any task within this process until the semaphore is closed by `Close`, `POSIX_Unsafe_Process_Primitives.Exec`, `POSIX_Unsafe_Process_Primitives.Exec_Search`, or `POSIX_Process_Primitives.Exit_Process`.

`Open` shall open an already existing semaphore and shall fail if the named semaphore does not exist.

`Open_Or_Create` shall either open an existing semaphore, except as noted under `Exclusive` below, or shall create and open a named semaphore.

After the semaphore named `Name` has been created by `Open_Or_Create`, other tasks (whether in the same process or in different processes) can connect to the semaphore by calling `Open` or `Open_Or_Create` with the same value of `Name`.

The `Name` parameter specifies a string naming a semaphore object.

It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as parameters.

The `Name` parameter shall conform to the construction rules for a pathname. If `Name` begins with the slash character, then processes calling `Open` or `Open_Or_Create` with the same value of `Name` shall refer to the same semaphore object, as long as that name has not been removed by a call to `Unlink_Semaphore`. (See 11.1.6). If `Name` does not begin with the slash character, the effect is implementation defined. The interpretation of slash characters other than the leading slash character in `Name` is implementation defined.

The `Options` parameter may specify either the empty set or the following value:

`Exclusive`

> If `Options` includes `POSIX_IO.Exclusive`, `Open_Or_Create` shall fail if the named semaphore already exists. If the named semaphore does not already exist, it shall be created. The check for the existence of the semaphore and the creation of the semaphore if it does not exist shall be atomic with respect to other processes executing `Open_Or_Create` naming the same semaphore with `Exclusive` specified.

NOTE: This standard does not require semaphore creation to be atomic with respect to tasks, since atomicity of this operation is not required by POSIX.1.

If `Options` does not include `POSIX_IO.Exclusive` and the named semaphore already exists, `Open_Or_Create` shall open the named semaphore as if `Open` was called.

The effect of specifying an option other than `POSIX_IO.Exclusive` is unspecified.

When a semaphore is created, its Value attribute shall be initialized to `Value` and its Process Shared attribute shall be set to `True`. Valid initial values for semaphores must be less than or equal to Semaphores Value Maximum. When the operation does not create a new semaphore, the parameter `Value` shall be ignored.

The user ID of the semaphore shall be set to the effective user ID of the process, and the group ID of the semaphore shall be set to a system default group ID or to the effective group ID of the process.

The access permissions for the newly created semaphore shall be those for which the permission is `True` in both `Permissions` and the allowed process permission set.

`Open` or `Open_Or_Create` may be interruptible by the delivery of a signal. `Masked_Signals` specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

If a process makes multiple calls (even from different tasks within the same process) to `Open` or `Open_Or_Create` with the same value for `Name`, the same `Semaphore_Descriptor` value shall be returned for each such call, provided that there have been no calls to `Unlink_Semaphore` for this semaphore.

11.1.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: The named semaphore exists, and the permissions were denied; or the named semaphore does not exist, and permission to create the named semaphore was denied.

`File_Exists`
: `POSIX_IO.Exclusive` is specified, and the named semaphore already exists.

`Interrupted_Operation`
: `Open` or `Open_Or_Create` was interrupted by a signal.

`Invalid_Argument`
: `Open` and `Open_Or_Create` are not supported for `Name`. The implementation shall document under what circumstances this error may be returned.
`Value` is greater than Semaphores Value Maximum.

`Too_Many_Open_Files`
: Too many semaphore descriptors or file descriptors are currently in use by this process.

`Too_Many_Open_Files_In_System`
: Too many semaphores are currently open in the system.

`Filename_Too_Long`
: The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

No_Such_File_Or_Directory

Upon a call to Open, the named semaphore does not exist.

No_Space_Left_On_Device

There is insufficient space for the creation of the new named semaphore.

Operation_Not_Implemented

Open and Open_Or_Create are not supported by this implementation.

11.1.5 Close a Named Semaphore

11.1.5.1 Synopsis

```
procedure Close (Sem: in out Semaphore_Descriptor);
```

11.1.5.2 Description

The procedure Close shall only be called for named semaphores and is used to indicate that all tasks within the calling process are finished using the named semaphore specified by Sem.

Close shall deallocate (that is, make available for reuse by a subsequent Open or Open_Or_Create by this process) any system resources allocated by the system for use by this process for this semaphore. The effect of subsequent use of Sem is undefined. If the semaphore has not been removed by a call to Unlink_Semaphore, then Close shall have no effect on the state of the semaphore.

11.1.5.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Sem is not a valid semaphore descriptor.

Operation_Not_Implemented

Close is not supported by this implementation.

11.1.6 Remove a Named Semaphore

11.1.6.1 Synopsis

```
procedure Unlink_Semaphore (Name: in POSIX.POSIX_String);
```

11.1.6.2 Description

Unlink_Semaphore shall remove (*i.e.*, make inaccessible) the semaphore named by the string Name. If the semaphore specified by Name is currently referenced by other processes, then Unlink_Semaphore shall have no effect on the state of the semaphore. If one or more processes have the semaphore open when Unlink_Semaphore is called, removal of the semaphore shall be postponed until all references to the semaphore have been severed by calls to Close, POSIX_Unsafe_Process_Primitives.Exec, POSIX_Unsafe_Process_Primitives.Exec_Search,

or POSIX_Process_Primitives.Exit_Process. The call to Unlink_Semaphore shall not block the caller; it shall return immediately.

After Unlink_Semaphore is called, subsequent calls to Open or Open_Or_Create to recreate or reconnect to the semaphore shall refer to a new semaphore object.

11.1.6.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Permission_Denied
: Permission is denied to unlink the named semaphore.

Filename_Too_Long
: The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

No_Such_File_Or_Directory
: The named semaphore does not exist.

Operation_Not_Implemented
: Unlink_Semaphore is not supported by this implementation.

11.1.7 Decrement a Semaphore

11.1.7.1 Synopsis

```
procedure Wait
   (Sem:            in Semaphore_Descriptor;
    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
function Try_Wait
   (Sem: Semaphore_Descriptor)
  return Boolean;
```

11.1.7.2 Description

Wait performs the semaphore decrement operation (see 2.2.2.162) on the semaphore specified by Sem. If the semaphore Value is currently zero, then the calling task shall be blocked and shall not return from the call to Wait until either it decrements the semaphore or the call is interrupted by a signal.

Try_Wait shall decrement the semaphore specified by Sem and return True only if the semaphore Value is currently positive. Otherwise, Try_Wait shall return False, and the semaphore Value attribute shall remain unchanged.

When Wait returns or Try_Wait returns with a value of True, the semaphore Value has been decremented. The semaphore Value shall remain unchanged until Post is called and returns normally.

Wait may be interruptible by the delivery of a signal. Masked_Signals specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

It is implementation defined whether this operation is interruptible. If the operation is not interruptible, the Masked_Signals parameter may be ignored.

11.1.7.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Sem is not a valid semaphore descriptor.

Operation_Not_Implemented

Wait and Try_Wait are not supported by this implementation.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Deadlock_Avoided

The operation did not complete since a deadlock condition would have resulted.

Interrupted_Operation

A signal interrupted this operation. It shall be documented in the system documentation whether this error code is returned.

11.1.8 Increment a Semaphore

11.1.8.1 Synopsis

```
procedure Post (Sem: in Semaphore_Descriptor);
```

11.1.8.2 Description

Post performs the semaphore increment operation (see 2.2.2.163) on the semaphore specified by Sem. If the semaphore Value resulting from this operation is positive, then no tasks were blocked waiting to decrement the semaphore; the semaphore Value is simply incremented.

If the Value of the semaphore resulting from this operation is zero, then one of the tasks blocked waiting to decrement the semaphore shall be unblocked and allowed to return from its call to Wait.

If the Priority Process Scheduling option is supported: If more than one task is blocked waiting to decrement the semaphore, the task to be unblocked shall be selected in a manner appropriate to the scheduling policies and parameters in effect for the blocked tasks and their processes.

If the Priority Process Scheduling option is not supported, the selection of which task to unblock is unspecified.

11.1.8.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: `Sem` is not a valid semaphore descriptor.

`Operation_Not_Implemented`
: `Post` is not supported by this implementation.

11.1.9 Get the Value of a Semaphore

11.1.9.1 Synopsis

```
function Get_Value (Sem:  Semaphore_Descriptor)
  return Integer;
```

11.1.9.2 Description

`Get_Value` returns the Value attribute of the semaphore specified by `Sem` without affecting the state of the semaphore. The value returned represents an actual semaphore Value that occurred at some unspecified time during the call. This value may be different from the actual Value of the semaphore when `Get_Value` returns.

If the Value of the semaphore specified by `Sem` is zero, then `Get_Value` shall return either zero or a negative number whose absolute value represents the number of tasks waiting for the semaphore at an unspecified time during the call.

11.1.9.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: `Sem` is not a valid semaphore descriptor.

`Operation_Not_Implemented`
: `Get_Value` is not supported by this implementation.

11.2 Package `POSIX_Mutexes`

This package provides access to services related to mutexes. Mutexes are a type of synchronization object that can be used to provide mutual exclusion between tasks. Certain mutexes (see 11.2.4), when allocated in shared memory (see 12.4), can also be used to provide mutual exclusion between tasks in different processes.

NOTE: Mutexes and condition variables are provided in this standard primarily for synchronization between Ada tasks and C-language threads, in mixed-language applications. The preferred mechanisms for intertask synchronization are those provided by the Ada language, namely, protected objects and task entries.

The functionality described in this clause is optional. If the Mutexes option is not supported, the implementation may cause all calls to the explicitly declared operations

defined in this clause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this clause.*

```
with System;
package POSIX_Mutexes is
   -- 11.2.2 Mutex and Mutex Descriptor Types
   type Mutex is limited private;
   type Mutex_Descriptor is private;
   -- 11.2.3 Mutex Attributes Type
   type Attributes is private;
   procedure Initialize (Attr: in out Attributes);
   procedure Finalize (Attr: in out Attributes);
   -- 11.2.4 Mutex Process Shared Attribute
   function Get_Process_Shared (Attr: Attributes)
     return Boolean;
   procedure Set_Process_Shared
      (Attr: in out Attributes;
       Is_Shared: in Boolean:= False);
   -- 11.2.5 Mutex Locking Policy Attributes
   subtype Ceiling_Priority is Integer range implementation-defined;
   type Locking_Policy is range implementation-defined;
   No_Priority_Inheritance:  constant Locking_Policy:= implementation-defined;
   Highest_Blocked_Task:     constant Locking_Policy:= implementation-defined;
   Highest_Ceiling_Priority: constant Locking_Policy:= implementation-defined;
   procedure Set_Locking_Policy
      (Attr: in out Attributes;
       Locking: in Locking_Policy);
   function Get_Locking_Policy
      (Attr: Attributes)
     return Locking_Policy;
   procedure Set_Ceiling_Priority
      (Attr: in out Attributes;
       New_Ceiling: in Ceiling_Priority);
   function Get_Ceiling_Priority (Attr: Attributes)
     return Ceiling_Priority;
   -- 11.2.6 Initialize and Finalize a Mutex
   procedure Initialize
      (M: in out Mutex;
       Attr: in Attributes);
   procedure Initialize (M: in out Mutex);
   function Descriptor_Of (M: Mutex) return Mutex_Descriptor;
   procedure Finalize (M: in out Mutex);
   -- 11.2.7 Change the Ceiling Priority of a Mutex
   procedure Set_Ceiling_Priority
      (M:           in Mutex_Descriptor;
       New_Ceiling: in Ceiling_Priority;
       Old_Ceiling: out Ceiling_Priority);
   function Get_Ceiling_Priority (M: Mutex_Descriptor)
     return Ceiling_Priority;
   -- 11.2.8 Lock and Unlock a Mutex
   procedure Lock (M: in Mutex_Descriptor);
   function  Try_Lock (M: Mutex_Descriptor) return Boolean;
   procedure Unlock (M: in Mutex_Descriptor);

private
   implementation-defined
end POSIX_Mutexes;
```

11.2.1 Mutex Ownership

A task becomes the *owner* of a mutex specified by the mutex descriptor `M` (see 2.2.2.99) when any of the following occur:

— A call to `Lock` with parameter `M` returns normally, or

— A call to `Try_Lock` with parameter `M` returns the value `True`, or

— A call to `POSIX_Condition_Variables.Wait` with parameter `M` returns normally or with an exception (except as explicitly indicated otherwise for certain errors), or

— A call to `POSIX_Condition_Variables.Timed_Wait` with parameter `M` returns normally or with an exception (except as explicitly indicated otherwise for certain errors).

The task remains the owner of the mutex specified by the mutex descriptor `M` until it does one of the following:

— Executes `Unlock` with parameter `M`, or

— Is blocked in a call to `POSIX_Condition_Variables.Wait` with parameter `M`, or

— Is blocked in a call to `POSIX_Condition_Variables.Timed_Wait` with parameter `M`.

The implementation shall behave, at each instant in time visible to the contending tasks, as if there is at most one owner of any mutex.

A task that becomes the owner of a mutex is said to have *locked* the mutex, and the mutex is said to have become locked. When a task gives up ownership of a mutex it is said to have *unlocked* the mutex, and the mutex is said to have become unlocked.

The effect of aborting or terminating a task that has locked a mutex is implementation defined, and shall be documented for each implementation.

NOTE: The user is responsible for ensuring that all mutexes owned by a task are unlocked before the task terminates.

11.2.2 Mutex and Mutex Descriptor Types

11.2.2.1 Synopsis

```
type Mutex is limited private;
type Mutex_Descriptor is private;
```

11.2.2.2 Description

The type `Mutex` is used to represent mutex objects.

Operating on copies of `Mutex` objects (which may be created by returning function results of this type or by using the type as an **`in out`** parameter) results in undefined behavior.

Objects of type `Mutex_Descriptor` are used as mutex descriptors; that is, handles for mutex objects. A mutex descriptor is *valid* if it is associated with a properly initialized mutex; that is, the mutex descriptor is returned by a call to `Descriptor_Of`

with a parameter that has been used in a prior call to `Initialize` and not yet used in a call to `Finalize`.

Copies of `Mutex_Descriptor` objects shall always specify the same mutex. Operating on an object of type `Mutex_Descriptor` after the corresponding mutex object is finalized results in undefined behavior, which might affect other processes sharing the same mutex.

11.2.3 Mutex Attributes Type

11.2.3.1 Synopsis

```
type Attributes is private;
procedure Initialize (Attr: in out Attributes);
procedure Finalize (Attr: in out Attributes);
```

11.2.3.2 Description

Objects of type `Attributes` are used to specify a set of mutex creation attributes.

This standard defines several attributes of mutexes:

Process Shared
: Defined in 11.2.4.

Locking Policy
: Defined in 11.2.5.

Ceiling Priority
: Defined in 11.2.5.

Additional attributes, their default values, and the names of the associated operations to get and set those attribute values are implementation defined.

`Initialize` shall initialize the mutex attributes object `Attr` with the default value for all of the attributes defined by the implementation.

The effect of initializing an already initialized mutex attributes object is undefined.

After a mutex attributes object has been used to initialize one or more mutexes, any operation affecting the attributes object (including `Finalize`) does not affect any previously initialized mutexes.

`Finalize` shall finalize the mutex attributes object `Attr`; the object becomes, in effect, uninitialized. An implementation may cause `Finalize` to set `Attr` to an invalid value. A finalized mutex attributes object can be reinitialized using `Initialize`; the results of otherwise referencing the attributes object after it has been finalized are undefined.

11.2.3.3 Error Handling

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

Not_Enough_Space

Upon Initialize, insufficient memory exists to initialize the mutex attributes object.

Invalid_Argument

Upon Finalize, Attr does not specify a properly initialized mutex attributes object.

11.2.4 Mutex Process Shared Attribute

11.2.4.1 Synopsis

```
function Get_Process_Shared (Attr: Attributes)
  return Boolean;
procedure Set_Process_Shared
   (Attr: in out Attributes;
    Is_Shared: in Boolean:= False);
```

11.2.4.2 Description

The functionality described in this subclause is optional. If the Process Shared option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

The implementation shall provide a Boolean attribute called Process Shared and the associated operations Get_Process_Shared and Set_Process_Shared. This attribute indicates whether a newly initialized mutex can be shared between tasks in different processes having access to that mutex.

Set_Process_Shared sets the Process Shared attribute in an initialized mutex attributes object Attr to the value Is_Shared.

Get_Process_Shared returns the value of the Process Shared attribute of the mutex attributes object Attr.

If the Process Shared attribute is True in a given mutex attributes object, a mutex initialized with this object shall be *sharable*; that is, the mutex can be operated upon by any task that has access to the memory where the mutex is allocated (even tasks from different processes). Otherwise, the mutex object shall not be sharable, that is, it can be operated upon only by tasks created within the same process as the task that initialized the mutex. If tasks of other processes attempt to operate on such a mutex, the behavior is undefined.

The default value of the attribute shall be False.

11.2.4.3 Error Handling

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Attr does not specify a properly initialized mutex attributes object.

11.2.5 Mutex Locking Policy Attributes

11.2.5.1 Synopsis

```
subtype Ceiling_Priority is Integer range implementation-defined;
type Locking_Policy is range implementation-defined;
No_Priority_Inheritance:  constant Locking_Policy:= implementation-defined;
Highest_Blocked_Task:     constant Locking_Policy:= implementation-defined;
Highest_Ceiling_Priority: constant Locking_Policy:= implementation-defined;
procedure Set_Locking_Policy
   (Attr: in out Attributes;
    Locking: in Locking_Policy);
function Get_Locking_Policy
   (Attr: Attributes)
  return Locking_Policy;
procedure Set_Ceiling_Priority
   (Attr: in out Attributes;
    New_Ceiling: in Ceiling_Priority);
function Get_Ceiling_Priority (Attr: Attributes)
  return Ceiling_Priority;
```

11.2.5.2 Description

The functionality described in this subclause is optional. If neither the Mutex Priority Ceiling option nor the Mutex Priority Inheritance option is supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

Objects of type `Locking_Policy` are used to specify the value of the Locking Policy attribute or a mutex.

The implementation shall support the value `No_Priority_Inheritance`.

If the Mutex Priority Inheritance option is supported: The implementation shall support the locking policy value `Highest_Blocked_Task`.

If the Mutex Priority Ceiling option is supported: The implementation shall support the locking policy value `Highest_Ceiling_Priority` and the integer mutex attribute Ceiling Priority, and the range of `Ceiling_Priority` shall include `System.Priority`.

If the Priority Task Scheduling option is supported: The effects of mutex Locking Policy on scheduling of tasks shall be as specified in 13.4.

If the Priority Task Scheduling option is not supported, but the task dispatching model of D.2.1 of the Ada RM {1} is supported, the following apply:

— When a task owns a mutex initialized with the `No_Priority_Inheritance` locking policy, its active priority and scheduling are not affected by its ownership of that mutex.

— If the implementation allows an application to specify `Ceiling_Locking` as the protected object locking policy, it shall also allow the application to specify `Highest_Ceiling_Priority` as the locking policy for mutexes.

— When a task owns one or more mutexes initialized with the `Highest_Ceiling_Priority` locking policy, it shall inherit the Ceiling Priority of every mutex owned by this task and initialized with this attribute, regardless of whether other tasks are blocked on any of these mutexes.

— When a task is blocking tasks with higher active priority because it owns one or more mutexes initialized with the `Highest_Blocked_Task` locking policy, it shall inherit the active priorities of all the tasks waiting on the mutexes owned by this task and initialized with this locking policy. If a task that owns mutexes becomes blocked on another mutex, the same priority inheritance effect described above shall be propagated to the owner of that mutex, in a recursive manner.

`Set_Locking_Policy` shall set the Locking Policy attribute of the previously initialized mutex attributes object specified by `Attr` to the value of `Locking`, provided the specified locking policy is supported.

`Get_Locking_Policy` shall return the Locking Policy attribute value of the previously initialized mutex attributes object `Attr`.

If the Mutex Priority Ceiling option is supported:

— `Set_Ceiling_Priority` shall set the Ceiling Priority attribute of the mutex attributes object specified by `Attr` to the value of `New_Ceiling`.

— `Get_Ceiling_Priority` shall return the Ceiling Priority attribute value of `Attr`.

— If `Set_Ceiling_Priority` and `Get_Ceiling_Priority` are called for a mutex attributes object whose Locking Policy attribute is not equal to `Highest_Ceiling_Priority`, `POSIX_Error` shall be raised.

NOTE: In order to avoid priority inversion, the priority ceiling of a mutex should be set to a priority equal to or greater than the anticipated maximum active priority of all tasks that may lock the mutex.

NOTE: All the rules above governing the locking and unlocking of mutexes also apply to the implicit locking and unlocking of mutexes through the `Wait` and `Timed_Wait` operations on condition variables (see 3.3).

11.2.5.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Operation_Not_Implemented`

Neither the Mutex Priority Inheritance option nor the Mutex Priority Ceiling option is supported, and the implementation does not support the `Get_Locking_Policy` and `Set_Locking_Policy` operations.

There is no support for the Mutex Priority Ceiling option, and the implementation does not support the `Set_Ceiling_Priority` and `Get_Ceiling_Priority` operations.

`Operation_Not_Supported`

The value specified to `Set_Locking_Policy` by `Locking` is an unsupported value.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

Invalid_Argument

Attr does not specify a properly initialized mutex attributes object.

Either Set_Ceiling_Priority or Get_Ceiling_Priority were called on a mutex attributes object whose Locking Policy attribute is not equal to Highest_Ceiling_Priority locking policy.

Operation_Not_Permitted

The caller does not have the appropriate privilege to perform the operation.

11.2.6 Initialize and Finalize a Mutex

11.2.6.1 Synopsis

```
procedure Initialize
   (M: in out Mutex;
    Attr: in Attributes);
procedure Initialize (M: in out Mutex);
function Descriptor_Of (M: Mutex) return Mutex_Descriptor;
procedure Finalize (M: in out Mutex);
```

11.2.6.2 Description

Initialize shall initialize the mutex M with attributes specified by Attr. If the version of Initialize without the Attr parameter is used, the mutex M shall be initialized to the default mutex attributes values.

Upon return from Initialize, the mutex is in an unlocked state. If Initialize is called again on the same mutex (without an intervening Finalize), the effect is undefined.

Descriptor_Of shall return a descriptor for the mutex object M that can later be used in mutex operations. If Descriptor_Of is called again, without an intervening call to Finalize, the same descriptor value shall be returned. If the mutex object M is not already initialized, the effect is undefined.

Finalize shall finalize the mutex M; the mutex becomes, in effect, uninitialized. An implementation may cause Finalize to set M to an invalid value. A finalized mutex can be reinitialized using Initialize; the results of otherwise referencing the mutex after it has been finalized are undefined.

Attempting to finalize a locked mutex results in undefined behavior.

NOTE: In order to provide the correct semantics, implementations must ensure that the parameter M of Initialize and Initialize is passed by reference.

11.2.6.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Temporarily_Unavailable

The system lacked the necessary resources other than memory to initialize another mutex.

Not_Enough_Space

Insufficient memory exists to initialize the mutex.

Operation_Not_Permitted

The caller does not have the privilege to initialize the mutex M.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Busy

The implementation has detected an attempt to initialize an already initialized mutex.

The implementation has detected an attempt to finalize a mutex that is locked or otherwise referenced (for example, it was specified on a pending call to Wait or Timed_Wait on a condition variable by another task, and that task has not been unblocked yet).

Invalid_Argument

Attr does not specify a properly initialized mutex attributes object.

M is not a properly initialized mutex.

The Resource_Busy and Invalid_Argument error checks, if implemented, shall act as if they were performed immediately at the beginning of the operation. If an exception is raised, it shall be raised prior to modifying the state of the mutex M.

11.2.7 Change the Ceiling Priority of a Mutex

11.2.7.1 Synopsis

```
procedure Set_Ceiling_Priority
   (M:           in Mutex_Descriptor;
    New_Ceiling: in Ceiling_Priority;
    Old_Ceiling: out Ceiling_Priority);
function Get_Ceiling_Priority (M: Mutex_Descriptor)
  return Ceiling_Priority;
```

11.2.7.2 Description

The functionality described in this subclause is optional. If the Mutex Priority Ceiling option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

Get_Ceiling_Priority shall return the current Ceiling Priority attribute of the mutex.

Set_Ceiling_Priority shall either lock the mutex specified by M if it is unlocked, or cause the caller to become blocked until the mutex can be locked. Set_Ceiling_Priority shall then change the Ceiling Priority attribute of the mutex and unlock the mutex. Upon a normal return, the previous value of the Ceiling Priority attribute shall be returned in Old_Ceiling. The process of locking the mutex is not required to adhere to the Highest_Ceiling_Priority locking policy.

Whether Set_Ceiling_Priority behaves as specified above or merely returns error code Operation_Not_Implemented is implementation defined.

If Set_Ceiling_Priority and Get_Ceiling_Priority are called for a mutex that was not initialized with the Highest_Ceiling_Priority locking policy, POSIX_Error may be raised.

If Set_Ceiling_Priority fails, the Ceiling Priority attribute of the mutex shall not be changed.

11.2.7.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Implemented
: The implementation does not support the Set_Ceiling_Priority and Get_Ceiling_Priority operations.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: The priority requested by New_Ceiling is an unsupported value.

  M does not specify a properly initialized mutex.

Operation_Not_Implemented
: The implementation does not support the Highest_Ceiling_Priority locking policy for mutexes.

  The implementation does not support the operation Set_Ceiling_Priority.

Operation_Not_Permitted
: The caller does not have the appropriate privilege to perform the operation.

11.2.8 Lock and Unlock a Mutex

11.2.8.1 Synopsis

```
procedure Lock (M: in Mutex_Descriptor);
function  Try_Lock (M: Mutex_Descriptor) return Boolean;
procedure Unlock (M: in Mutex_Descriptor);
```

11.2.8.2 Description

Lock shall lock the mutex object specified by M. If the mutex is already locked, the calling task is blocked until the mutex becomes available. This operation returns with the mutex object specified by M in the locked state with the calling task as its owner. An attempt by the owner of a mutex to relock the mutex results in undefined behavior.

Try_Lock is identical to Lock except that it always returns immediately. If the mutex object specified by M is currently locked by any task, including the current task, False is returned. Otherwise, True is returned.

`Unlock` is called by the owner of the mutex object specified by `M` to release it.

If `Unlock` is called by a task that is not the owner of the mutex object specified by `M`, or if the mutex already is in an unlocked state, the effect is undefined.

If tasks are blocked on the mutex object specified by `M` when `Unlock` is called, then one of the blocked tasks is unblocked and allowed to return from its call to `Lock`, and it becomes the owner of the mutex specified in its call to `Lock`.

If the Priority Task Scheduling option is supported: If more than one task is blocked waiting for the mutex, the scheduling policies and parameters in effect for the blocked tasks shall determine which task is allowed to acquire the mutex. When tasks executing with the policies `Sched_FIFO` or `Sched_RR` are waiting on a mutex, they shall acquire the mutex in priority order when the mutex is unlocked. For other scheduling policies, the selection of which task to unblock is unspecified.

If the Priority Task Scheduling option is not supported, the order in which to unblock the tasks is unspecified.

11.2.8.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: `Lock` or `Try_Lock` was called for a mutex initialized with `Highest_Ceiling_Priority`, and the calling task has a priority higher than the ceiling of the mutex. The implementation may instead raise `Program_Error`.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: `M` does not specify a properly initialized mutex.

`Resource_Deadlock_Avoided`
: The task calling `Lock` already owns the mutex.

`Operation_Not_Permitted`
: For `Unlock`, the current task does not own the mutex.

11.3 Package `POSIX_Condition_Variables`

This package provides access to services related to condition variables. A condition variable is a synchronization object that can be used, in combination with a mutex, to wait for a logical condition.

NOTE: Mutexes and condition variables are provided in this standard primarily for synchronization between Ada tasks and C-language threads in mixed-language applications. The preferred mechanisms for intertask synchronization are those provided by the Ada langauge, namely, protected objects and task entries.

The functionality described in this clause is optional. If the Mutexes option is not supported, the implementation may cause all calls to the explicitly declared operations

defined in this clause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this clause.*

```
with POSIX,
     POSIX_Mutexes;
package POSIX_Condition_Variables is
   -- 11.3.1 Condition and Condition Descriptor Types
   type Condition is limited private;
   type Condition_Descriptor is private;
   -- 11.3.2 Condition Variable Attributes Type
   type Attributes is private;
   procedure Initialize (Attr: in out Attributes);
   procedure Finalize (Attr: in out Attributes);
   -- 11.3.3 Condition Process Shared Attribute
   function Get_Process_Shared (Attr: Attributes)
     return Boolean;
   procedure Set_Process_Shared
      (Attr:      in out Attributes;
       Is_Shared: in Boolean:= False);
   -- 11.3.4 Initialize and Finalize a Condition
   procedure Initialize
      (Cond: in out Condition;
       Attr: in Attributes);
   procedure Initialize (Cond: in out Condition);
   function Descriptor_Of (Cond: Condition)
     return Condition_Descriptor;
   procedure Finalize (Cond: in out Condition);
   -- 11.3.5 Broadcast and Signal a Condition
   procedure Signal (Cond: in Condition_Descriptor);
   procedure Broadcast (Cond: in Condition_Descriptor);
   -- 11.3.6 Wait on a Condition
   procedure Wait
      (Cond: in Condition_Descriptor;
       M:    in POSIX_Mutexes.Mutex_Descriptor);
   procedure Timed_Wait
      (Cond:    Condition_Descriptor;
       M:       POSIX_Mutexes.Mutex_Descriptor;
       Timeout: POSIX.Timespec);

private
 implementation-defined
end POSIX_Condition_Variables;
```

11.3.1 Condition and Condition Descriptor Types

11.3.1.1 Synopsis

```
   type Condition is limited private;
   type Condition_Descriptor is private;
```

11.3.1.2 Description

The type `Condition` is used to represent condition variable objects.

Operating on copies of `Condition` objects (which may be created by returning function results of this type or by using the type as an **`in out`** parameter) results in undefined behavior.

Objects of type Condition_Descriptor are used as condition variable descriptors; that is, handles for condition variable objects. A condition variable descriptor is *valid* if it is associated with a properly initialized condition variable; that is, the condition variable descriptor is returned by a call to Descriptor_Of with a parameter that has been used in a prior call to Initialize and not yet used in a call to Finalize.

Copies of Condition_Descriptor objects shall always specify the same condition variable. Operating on an object of type Condition_Descriptor after the corresponding condition variable object is finalized results in undefined behavior, which might affect other processes sharing the same condition variable.

11.3.2 Condition Variable Attributes Type

11.3.2.1 Synopsis

```
type Attributes is private;
procedure Initialize (Attr: in out Attributes);
procedure Finalize (Attr: in out Attributes);
```

11.3.2.2 Description

Objects of type Attributes specify a set of condition variable creation attributes.

Initialize shall initialize the condition variable attributes object Attr with the default values for all of the attributes defined by the implementation.

Attempting to initialize an already initialized condition variable attributes object results in undefined behavior.

After a condition variable attributes object has been used to initialize one or more condition variables, any operation affecting the attributes object (including Finalize) does not affect any previously initialized condition variables.

Finalize shall finalize the condition variable attributes object Attr; the object becomes, in effect, uninitialized. An implementation may cause Finalize to set the object Attr to an invalid value. A finalized condition variable attributes object can be reinitialized using Initialize; the results of otherwise referencing the object after it has been finalized are undefined.

This standard specifies certain attributes of condition variables (see 11.3.3). Additional attributes, their default values, and the names of the associated operations to get and set those attribute values are implementation defined.

11.3.2.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Not_Enough_Space

Insufficient memory exists to initialize the condition variable attributes object.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Upon Finalize, Attr does not specify a properly initialized condition attributes object.

11.3.3 Condition Process Shared Attribute

11.3.3.1 Synopsis

```
function Get_Process_Shared (Attr: Attributes)
  return Boolean;
procedure Set_Process_Shared
   (Attr:      in out Attributes;
    Is_Shared: in Boolean:= False);
```

11.3.3.2 Description

The functionality described in this subclause is optional. If the Process Shared option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

The implementation shall support a condition variable attribute called Process Shared and the associated operations Get_Process_Shared and Set_Process_Shared. This attribute indicates whether use of a newly initialized condition variable object can be shared by tasks in different processes having access to that condition variable.

Set_Process_Shared sets the Process Shared attribute of an initialized condition variable attributes object Attr to Is_Shared.

Get_Process_Shared returns the value of the Process Shared attribute of the condition variable attributes object Attr.

If the Process Shared attribute of a given condition variable attributes object is True, a condition variable initialized with this object shall be *sharable*; that is, the condition variable can be operated upon by any task that has access to the memory where the condition variable is allocated (even tasks from other processes). Otherwise, the condition variable object shall not be sharable, that is, it can be operated upon only by tasks created within the same process as the task that initialized the condition variable object. If tasks of other processes attempt to operate on such a condition variable, the behavior is undefined.

The default value of the Process Shared attribute shall be False.

11.3.3.3 Error Handling

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Attr does not specify a properly initialized condition attributes object.

11.3.4 Initialize and Finalize a Condition

11.3.4.1 Synopsis

```
procedure Initialize
   (Cond: in out Condition;
    Attr: in Attributes);
procedure Initialize (Cond: in out Condition);
function Descriptor_Of (Cond: Condition)
  return Condition_Descriptor;
procedure Finalize (Cond: in out Condition);
```

11.3.4.2 Description

`Initialize` shall initialize the condition variable `Cond` with attributes specified by `Attr`. If the version of `Initialize` without the `Attr` parameter is used, `Cond` shall be initialized to the default values for all condition variable attributes. If `Initialize` is called again on the same condition variable (without an intervening `Finalize`), the effect is undefined.

`Descriptor_Of` shall return a descriptor for the previously initialized condition variable object `Cond` that can later be used in condition variable operations. If `Descriptor_Of` is called again, without an intervening call to `Finalize`, the same descriptor value shall be returned. If the condition variable object `Cond` is not already initialized, the effect is undefined.

NOTE: In order to provide the correct semantics, implementations must ensure that the parameter `Cond` of `Initialize` and `Finalize` is passed by reference.

`Finalize` shall finalize the condition variable `Cond`; the condition variable becomes, in effect, uninitialized. An implementation may cause `Finalize` to set `Cond` to an invalid value. A finalized condition variable can be reinitialized using `Initialize`; the results of otherwise referencing the condition variable after it has been finalized are undefined.

Attempting to finalize a condition variable upon which tasks are currently blocked results in undefined behavior.

11.3.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Resource_Temporarily_Unavailable`

The system lacked the necessary resources (other than memory) to initialize another condition variable.

`Not_Enough_Space`

Insufficient memory exists to initialize the condition variable.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

Resource_Busy

The implementation has detected an attempt to initialize an already initialized condition variable object.

The implementation has detected an attempt to finalize the object Cond while it is referenced (for example, while being used in a Wait or Timed_Wait by another task).

Invalid_Argument

Attr does not specify a properly initialized condition attributes object.

Cond is not a properly initialized condition variable.

The Resource_Busy and Invalid_Argument error checks, if implemented, shall act as if they were performed immediately at the beginning of the operation. If an exception is raised, it shall be raised prior to modifying the state of the condition variable Cond.

11.3.5 Broadcast and Signal a Condition

11.3.5.1 Synopsis

```
procedure Signal (Cond: in Condition_Descriptor);
procedure Broadcast (Cond: in Condition_Descriptor);
```

11.3.5.2 Description

The procedures Signal and Broacast are used to unblock tasks that are blocked on a condition variable.

If any tasks are blocked on the condition variable specified by Cond, Signal shall unblock at least one of these tasks.

Broadcast shall unblock all tasks currently blocked on the condition variable specified by Cond.

If the Priority Task Scheduling option is supported: If more than one task is blocked on the condition variable specified by Cond, the scheduling policies and parameters in effect for the blocked tasks shall determine the order in which the tasks shall be unblocked. Tasks executing with the scheduling policies Sched_FIFO or Sched_RR shall be unblocked in priority order. For other scheduling policies, the order in which to unblock the tasks is unspecified.

If the Priority Task Scheduling option is not supported, the order in which to unblock the tasks is unspecified.

After a Signal or Broadcast operation has unblocked a given task that has called Wait or Timed_Wait, the task shall contend for the mutex that was specified in the Wait or Timed_Wait operation as if it had called Lock. (See 11.2.8). Upon return from Wait or Timed_Wait, the task shall be the owner of the mutex.

Signal or Broadcast can be called by a task whether or not it is the owner of the mutex that tasks calling Wait or Timed_Wait have associated with the condition variable in their calls.

Signal and Broadcast shall have no effect if no tasks are currently blocked on the condition variable specified by Cond.

11.3.5.3 Error Handling

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

`Cond` is not a valid condition variable descriptor.

11.3.6 Wait on a Condition

11.3.6.1 Synopsis

```
procedure Wait
   (Cond: in Condition_Descriptor;
    M:    in POSIX_Mutexes.Mutex_Descriptor);
procedure Timed_Wait
   (Cond:    Condition_Descriptor;
    M:       POSIX_Mutexes.Mutex_Descriptor;
    Timeout: POSIX.Timespec);
```

11.3.6.2 Description

`Wait` and `Timed_Wait` cause the calling task to be blocked on a condition variable until it is signaled or broadcast. The calling task is required to be the owner of the mutex specified by `M`; otherwise, the effect is undefined.

These operations atomically unlock the mutex specified by `M` and cause the calling task to become blocked on the condition variable specified by `Cond`. In other words, if another task is able to lock the mutex after the about-to-block task has unlocked it, then a subsequent call to `Signal` or `Broadcast` in that task shall behave as if it were issued after the about-to-block task has blocked.

Upon a normal return, the calling task shall, again, be the owner of the mutex specified by `M`.

*NOTE: Since spurious wakeups from `Wait` or `Timed_Wait` may occur, an appliction that uses condition variables for synchronization needs to have an independent way to check whether the logical condition or event for which a task is waiting has actually occurred. Checking is normally accomplished by enclosing each call to `Wait` or `Timed_Wait` in a **`while`** loop that tests the logical condition and executes the wait operation repeatedly until the logical condition is true. The test for the logical condition is normally coded as a `Boolean` predicate involving shared variables that are protected by the mutex.*

The effect of using more than one mutex for concurrent `Wait` or `Timed_Wait` operations on the same condition variable is undefined.

NOTE: A condition variable becomes associated with a unique mutex when a task waits on the condition variable, and this (dynamic) association ends when the wait returns.

A condition wait (whether timed or not) is an abort completion point as defined in 9.8 (15) of the Ada RM {1}.

`Timed_Wait` shall behave the same as `Wait` except that an error occurs if and when the absolute time specified by `Timeout` is reached (*i.e.*, when the system time as measured by `Clock_Realtime` equals or exceeds `Timeout`) before the condition specified

by Cond is signaled or broadcast. If the absolute time specified by Timeout has already been reached at the time of the call, Timed_Wait shall return immediately with an error. When this occurs, Timed_Wait shall nonetheless unlock and relock the mutex specified by M.

The operation Wait shall not be interruptible by a signal.

11.3.6.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Timed_Out

The time specified by Timeout has passed.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Cond is not a valid condition variable descriptor.

M is not a valid mutex descriptor.

Different mutexes were specified for concurrent Wait or Timed_Wait operations on the same condition variable.

The mutex was not owned by the current task at the time of the call.

The Timeout value cannot be interpreted as a valid Timespec value. The implementation may instead raise Constraint_Error.

Except for Timed_Out, all these error checks shall act as if they were performed immediately at the beginning of the operation and shall raise an exception prior to modifying the state of the mutex specified by M or the condition variable specified by Cond.

Section 12: Memory Management

This section specifies interfaces for memory management, including locking portions of the address space of a process so that they are continually resident in memory, and mapping files and shared memory objects to the address space of a process.

Memory range locking and memory mapping operations are defined in terms of pages. Implementations may restrict the size and alignment of range lockings and mappings to be on page size boundaries. The page size, as a count of storage units, is the value of the runtime invariant limit Page Size. If an implementation has no restrictions on size or alignment, it may specify a one-unit page size.

Memory locking guarantees the residence of portions of the address space. It is implementation defined whether locking memory guarantees fixed translation between virtual addresses (as seen by the process) and physical addresses. Per-process memory locks are not inherited across a `POSIX_Unsafe_Process_Primitives.Fork`, `POSIX_Process_Primitives.Start_Process`, and `POSIX_Process_Primitives.Start_Process_Search`. All memory locks owned by a process are unlocked upon `POSIX_Unsafe_Process_Primitives.Exec`, `POSIX_Unsafe_Process_Primitives.Exec_Search`, or process termination. Unmapping of an address range removes any memory locks established on that address range by the process.

If the Memory Mapped Files option is supported: A process may access files by directly incorporating file data into the address space of a process. Once a file is mapped into the address space of a process, the data can be manipulated as memory. If more than one process map a file, its contents are shared among them. If the mappings allow shared write access, then data written into the memory object through the address space of one process shall appear in the address spaces of all processes that similarly map the same portion of the memory object (see 12.4.)

If the Shared Memory Objects option is supported: Regions of storage may be created, independent of the file system, and mapped into the address space of one or more processes to allow them to share the associated memory.

Implementations may support the Shared Memory Objects option without supporting the Memory Mapped Files option.

The operation `POSIX_Files.Unlink` of a file or the operation `POSIX_Shared_Memory_Objects.Unlink_Shared_Memory` of a shared memory object, while causing the removal of the name, does not unmap any mappings established for the object. Once the name has been removed, the contents of the memory object are preserved as long as it is referenced. The memory object remains referenced as long as a process has the memory object open or has some area of the memory object mapped.

The following additional specifications apply if the Memory Protection option is supported, the following apply:

— The mapping may be restricted to disallow some types of access.

— References to whole pages within the mapping but beyond the current length in storage units of an object shall result in `Signal_Bus_Error`.

— Write attempts to memory that was mapped without write access, or any access to memory mapped with `Protection_Options` equal to `POSIX.Empty_Set`, shall result in `Signal_Segmentation_Violation`.

— References to unmapped addresses shall result in `Signal_Segmentation_Violation`.

NOTE: As specified in 3.3.3, the Ada language implementation is required to translate occurrences of `Signal_Bus_Error` and `Signal_Segmentation_Violation` to `Program_Error`, unless they are identifiable as corresponding to a check that requires some other exception to be raised (such as `Storage_Error` or `Constraint_Error`).

If the Memory Protection option is not supported, the effect of references to unmapped addresses is undefined.

In any case, the size of a memory object is unaffected by access beyond the end of the object.

12.1 Package POSIX_Memory_Locking

This package provides access to services that a process can use to control whether all of its address space is continually resident in memory.

The functionality described in this clause is optional. If the Memory Locking option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this clause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this clause.

```
with POSIX;
package POSIX_Memory_Locking is
   -- 12.1.1 Lock/Unlock the Address Space of a Process
   type Memory_Locking_Options is new POSIX.Option_Set;
   Current_Pages: constant Memory_Locking_Options:= implementation-defined;
   Future_Pages:  constant Memory_Locking_Options:= implementation-defined;
   procedure Lock_All (Options: in Memory_Locking_Options);
   procedure Unlock_All;

end POSIX_Memory_Locking;
```

12.1.1 Lock/Unlock the Address Space of a Process

12.1.1.1 Synopsis

```
type Memory_Locking_Options is new POSIX.Option_Set;
Current_Pages: constant Memory_Locking_Options:= implementation-defined;
Future_Pages:  constant Memory_Locking_Options:= implementation-defined;
procedure Lock_All (Options: in Memory_Locking_Options);
procedure Unlock_All;
```

12.1.1.2 Description

`Lock_All` shall cause all of the pages mapped by the address space of a process to be memory-resident until unlocked or until the process exits or issues `POSIX_Unsafe_Process_Primitives.Exec` or `POSIX_Unsafe_Process_Primitives.Exec_Search` for another process image. `Options` determines whether the pages to

be locked are those currently mapped by the address space of a process, those that will be mapped in the future, or both. `Options` is constructed from the union ("+") of one or more of the following constants:

`Current_Pages`
: Lock all of the pages currently mapped into the address space of a process.

`Future_Pages`
: Lock all of the pages that become mapped into the address space of a process in the future when those mappings are established.

If `Future_Pages` is specified and the automatic locking of future mappings eventually causes the amount of locked memory to exceed the amount of available physical memory or any other implementation-defined limit, the behavior is implementation defined. The manner in which the implementation informs the application of these situations is implementation defined.

`Unlock_All` unlocks all currently mapped pages of the address space of a process. After a call to `Unlock_All`, any pages that become mapped into the address space of the process shall not be locked, unless there was an intervening call to `Lock_All` specifying `Future_Pages` or a subsequent call to `Lock_All` specifying `Current_Pages`. If pages mapped into the address space of the process are also mapped into other processes' address spaces and are locked by those processes, the locks established by the other processes are unaffected by the call to `Unlock_All`.

Upon successful return from the `Lock_All` operation that specifies `Current_Pages`, all currently mapped pages of the address space of the process shall be memory-resident and locked. Upon return from the `Unlock_All` operation, all currently mapped pages of the address space of the process shall be unlocked with respect to the process. The memory-residency of unlocked pages is unspecified.

The appropriate privilege is required to lock process memory with `Lock_All`.

12.1.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Operation_Not_Implemented`
: The implementation does not support this memory-locking interface.

`Invalid_Argument`
: `Options` is the empty set.

`Resource_Temporarily_Unavailable`
: Upon `Lock_All`, some or all of the memory identified by the operation could not be locked when the call was made.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Not_Enough_Space`
: Upon `Lock_All`, locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process can lock.

Operation_Not_Permitted

Upon Lock_All, the calling process does not have the appropriate privilege to perform the requested operation.

12.2 Package POSIX_Memory_Range_Locking

This package provides access to services that a process can use to control whether specific ranges of its address space are continually resident in memory.

The functionality described in this clause is optional. If the Memory Range Locking option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this clause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this clause.

```
with System_Storage_Elements,
     System;
package POSIX_Memory_Range_Locking is
   -- 12.2.1 Lock/Unlock a Range of Process Address Space
   procedure Lock_Range
      (First:  in System.Address;
       Length: in System_Storage_Elements.Storage_Offset);
   procedure Unlock_Range
      (First:  in System.Address;
       Length: in System_Storage_Elements.Storage_Offset);

end POSIX_Memory_Range_Locking;
```

12.2.1 Lock/Unlock a Range of Process Address Space

12.2.1.1 Synopsis

```
   procedure Lock_Range
      (First:  in System.Address;
       Length: in System_Storage_Elements.Storage_Offset);
   procedure Unlock_Range
      (First:  in System.Address;
       Length: in System_Storage_Elements.Storage_Offset);
```

12.2.1.2 Description

Lock_Range shall cause the whole pages containing any part of the address space of the process starting at address First and continuing for Length storage units to be memory-resident until unlocked or until the process exits or issues POSIX_Unsafe_Process_Primitives.Exec or POSIX_Unsafe_Process_Primitives.Exec_Search for another process image. The implementation may require that First be a multiple of Page Size.

NOTE: It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of First.

Unlock_Range shall unlock the whole pages containing any part of the address space of the process starting at address First and continuing for Length storage units, regardless of how many times Lock_Range has been called by the process for any of

the pages in the specified range. The implementation may require that `First` be a multiple of Page Size.

NOTE: It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of `First`

If some of the pages in the range specified by a call to `Unlock_Range` are also mapped into other processes' address spaces, any locks established on those pages by another process are unaffected by the call to `Unlock_Range`. If some of the pages in the range specified by a call to `Unlock_Range` are also mapped into other portions of the address space of the calling process outside the range specified, any locks established on those pages via the other mappings are also unaffected by this call.

Upon successful return from `Lock_Range`, pages in the specified range shall be locked and memory-resident. Upon successful return from `Unlock_Range`, pages in the specified range shall be unlocked with respect to the address space of the calling process. Memory-residency of unlocked pages is unspecified.

The appropriate privilege is required to lock process memory with `Lock_Range`.

12.2.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Not_Enough_Space`

Some or all of the address range specified by the `First` and `Length` arguments does not correspond to valid mapped pages in the address space of the calling process.

`Operation_Not_Implemented`

The implementation does not support this memory-locking interface.

`Resource_Temporarily_Unavailable`

Upon `Lock_Range`, some or all of the memory identified by the operation could not be locked when the call was made.

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

The `First` argument is not a multiple of Page Size, and the implementation imposes a restriction that `First` be page-aligned.

`Not_Enough_Space`

Upon `Lock_Range`, locking the pages mapped by the specified range would exceed an implementation-defined limit on the amount of memory that the process can lock.

`Operation_Not_Permitted`

Upon `Lock_Range`, the calling process does not have the appropriate privilege to perform the requested operation.

12.3 Package POSIX_Memory_Mapping

This package provides access to services that a process can use to map specified portions of certain files to specified ranges of the address space of the calling process.

```
with POSIX,
     POSIX_IO,
     POSIX_Signals,
     System_Storage_Elements,
     SYSTEM;
package POSIX_Memory_Mapping is
   -- 12.3.1 Map Process Addresses to a Memory Object
   type Protection_Options is new POSIX.Option_Set;
   Allow_Read:    constant Protection_Options:= implementation-defined;
   Allow_Write:   constant Protection_Options:= implementation-defined;
   Allow_Execute: constant Protection_Options:= implementation-defined;
   type Mapping_Options is range implementation-defined;
   Map_Shared:    constant Mapping_Options:= implementation-defined;
   Map_Private:   constant Mapping_Options:= implementation-defined;
   type Location_Options is range implementation-defined;
   Exact_Address:  constant Location_Options:= implementation-defined;
   Nearby_Address: constant Location_Options:= implementation-defined;
   function Map_Memory
      (First:      System.Address;
       Length:     System_Storage_Elements.Storage_Offset;
       Protection: Protection_Options;
       Mapping:    Mapping_Options;
       Location:   Location_Options;
       File:       POSIX_IO.File_Descriptor;
       Offset:     POSIX_IO.IO_Offset)
     return System.Address;
   function Map_Memory
      (Length:     System_Storage_Elements.Storage_Offset;
       Protection: Protection_Options;
       Mapping:    Mapping_Options;
       File:       POSIX_IO.File_Descriptor;
       Offset:     POSIX_IO.IO_Offset)
     return System.Address;
   -- 12.3.2 Unmap Memory
   procedure Unmap_Memory
      (First:  in System.Address;
       Length: in System_Storage_Elements.Storage_Offset);
   -- 12.3.3 Change Memory Protection
   procedure Change_Protection
      (First:      in System.Address;
       Length:     in System_Storage_Elements.Storage_Offset;
       Protection: in Protection_Options);
   -- 12.3.4 Memory Object Synchronization
   type Synchronize_Memory_Options is new POSIX.Option_Set;
   Wait_For_Completion:    constant Synchronize_Memory_Options
      := implementation-defined;
   Invalidate_Cached_Data: constant Synchronize_Memory_Options
      := implementation-defined;
```

```
   procedure Synchronize_Memory
      (First:   in System.Address;
       Length:  in System_Storage_Elements.Storage_Offset;
       Options: in Synchronize_Memory_Options:= Empty_Set);

end POSIX_Memory_Mapping;
```

12.3.1 Map Process Addresses to a Memory Object

12.3.1.1 Synopsis

```
   type Protection_Options is new POSIX.Option_Set;
   Allow_Read:    constant Protection_Options:= implementation-defined;
   Allow_Write:   constant Protection_Options:= implementation-defined;
   Allow_Execute: constant Protection_Options:= implementation-defined;
   type Mapping_Options is range implementation-defined;
   Map_Shared:    constant Mapping_Options:= implementation-defined;
   Map_Private:   constant Mapping_Options:= implementation-defined;
   type Location_Options is range implementation-defined;
   Exact_Address:  constant Location_Options:= implementation-defined;
   Nearby_Address: constant Location_Options:= implementation-defined;
   function Map_Memory
      (First:      System.Address;
       Length:     System_Storage_Elements.Storage_Offset;
       Protection: Protection_Options;
       Mapping:    Mapping_Options;
       Location:   Location_Options;
       File:       POSIX_IO.File_Descriptor;
       Offset:     POSIX_IO.IO_Offset)
     return System.Address;
   function Map_Memory
      (Length:     System_Storage_Elements.Storage_Offset;
       Protection: Protection_Options;
       Mapping:    Mapping_Options;
       File:       POSIX_IO.File_Descriptor;
       Offset:     POSIX_IO.IO_Offset)
     return System.Address;
```

12.3.1.2 Description

The functionality described in this subclause is optional. If neither the Memory Mapped Files option nor the Shared Memory Objects option is supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`. *Otherwise, the behavior shall be as specified in this subclause.*

Two overloaded subprograms are defined for `Map_Memory`. The effects differ only in how the starting point address for the mapping, within the address space of the calling process, is determined.

The form of `Map_Memory` without the `First` and `Location` parameters grants the system complete freedom in selecting the starting point address for the memory mapping, except that it shall never place a mapping at address zero, nor shall it replace an extant mapping.

The other form of `Map_Memory`, with the `First` and `Location` parameters, requests that the starting point be at or near the specific address `First`. For this form, the

`Location` argument shall specify `Exact_Address` or `Nearby_Address`, but not both.

For both versions, the effect of the operation is to establish a mapping between the specified region of the address space of the calling process, from the specified starting point and continuing for `Length` storage units to the specified region of the memory object represented by the file descriptor `File` at offset `Offset` storage units and continuing for `Length` storage units. The starting-point address is returned by the call. The range of addresses from the starting point and continuing for `Length` storage units is required to be legitimate for the possible (not necessarily current) address space of the process. The range of storage units starting at `Offset` storage units and continuing for `Length` storage units shall be legitimate for the possible (not necessarily current) offsets in the file or shared memory object specified by `File`.

The mapping established by `Map_Memory` shall replace any previous mappings for the whole pages containing any part of the specified region of the address space of the calling process.

The system performs mapping operations over whole pages. Thus, while the parameter `Length` need not meet a size or alignment constraint, the system shall include, in any mapping operation, any partial page included in the specified region of the address space.

If `Location` specifies `Exact_Address`, the starting point address of the mapping shall be `First` exactly. `First` is required to be nonzero and to the same remainder, modulo Page Size, as the `Offset` parameter. The implementation may require that `Offset` be a multiple of Page Size. The implementation may require that `First` be a multiple of Page Size.

NOTE: It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of the `Offset` and `First` arguments.

It is implementation defined whether `Exact_Address` is supported.

NOTE: For implementations that support `Exact_Address`, its use may result in poor performance.

If `Location` specifies `Nearby_Address`, the system uses `First` in an implementation-defined manner to arrive at the starting point address. The value so chosen shall be an area of the address space that the system deems suitable for a mapping of `Length` storage units to the specified object. `First` is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a starting point address, it shall never place a mapping at address zero, nor shall it replace an extant mapping.

The parameter `Protection` determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The `Protection` is required to be either `Empty_Set`, meaning the data cannot be accessed, or the union ("+") of one or more of the other `Protection_Options` with the following associated meanings:

`Allow_Read`

Data can be read.

`Allow_Write`

Data can be written.

`Allow_Execute`

Data can be executed.

If an implementation cannot support the combination of access types specified by `Protection`, the call to `Map_Memory` shall fail. An implementation may permit accesses of other than those specified by `Protection`, except as follows:

If the Memory Protection option is supported, the implementation shall not permit a write to succeed where `Allow_Write` has not been specified, or permit any access where `Empty_Set` has been specified.

If the Memory Protection option is supported: The implementation shall support at least the following values of `Protection`: `Empty_Set`, `Allow_Read`, `Allow_Write` and the union ("+") of `Allow_Read` and `Allow_Write`. Otherwise, if the Memory Protection option is not supported, the result of any access that conflicts with the specified protection is undefined.

The file descriptor `File` is required to have been opened previously by the application with read permission, regardless of the protection options specified. If `Allow_Write` is specified, the application is required to have opened the file descriptor `File` with write permission unless `Map_Private` is specified in `Mapping` as described below.

`Mapping` values of type `Mapping_Options` describe the disposition of write references to the memory object.

If `Map_Shared` is specified, write references change the underlying object. If the implementation supports `Map_Private` and it is specified, modifications to the mapped data by the calling process shall be visible only to the calling process and shall not change the underlying object. It is unspecified whether modifications to the underlying object done after the `Map_Private` mapping is established are visible through the `Map_Private` mapping. `Mapping` is required to specify either `Map_Shared` or `Map_Private`. The type of memory mapping in effect (private or shared) for each region of the address space is retained across `POSIX_Unsafe_Process_Primitives.Fork`.

Implementations in which the Memory Mapped Files option is not supported are not required to support `Map_Private`.

The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object that are beyond the end of the object.

If the Memory Protection option is supported: References to addresses that are both within the mapped region and within whole pages that are past the end of the mapped memory object shall result in raising `Program_Error`.

If the Memory Protection option is not supported, the result of such references is undefined.

An implementation may raise the exception `Program_Error` when a reference would cause an error in the mapped object, such as out-of-space condition.

12.3.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

The file descriptor `File` is not open for read, regardless of the protection specified.

The file descriptor `File` is not open for write and `Allow_Write` was specified for a `Map_Shared` mapping.

`Resource_Temporarily_Unavailable`

The mapping could not be locked in memory, if required by `Lock_All` due to a lack of resources.

`Bad_File_Descriptor`

The `File` argument is not a valid open file descriptor.

`Invalid_Argument`

The value in `Mapping` is invalid (*i.e.*, not equal to either `Map_Private` or `Map_Shared`).

The value in `Location` is invalid (*i.e.*, not equal to either `Exact_Address` or `Nearby_Address`).

`No_Such_Operation_On_Device`

The `File` argument refers to an object for which `Map_Memory` is meaningless, such as a terminal.

`Not_Enough_Space`

`Exact_Address` was specified,, and the address range starting at `First` and continuing for `Length` storage units exceeds that allowed for the address space of a process; or `Exact_Address` was not specified and there is insufficient room in the address space to effect the mapping.

The mapping could not be locked in memory, if required by `Lock_All`, because it would require more space than the system is able to supply.

`Operation_Not_Implemented`

The operation `Map_Memory` is not supported by this implementation.

`Operation_Not_Supported`

`Exact_Address` was specified in the `Location` parameter and the implementation does not support this functionality.

`Map_Private` was specified in the `Mapping` parameter, and the implementation does not support this functionality.

The implementation does not support the combination of accesses requested in the `Protection` parameter.

`No_Such_Device_Or_Address`

The addresses in the range starting at `Offset` storage units and continuing for `Length` storage units are invalid for the object specified by `File`.

`Exact_Address` was specified in `Location`, and the combination of `First`, `Length`, and `Offset` is invalid for the object specified by `File`.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The arguments First (if Exact_Address was specified) or Offset are not multiples of Page Size, and the implementation imposes a restriction that these values be page-aligned.

If Map_Memory fails for reasons other than Bad_File_Descriptor, Invalid_Argument, or Operation_Not_Supported, some of the mappings in the address range starting at First and continuing for Length storage units may have become unmapped.

12.3.1.4 Required Representation Support and Shared Variable Control

If either the Memory Mapped Files option or the Shared Memory Objects option is supported, for all regions of the address space of the callig process that are currently mapped to a memory object by Map_Memory, POSIX_Generic_Shared_Memory.Open_And_Map_Shared_Memory, or POSIX_Generic_Shared_Memory.Open_Or_Create_And_Map_Shared_Memory, the following apply::

— The implementation shall support at a minimum the recommended levels of representation support specified by C.2 of the Ada RM {1}.

— The implementation shall support the Atomic, Atomic_Components, Volatile, and Volatile_Components pragmas specified by C.6 of the Ada RM {1}.

12.3.2 Unmap Memory

12.3.2.1 Synopsis

```
procedure Unmap_Memory
   (First:  in System.Address;
    Length: in System_Storage_Elements.Storage_Offset);
```

12.3.2.2 Description

The functionality described in this subclause is optional. If neither the Memory Mapped Files option nor the Shared Memory Objects option is supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

Unmap_Memory removes any mappings for the whole pages containing any part of the address space of the calling process starting at First and continuing for Length storage units. Further references to these pages shall result in the exception Program_Error being raised. If no mappings are in the specified address range, then Unmap_Memory shall have no effect. The implementation may require that First be a multiple of Page Size.

NOTE: It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of First.

If a mapping to be removed was private, any modifications made in this address range shall be discarded.

Any memory locks (see 12.1.1 and 12.2.1) associated with this address range shall be removed, as if by an appropriate call to Unlock_All or Unlock_Range.

The behavior of this operation is unspecified if the mapping was not established by a call to Map_Memory.

12.3.2.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: Some of the addresses in the range starting at First and continuing for Length storage units are outside the range allowed for the address space of a process.

Operation_Not_Implemented
: The procedure Unmap_Memory is not supported by this implementation.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: The value of First is not a multiple of Page Size, and the implementation imposes a restriction that First be page-aligned.

12.3.3 Change Memory Protection

12.3.3.1 Synopsis

```
procedure Change_Protection
   (First:      in System.Address;
    Length:     in System_Storage_Elements.Storage_Offset;
    Protection: in Protection_Options);
```

12.3.3.2 Description

The functionality described in this subclause is optional. If the Memory Protection option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

Change_Protection changes the access protections to be that specified by Protection for the whole pages containing any part of the address space of the process starting at address First and continuing for Length storage units. The parameter Protection determines whether read, write, execute, or a combination of accesses is permitted to the data being mapped. The Protection argument shall be either the Empty_Set or the union ("+") of one or more of the other constants of type Protection_Options.

If an implementation cannot support the combination of access types specified by Protection, the call to Change_Protection shall fail. An implementation may permit accesses other than those specified by Protection; however, no implementation shall permit a write to succeed where Allow_Write has not been set or permit

any access where Protection is set to the Empty_Set. The implementation shall support at least the following values of Protection: the Empty_Set, Allow_Read, Allow_Write, and the union ("+") of Allow_Read and Allow_Write.

If Allow_Write is specified, the application is required to have opened the mapped objects in the specified address range with write permission, unless Map_Private was specified in the original mapping, regardless of whether the file descriptors used to map the objects have since been closed.

The implementation may require that First be a multiple of Page Size.

NOTE: It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of First.

The behavior of this operation is unspecified if the mapping was not established by a call to Map_Memory.

12.3.3.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Permission_Denied

The memory object was not opened for read, regardless of the protection specified.

The memory object was not opened for write, and Allow_Write was specified for a Map_Shared type mapping.

Resource_Temporarily_Unavailable

The Protection argument specifies Allow_Write on a Map_Private mapping and there are insufficient memory resources to reserve for locking the private pages, if required.

Not_Enough_Space

The addresses in the range starting at First and continuing for Length storage units are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.

The Protection argument specifies Allow_Write on a Map_Private mapping, and it would require more space than the system is able to supply for locking the private pages, if required.

Operation_Not_Implemented

Change_Protection is not supported by this implementation.

Operation_Not_Supported

The implementation does not support the combination of accesses requested in the Protection argument.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The value of First is not a multiple of Page Size, and the implementation imposes a restriction the First be page-aligned.

If Change_Protection fails for reasons other than Invalid_Argument, the protections on some of the pages in the address range starting at First and continuing for Length storage units may have been changed.

12.3.4 Memory Object Synchronization

12.3.4.1 Synopsis

```
type Synchronize_Memory_Options is new POSIX.Option_Set;
Wait_For_Completion:    constant Synchronize_Memory_Options
   := implementation-defined;
Invalidate_Cached_Data: constant Synchronize_Memory_Options
   := implementation-defined;
procedure Synchronize_Memory
   (First:   in System.Address;
    Length:  in System_Storage_Elements.Storage_Offset;
    Options: in Synchronize_Memory_Options:= Empty_Set);
```

12.3.4.2 Description

The functionality described in this subclause is optional. If either the Memory Mapped Files option or the Synchronized I/O option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

The Synchronize_Memory operation writes all modified data to permanent storage locations, if any, in the whole pages containing any part of the address space of the process, starting at address First and continuing for Length storage units. If no such storage exists, Synchronize_Memory need not have any effect. If requested, the Synchronize_Memory operation then invalidates cached copies of data.

The implementation may require that First be a multiple of Page Size.

NOTE: It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of First.

For mappings to files, the Synchronize_Memory procedure shall assure that all write operations are completed as defined for synchronized I/O data integrity completion. It is unspecified whether the implementation also writes out other file attributes. When the Synchronize_Memory procedure is called on Map_Private mappings, any modified data shall not be written to the underlying object and shall not cause such data to be made visible to other processes. It is unspecified whether data in Map_Private mappings have any permanent storage locations. The effect of Synchronize_Memory on shared memory objects is unspecified.

If Wait_For_Completion is not specified in Options, Synchronize_Memory returns immediately once all the write operations are initiated or queued for servicing. When Wait_For_Completion is specified in Options, Synchronize_Memory shall not return until all write operations are completed as defined for synchronized I/O data integrity completion.

When Invalidate_Cached_Data is specified in Options, Synchronize_Memory invalidates all cached copies of mapped data that are inconsistent with the permanent storage locations so that subsequent references shall obtain data that was consistent with the permanent storage locations sometime between the call to Synchronize_Memory and the first subsequent memory reference to the data.

The behavior of this operation is unspecified if the mapping was not established by a call to Map_Memory.

12.3.4.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Busy
: Some or all of the addresses in the range starting at First and continuing for Length storage units are locked, and Invalidate_Cached_Data is specified in Options.

Invalid_Argument
: The value in Options is invalid.

Not_Enough_Space
: The addresses in the range starting at First and continuing for Length storage units are outside the range allowed for the address space of a process, or specify one or more pages that are not mapped.

Operation_Not_Implemented
: The procedure Synchronize_Memory is not supported by this implementation.

If any of the following conditions is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument
: The value of First is not a multiple of Page Size, and the implementation imposes a restriction that First be page-aligned.

12.4 Package POSIX_Shared_Memory_Objects

This package provides access to services that a process can use to create, access, and remove memory objects that can be shared between processes.

The functionality described in this subclause is optional. If the Shared Memory Objects option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

```
with POSIX,
     POSIX_IO,
     POSIX_Permissions;
package POSIX_Shared_Memory_Objects is
    -- 12.4.1 Open a Shared Memory Object
   function Open_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Mode:           POSIX_IO.File_Mode;
       Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
```

```
   function Open_Or_Create_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Mode:           POSIX_IO.File_Mode;
       Permissions:    POSIX_Permissions.Permission_Set;
       Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
   -- 12.4.2 Remove a Shared Memory Object
   procedure Unlink_Shared_Memory
      (Name: in POSIX.POSIX_String);

end POSIX_Shared_Memory_Objects;
```

12.4.1 Open a Shared Memory Object

12.4.1.1 Synopsis

```
   function Open_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Mode:           POSIX_IO.File_Mode;
       Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
   function Open_Or_Create_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Mode:           POSIX_IO.File_Mode;
       Permissions:    POSIX_Permissions.Permission_Set;
       Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
```

12.4.1.2 Description

The Open_Shared_Memory and Open_Or_Create_Shared_Memory operations establish a connection between a shared memory object and a file descriptor. Open_Shared_Memory shall open a shared memory object that already exists. Open_Or_Create_Shared_Memory shall either open an existing shared memory object or create a new shared memory object if none exists. Both functions create an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that shared memory object.

The Name argument specifies a string naming a shared memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The Name argument shall conform to the construction rules for a pathname. If Name begins with the slash character, then processes calling Open_Shared_Memory and Open_Or_Create_Shared_Memory with the same value of Name shall refer to the same shared memory object, as long as that name has not been removed. If Name does not begin with the slash character, the effect is implementation defined. The interpretation of slash characters other than the leading slash character in Name is implementation defined.

If successful, Open_Shared_Memory and Open_Or_Create_Shared_Memory return a file descriptor for the shared memory object that is the lowest numbered file

descriptor not currently open for that process. The open file description is new; therefore, the file descriptor does not share it with any other processes. It is unspecified whether the file offset is set. The new file descriptor shall be marked so that `POSIX_IO.Get_Close_On_Exec` would return `True`.

`Mode` specifies the file access modes of the open file description, using some of the values of type `POSIX_IO.File_Mode`. Applications shall specify exactly one of the following two values of type `POSIX_IO.File_Mode`:

`Read_Only`

Open for read access only.

`Read_Write`

Open for read or write access.

`Open_Shared_Memory` uses only one value from `POSIX_IO.Open_Option_Set` in `Options`. The effect of setting other values from `POSIX_IO.Open_Option_Set` in `Options` is undefined.

`Truncate`

If the shared memory object is successfully opened `Read_Write`, the object shall be truncated to zero length, and the mode and owner shall be unchanged by this function call. The result of using `Truncate` with `Read_Only` is undefined.

`Open_Or_Create_Shared_Memory` either opens an existing shared memory object, except as noted under `Exclusive` below, or creates the shared memory object. If the shared memory object is created, the user ID of the shared memory object shall be set to the effective user ID of the process; the shared memory group ID of the shared memory object shall be set to a system default group ID or to the effective group ID of the process. The permissions of the shared memory object shall be set to the value of the `Permissions` argument except those set in the file mode creation mask of the process. The `Permissions` argument does not affect whether the shared memory object is opened for reading, for writing, or for both. The shared memory object shall have a size of zero. `Open_Or_Create_Shared_Memory` uses either or both of the following values from `POSIX_IO.Open_Option_Set` in `Options`. The effect of setting other values from `POSIX_IO.Open_Option_Set` in `Options` is undefined:

`Exclusive`

If `Exclusive` is set, `Open_Or_Create_Shared_Memory` shall fail if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist shall be atomic with respect to other processes executing `Open_Or_Create_Shared_Memory` naming the same shared memory object with `Exclusive` set.

`Truncate`

If the shared memory object exists and it is successfully opened `Read_Write`, the object shall be truncated to zero length, and the mode and owner shall be unchanged by this function call. The result of using `Truncate` with `Read_Only` is undefined.

`Open_Shared_Memory` and `Open_Or_Create_Shared_Memory` shall be interruptible by the delivery of a signal. `Masked_Signals` specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, shall persist until the shared memory object is unlinked and all other references are gone. It is unspecified whether the name and shared memory object state remain valid after a system reboot.

NOTE: The `POSIX_IO.Close` operation, which is specified in 6.1.1, is used to close a shared memory object.

12.4.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

The shared memory object exists, and the permissions specified by `Permissions` are denied; or the shared memory object does not exist, and permission to create the shared memory object is denied; or `Truncate` is specified, and write permission is denied.

`File_Exists`

Upon `Open_Or_Create_Shared_Memory`, `Exclusive` is specified, and the named shared memory object already exists.

`Interrupted_Operation`

The `Open_Shared_Memory` or `Open_Or_Create_Shared_Memory` operation was interrupted by a signal.

`Invalid_Argument`

The `Open_Shared_Memory` or `Open_Or_Create_Shared_Memory` operation is not supported for the given name. The implementation shall document under what circumstances this error may be returned.

`Too_Many_Open_Files`

Too many file descriptors are currently in use by this process.

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`Too_Many_Open_Files_In_System`

Too many shared memory objects are currently open in the system.

`No_Such_File_Or_Directory`

Upon `Open_Shared_Memory`, the named shared memory object does not exist.

`No_Space_Left_On_Device`

There is insufficient space for the creation of the new shared memory object.

`Operation_Not_Implemented`

`Open_Shared_Memory` and `Open_Or_Create_Shared_Memory` are not supported by this implementation.

12.4.2 Remove a Shared Memory Object

12.4.2.1 Synopsis

```
procedure Unlink_Shared_Memory
   (Name: in POSIX.POSIX_String);
```

12.4.2.2 Description

`Unlink_Shared_Memory` shall remove the name of the shared memory object specified by `Name`. If one or more references to the shared memory object exist when the object is unlinked, the name shall be removed before `Unlink_Shared_Memory` returns. However, the removal of the memory object contents shall be postponed until all open file descriptors connected to the shared memory object (see 12.4.1) have been closed and all mappings established for the shared memory object have been removed (see 12.3.2).

12.4.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

Permission is denied to unlink the named shared memory object.

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`No_Such_File_Or_Directory`

The named shared memory object does not exist.

`Operation_Not_Implemented`

`Unlink_Shared_Memory` is not supported by this implementation.

12.5 Package `POSIX_Generic_Shared_Memory`

An instantiation of the generic package `POSIX_Generic_Shared_Memory` provides operations for managing shared memory objects of the actual type corresponding to the generic formal type `Object_Type`. This type shall be a constrained type such that the representation of an object `X` is contained within the contiguous memory region covered by the range `X'Address` through `X'Address + (X'Size / System.Storage_Unit)`.

The functionality described in this subclause is optional. If the Shared Memory Objects option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

```
with POSIX,
     POSIX_IO,
     POSIX_Permissions,
     POSIX_Memory_Mapping;
generic
  type Object_Type is private;
package POSIX_Generic_Shared_Memory is
  type Shared_Access is access Object_Type;
   -- 12.5.1 Open Shared Memory
   function Open_And_Map_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Protection:     POSIX_Memory_Mapping.Protection_Options;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
   function Open_Or_Create_And_Map_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Protection:     POSIX_Memory_Mapping.Protection_Options;
       Permissions:    POSIX_Permissions.Permission_Set;
       Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
   -- 12.5.2 Access Shared Memory
   function Access_Shared_Memory
      (File: POSIX_IO.File_Descriptor)
     return Shared_Access;
   -- 12.5.3 Close Shared Memory
   procedure Unmap_And_Close_Shared_Memory
      (File: in POSIX_IO.File_Descriptor);
   -- 12.5.5 Lock/Unlock Shared Memory
   procedure Lock_Shared_Memory
      (File: in POSIX_IO.File_Descriptor);
   procedure Unlock_Shared_Memory
      (File: in POSIX_IO.File_Descriptor);

end POSIX_Generic_Shared_Memory;
```

12.5.1 Open Shared Memory

12.5.1.1 Synopsis

```
   function Open_And_Map_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Protection:     POSIX_Memory_Mapping.Protection_Options;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
   function Open_Or_Create_And_Map_Shared_Memory
      (Name:           POSIX.POSIX_String;
       Protection:     POSIX_Memory_Mapping.Protection_Options;
       Permissions:    POSIX_Permissions.Permission_Set;
       Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return POSIX_IO.File_Descriptor;
```

12.5.1.2 Description

The Open_And_Map_Shared_Memory operation and the Open_Or_Create_And_-Map_Shared_Memory operation shall open or create a POSIX shared memory object that holds one Ada object of the actual type corresponding to the generic formal type Object_Type. The shared memory object is mapped into the address space of the process at an address defined by the system. These operations shall return a file descriptor designating the open shared memory object.

If the actual type corresponding to Object_Type has default initialization, the effect is undefined.

The effect of Open_And_Map_Shared_Memory shall be equivalent to a sequence of calls to the three following operations, which are defined elsewhere in this standard, using the supplied parameter values together with the listed implicit parameter values:

— Open_Shared_Memory to open a POSIX shared memory object. If the value of Protection is set to Allow_Write, Mode is Read_Write; otherwise Mode is Read_Only.

— POSIX_IO.Truncate_File to allocate enough memory within the POSIX shared memory object to hold the shared Ada object. The File parameter corresponds to the value returned by Open_Shared_Memory if that operation is successful. The Length parameter shall specify a count of storage units sufficient to contain an Ada object of the generic actual type Object_Type.

— Map_Memory to map the POSIX shared memory object into the address space of the process. The File parameter corresponds to the value returned by Open_-Shared_Memory if that operation is successful. The Length parameter shall be chosen to cover the size of the Ada object of type Object_Type that is created. The Mapping parameter is set to Map_Shared. The Location parameter is not set to Exact_Address.

The effect of Open_Or_Create_And_Map_Shared_Memory shall be equivalent to a sequence of calls to the three following operations that are defined elsewhere in this standard, using the supplied parameter values together with the listed implicit parameter values:

— Open_Or_Create_Shared_Memory to open a POSIX shared memory object. If the value of Protection is set to Allow_Write, Mode is Read_Write; otherwise Mode is Read_Only. If Exclusive is specified on a call to Open_Or_Create_And_Map_Shared_Memory, then its effect is to specify Exclusive on the corresponding call to Open_Or_Create_Shared_Memory.

— POSIX_IO.Truncate_File to allocate enough memory within the POSIX shared memory object to hold the shared Ada object. The File parameter corresponds to the value returned by Open_Or_Create_Shared_Memory if that operation is successful. The Length parameter shall specify a count of storage units sufficient to contain an Ada object of the generic actual type Object_Type.

— Map_Memory to map the POSIX shared memory object into the address space of the process. The File parameter corresponds to the value returned by Open_-Or_Create_Shared_Memory, if that operation is successful. The First and

Length parameters shall be chosen to cover the virtual addresses occupied by the Ada object of type Object_Type that is created. The Mapping parameter is set to Map_Shared. The Location parameter is not set to Exact_Address.

Open_Or_Create_And_Map_Shared_Memory uses only one value from POSIX_IO.Open_Option_Set in Options. The effect of setting other values from POSIX_IO.Open_Option_Set in Options is undefined.

Exclusive

If Exclusive is set, Open_Or_Create_And_Map_Shared_Memory shall fail if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist shall be atomic with respect to other processes executing Open_Or_Create_And_Map_Shared_Memory naming the same shared memory object with Exclusive set.

If any operation on the type POSIX_IO.File_Descriptor that is not specified in 12.5 is applied to a file descriptor returned by Open_And_Map_Shared_Memory or Open_Or_Create_And_Map_Shared_Memory, the effect of that and subsequent operations on that file descriptor is undefined.

Open_And_Map_Shared_Memory and Open_Or_Create_And_Map_Shared_Memory shall be interruptible by the delivery of a signal. Masked_Signals specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

If the operation propagates an exception, it shall have no residual effect. In particular, no new shared memory object shall exist, no new memory mapping shall exist, and no new file descriptor shall be allocated.

12.5.1.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Filename_Too_Long

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

File_Exists

Open_Or_Create_Shared_Memory was called with Exclusive specified in Options, and the named shared memory object already exists.

Interrupted_Operation

The operation was interrupted by a signal.

Invalid_Argument

Permissions is the empty set.

The operation is not supported for the given name. The implementation shall document under what circumstances this error may be returned.

`No_Such_File_Or_Directory`

`Open_Shared_Memory` was called, and the named shared memory object does not exist.

`No_Space_Left_On_Device`

There is insufficient space for the creation of the new shared memory object.

`Not_Enough_Space`

There is insufficient room in the address space to effect the mapping; or the mapping could not be locked in memory, if required by `Lock_All`, because it would require more space than the system is able to supply.

`Operation_Not_Implemented`

The operation is not supported by this implementation.

`Operation_Not_Supported`

The implementation does not support the combination of accesses requested in the `Protection` argument.

`Permission_Denied`

The shared memory object exists, and the operation requests a mode of access that is denied; or the shared memory object does not exist, and permission to create the shared memory object is denied; or `POSIX_IO.Truncate` is specified, and write permission is denied.

`Resource_Temporarily_Unavailable`

The mapping could not be locked in memory, if required by `Lock_All`, due to a lack of resources.

`Too_Many_Open_Files`

Too many file descriptors are currently in use by this process.

`Too_Many_Open_Files_In_System`

Too many shared memory objects are currently open in the system.

12.5.2 Access Shared Memory

12.5.2.1 Synopsis

```
function Access_Shared_Memory
   (File: POSIX_IO.File_Descriptor)
  return Shared_Access;
```

12.5.2.2 Description

If `File` is an open file descriptor returned by a call to `Open_And_Map_Shared_Memory` or `Open_Or_Create_And_Map_Shared_Memory`, then a call to `Access_Shared_Memory` of the same generic instantiation shall return an access value that designates the Ada object of type `Object_Type` within the shared memory object designated by the file descriptor.

12.5.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: `File` is not the file descriptor for an open shared memory object defined by this generic instantiation.

12.5.3 Close Shared Memory

12.5.3.1 Synopsis

```
procedure Unmap_And_Close_Shared_Memory
   (File: in POSIX_IO.File_Descriptor);
```

12.5.3.2 Description

If `File` is an open file descriptor returned by a call to `Open_And_Map_Shared_Memory` or `Open_Or_Create_And_Map_Shared_Memory`, then a call to `Unmap_And_Close_Shared_Memory` of the same generic instantiation shall unmap and close the shared memory object designated by the file descriptor.

The effect of this operation shall correspond to a series of calls to the following operations, which are defined elsewhere in this standard:

(1) A call to `Unmap_Memory` to unmap the shared object from the address space of the process.

(2) A call to `POSIX_IO.Close` to close the file descriptor.

12.5.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: `File` is not the file descriptor for an open shared memory object defined by this generic instantiation.

12.5.4 Remove Shared Memory

There is no procedure to unlink a shared memory object defined in this package. The `Unlink_Shared_Memory` procedure defined in `POSIX_Shared_Memory_Objects` package shall apply to these shared memory objects as well as other shared memory objects.

12.5.5 Lock/Unlock Shared Memory

12.5.5.1 Synopsis

```
procedure Lock_Shared_Memory
   (File: in POSIX_IO.File_Descriptor);
procedure Unlock_Shared_Memory
   (File: in POSIX_IO.File_Descriptor);
```

12.5.5.2 Description

The functionality described in this subclause is optional. If the Memory Range Locking option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

If `File` is an open file descriptor returned by a call to `Open_And_Map_Shared_Memory` or `Open_Or_Create_And_Map_Shared_Memory`, a call to `Lock_Shared_Memory` or `Unlock_Shared_Memory` of the same generic instantiation shall be equivalent to a call to `Lock_Range` or `Unlock_Range` for the range of memory that is mapped to the shared memory object designated by the file descriptor.

NOTE: These operations do not provide locking in the sense of protection against concurrent access. The meaning of lock here is that the range of virtual addresses mapped by the shared memory object are locked into main memory; they cannot be paged out onto a secondary storage device.

12.5.5.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`

: `File` is not the file descriptor for a shared memory object defined by this generic instantiation.

`Operation_Not_Implemented`

: The implementation does not support this memory-locking operation.

`Resource_Temporarily_Unavailable`

: Some or all of the memory mapped to the shared memory object could not be locked when the call was made.

`Not_Enough_Space`

: Locking the pages mapped by the specified shared memory object would exceed an implementation-defined limit on the amount of memory that the process can lock.

`Operation_Not_Permitted`

: The calling process does not have the appropriate privilege to perform the requested operation.

Section 13: Execution Scheduling

This section describes the facilities available under this standard by which an application can influence the scheduling of Ada tasks.

NOTE: This section is a thinner binding to the base standards than the rest of this standard. See B.13 for more information.

13.1 Scheduling Concepts and Terminology

The definitions of several terms used in this section, including blocked task, environment task, priority, ready task, task, thread of control, and process, are given in 2.2. Some other terms are defined by reference to POSIX.1.

The Scheduling Contention Scope attribute of a task shall correspond to the scheduling contention scope of the associated thread of control in POSIX.1 (see Section 13 of POSIX.1 {1}). The Scheduling Contention Scope of a task defines the set of threads of control (not necessarily limited to Ada tasks) with which the task must compete for use of processing resources. The scheduling contention scope names `System_Wide` and `Within_Process` shall correspond, respectively, to the names PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS in POSIX.1.

13.2 Package `POSIX_Process_Scheduling`

This package provides access to services for influencing the scheduling at the process level.

The functionality described in this clause is optional. If the Priority Process Scheduling option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this clause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this clause.

```
with POSIX,
     POSIX_Process_Identification;
package POSIX_Process_Scheduling is
   -- 13.2.1 Scheduling Parameters
   subtype Scheduling_Priority is Integer;
   type Scheduling_Parameters is private;
   function Get_Priority (Parameters: Scheduling_Parameters)
     return Scheduling_Priority;
   procedure Set_Priority
      (Parameters: in out Scheduling_Parameters;
       Priority:   in Scheduling_Priority);
   -- 13.2.2 Scheduling Policies
   type Scheduling_Policy is range implementation-defined;
   Sched_FIFO:  constant Scheduling_Policy:= implementation-defined;
   Sched_RR:    constant Scheduling_Policy:= implementation-defined;
   Sched_Other: constant Scheduling_Policy:= implementation-defined;
   -- 13.2.3 Modify Process Scheduling Policy and Parameters
   procedure Set_Scheduling_Parameters
      (Process:    in POSIX_Process_Identification.Process_ID;
       Parameters: in Scheduling_Parameters);
   function Get_Scheduling_Parameters
      (Process:    POSIX_Process_Identification.Process_ID)
     return Scheduling_Parameters;
```

```
   procedure Set_Scheduling_Policy
      (Process:    in  POSIX_Process_Identification.Process_ID;
       New_Policy: in  Scheduling_Policy;
       Parameters: in  Scheduling_Parameters);
   function Get_Scheduling_Policy
      (Process: POSIX_Process_Identification.Process_ID)
     return Scheduling_Policy;
   -- 13.2.4 Process Yield CPU
   procedure Yield;
   -- 13.2.5 Get Scheduling Limits
   function Get_Maximum_Priority (Policy: Scheduling_Policy)
     return Scheduling_Priority;
   function Get_Minimum_Priority (Policy: Scheduling_Policy)
     return Scheduling_Priority;
   function Get_Round_Robin_Interval
      (Process: POSIX_Process_Identification.Process_ID)
     return POSIX.Timespec;
private
   implementation-defined
end POSIX_Process_Scheduling;
```

13.2.1 Scheduling Parameters

13.2.1.1 Synopsis

```
   subtype Scheduling_Priority is Integer;
   type Scheduling_Parameters is private;
   function Get_Priority (Parameters: Scheduling_Parameters)
     return Scheduling_Priority;
   procedure Set_Priority
      (Parameters: in out Scheduling_Parameters;
       Priority:   in Scheduling_Priority);
```

13.2.1.2 Description

Values of the subtype `Scheduling_Priority` are used to specify the Scheduling Priority attribute of processes. The range of priority values that are valid for scheduling processes is implementation defined.

Values of the type `Scheduling_Parameters` are used to specify the per-process Scheduling Parameters required for each scheduling policy supported. This type shall support at least one attribute, Scheduling Priority, of type `Scheduling_Priority`.

`Set_Priority` shall set the Scheduling Priority attribute of the object specified by `Parameters` to the value specified by `Priority`.

`Get_Priority` shall return the Scheduling Priority attribute of the object specified by `Parameters`.

Implementations may add other attributes, as permitted in 1.3.1.1.
NOTE: Adding attributes that may change the behavior of applications with respect to this standard when those attributes are uninitialized also requires that the extension be activated as described in 1.3.1.1.

Operations for getting and setting the values of implementation-defined attributes of objects of the `Scheduling_Parameters` type may be defined in separate extension packages. The names of these operations shall be of the form `Get_X` and `Set_X`, where `X` is the name of the attribute.

13.2.2 Scheduling Policies

13.2.2.1 Synopsis

```
type Scheduling_Policy is range implementation-defined;
Sched_FIFO:  constant Scheduling_Policy:= implementation-defined;
Sched_RR:    constant Scheduling_Policy:= implementation-defined;
Sched_Other: constant Scheduling_Policy:= implementation-defined;
```

13.2.2.2 Description

Values of the type Scheduling_Policy are used to specify process scheduling policies. The names Sched_FIFO, Sched_RR, and Sched_Other shall correspond, respectively, to the scheduling policy names SCHED_FIFO, SCHED_RR, and SCHED_OTHER in POSIX.1 (see 13.2 of {2}).

Other values of type Scheduling_Policy may be implementation defined. The names of the implementation-defined policies shall begin with "Sched_".

13.2.3 Modify Process Scheduling Policy and Parameters

13.2.3.1 Synopsis

```
procedure Set_Scheduling_Parameters
   (Process:    in POSIX_Process_Identification.Process_ID;
    Parameters: in Scheduling_Parameters);
function Get_Scheduling_Parameters
   (Process:    POSIX_Process_Identification.Process_ID)
  return Scheduling_Parameters;
procedure Set_Scheduling_Policy
   (Process:    in  POSIX_Process_Identification.Process_ID;
    New_Policy: in  Scheduling_Policy;
    Parameters: in  Scheduling_Parameters);
function Get_Scheduling_Policy
   (Process: POSIX_Process_Identification.Process_ID)
  return Scheduling_Policy;
```

13.2.3.2 Description

Set_Scheduling_Parameters shall set the Scheduling Parameters of the process specified by Process to the value specified by Parameters, provided that such a process exists and the calling process has permission.

Set_Scheduling_Policy shall set the Scheduling Policy and scheduling parameters of the process specified by Process to the values specified by New_Policy and Parameters, respectively, provided that such a process exists and the calling process has permission.

The effect of Set_Scheduling_Parameters and Set_Scheduling_Policy on individual tasks within the process is dependent on the Scheduling Contention Scope of the tasks. (See 13.1.)

— For tasks with System_Wide Scheduling Contention Scope, these operations shall have no effect on their scheduling.

— For tasks with `Within_Process` scheduling contention scope, scheduling with respect to threads of control in other processes may be dependent on the scheduling policy and scheduling parameters of their process, which is governed using these functions.

`Get_Scheduling_Parameters` shall return the scheduling parameters of the process specified by `Process`, provided that such a process exists and the calling process has permission.

`Get_Scheduling_Policy` shall return the Scheduling Policy of the process specified by `Process`, provided that such a process exists and the calling process has permission.

For `Set_Scheduling_Parameters` and `Set_Scheduling_Policy`, the following apply:

— Implementations may require that the requesting process have permission to set its own scheduling parameters or those of another process. Additionally, implementation defined restrictions may apply to the appropriate privilege required to set the Scheduling Policy of a process or the Scheduling Policy of another process to a particular value.

— If the new Scheduling Policy of the process is `Sched_FIFO` or `Sched_RR`, only the Scheduling Priority attribute is required to be supported. In this case, the Scheduling Priority scheduling parameter is required to be within the range of valid priorities for the new scheduling policy.

— If the new Scheduling Policy is `Sched_Other`, the affected scheduling parameters are implementation defined.

— If the operation fails, it shall propagate an exception, and the Scheduling Policy and scheduling parameters of the specified process shall not have been affected.

NOTE: A call to `Set_Scheduling_Parameters` or `Set_Scheduling_Policy` may result in preemption of the calling task before the call returns.

13.2.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

One or more of the requested scheduling parameters are outside the range defined for the specified Scheduling Policy of `Process`.

The value of the `Policy` parameter is invalid, or one or more of the attributes of `Parameters` is outside the valid range for the specified scheduling policy.

`Operation_Not_Implemented`

The operation is not supported by this implementation.

`Operation_Not_Permitted`

The requesting process does not have permission to perform the operation for the specified process.

The requesting process does not have permission to invoke `Set_Scheduling_Parameters`.

No_Such_Process

No process can be found corresponding to that specified by Process.

13.2.4 Process Yield CPU

13.2.4.1 Synopsis

```
procedure Yield;
```

13.2.4.2 Description

The Yield operation shall cause the calling task to yield the processor to other tasks of equal process or task priority. The specific effects are implementation defined.

NOTE: This feature is provided to allow voluntary sharing of processing resources between cooperating processes of equal priority, under the Sched_FIFO policy, on systems where tasks are implemented via interleaved execution of a per-process kernel entity.

NOTE: See 13.3.4 for a task-level yield operation.

13.2.4.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Implemented

The Yield operation is not supported by this implementation.

13.2.5 Get Scheduling Limits

13.2.5.1 Synopsis

```
function Get_Maximum_Priority (Policy: Scheduling_Policy)
  return Scheduling_Priority;
function Get_Minimum_Priority (Policy: Scheduling_Policy)
  return Scheduling_Priority;
function Get_Round_Robin_Interval
   (Process: POSIX_Process_Identification.Process_ID)
  return POSIX.Timespec;
```

13.2.5.2 Description

Get_Maximum_Priority and Get_Minimum_Priority shall return the maximum and minimum priority values that are valid for the policy specified by Policy.

Get_Round_Robin_Interval shall return the current *round-robin-interval* of the process specified by Process. The value returned represents the quantum under the Sched_RR policy at which a scheduling decision shall be made when another process at the same priority is ready to execute. (See 13.2.2 of {2}.)

13.2.5.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`
: The value of the `Policy` parameter does not represent a defined scheduling policy.

`Operation_Not_Implemented`
: The functions `Get_Maximum_Priority`, `Get_Minimum_Priority`, and `Get_Round_Robin_Interval` are not supported by this implementation.

`No_Such_Process`
: No process can be found corresponding to that specified by `Process`.

13.3 Task Scheduling

The functionality described in this clause is optional. If the Priority Task Scheduling option is not supported, the implementation may ignore or reject the pragmas defined here, or raise `POSIX_Error` for tasks to which an unsupported pragma is applied. Likewise, if the Priority Task Scheduling option is not supported, the implementation may cause all calls to the explicitly declared operations defined in the packages specified in this clause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified below.

A prerequisite for support of the Priority Task Scheduling option (see 2.5.1) is support for the priority model defined in D.1 of the Ada RM {1} and the pragmas and package interfaces defined in D.2-D.5 of the Ada RM {1}.

13.3.1 Dynamic Priorities

NOTE: The initial base priority of a task may be specified via the `Priority` pragma, defined in D.1 of the Ada RM {1}. Thereafter, the base priority may be modified by means of calls to the procedure `Set_Priority`, which is defined in `Ada.Dynamic_Priorities` in D.5 of the Ada RM {1}.

Implementations of this standard that support child library packages 10.1 of the Ada RM {1} shall support the package `Ada.Dynamic_Priorities` and provide a library-level renaming of that package as `Ada_Dynamic_Priorities`.

During transition from Ada 83 to Ada 95, implementations of this standard that do not support child library units shall provide an equivalent library-level package named `Ada_Dynamic_Priorities` with the same visible declarations as `Ada.Dynamic_Priorities`.

13.3.2 Task Dispatching Policy Pragma

The Ada implementation shall support use of the `Task_Dispatching_Policy` pragma D.2.2 of the Ada RM {1} with the task dispatching policy identifier `POSIX_Task_Dispatching`. If this policy is selected for a partition, the effect shall be that tasks are dispatched according to the scheduling policies defined in POSIX.1 (see Section 13 of {2}) for some implementation-defined correspondence of Ada tasks to POSIX threads.

NOTE: The effect of the `Task_Dispatching_Policy` *pragma is to specify the policy by which all the tasks in a process are scheduled. In contrast, the effect of the* `Task_Creation_Attributes` *pragma is to specify the scheduling policy for tasks of just one type. Thus, although the rules for the Ada* `FIFO_Within_Priorities` *task dispatching policy are very similar to the rules for the POSIX* `Sched_FIFO` *task scheduling policy, one should not be confused with the other. The former is process wide and the latter is task-specific.*

13.3.3 Task Creation Attributes Pragma

The Ada implementation shall support the `Task_Creation_Attributes` pragma. The `Task_Creation_Attributes` pragma is allowed within a task specification, in which case it shall apply to all tasks of the specified task type. Only one occurrence of the pragma is allowed in each task specification.

The `Task_Creation_Attributes` pragma shall be recognized at least for compilations that have specified `POSIX_Task_Dispatching` as the partition wide task dispatching policy. Otherwise, it may be rejected or not recognized.

The `Task_Creation_Attributes` pragma shall allow (at least) argument associations with argument identifiers `Scheduling_Contention_Scope` and `Scheduling_Policy`, in any order. For the argument identifier `Scheduling_Contention_Scope`, at least the names `System_Wide` and `Within_Process` shall be recognized. For the argument identifier `Scheduling_Policy`, at least the names `Sched_FIFO`, `Sched_RR`, and `Sched_Other` shall be recognized.

An implementation may support implementation-defined additional argument identifiers for the `Task_Creation_Attributes` pragma, and additional scheduling policy names. Scheduling policy names shall begin with "`Sched_`".

13.3.3.1 Error Handling

If the `Task_Creation_Attributes` pragma is not supported, or the specified combination of arguments is not supported, then the pragma shall not be recognized, or the compilation shall be rejected, or `POSIX_Error` shall be raised during the creation or activation of the task to which the pragma is applied.

If any of the following conditions is detected during the creation or activation of a task containing the `Task_Creation_Attributes` pragma, the exception `POSIX_Error` shall be raised, with the corresponding error code:

`Invalid_Argument`
: The name or expression specified for a supported argument identifier is not valid.

`Operation_Not_Supported`
: The combination of argument associations given in the pragma is not supported.

13.3.4 Task Yield CPU

Execution of a delay statement with duration zero shall cause the calling task to yield the processor to other ready tasks of equal priority. The specific effects are implementation defined.

13.4 Synchronization Scheduling

This clause specifies the interactions between the synchronization operations of Section 11 and priority task scheduling.

When a task owns a mutex with the `No_Priority_Inheritance` protocol attribute (see 11.2.5), its priority and scheduling are not affected by its mutex ownership.

When a task is blocking higher priority tasks because of owning one or more mutexes with the `Highest_Blocked_Task` protocol attribute, it inherits the active priorities of all the tasks waiting on any of the mutexes owned by this task and initialized with this protocol.

When a task owns one or more mutexes initialized with the `Highest_Ceiling_Priority` protocol, it inherits the ceiling priorities of all the mutexes owned by this task and initialized with this attribute, regardless of whether other tasks are blocked on any of these mutexes. The Ceiling Priority attribute defines the ceiling priority of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the Ceiling Priority of the mutex shall be set to a priority higher than or equal to the highest priority of all the tasks that may lock that mutex.

If the Mutex Priority Inheritance option is supported: When a task makes a call to `POSIX_Mutexes.Lock`, if the mutex was initialized with the protocol attribute having the value `Highest_Blocked_Task` and the calling task is blocked because the mutex is owned by another task, that owner task shall inherit the priority of the calling task as long as it continues to own the mutex. The implementation shall update its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner task itself becomes blocked on another mutex, the same priority inheritance effect shall be propagated to the other owner task, in a recursive manner.

NOTE: If a task simultaneously owns several mutexes initialized with different protocols, it will inherit all the priorities that it would have obtained by each of these protocols.

Section 14: Clocks and Timers

‘

This section describes the high-resolution clock and timer facilities available under this standard.

The functionality described in this section is optional. If the Timers option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this section to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this section.*

14.1 Package POSIX_Timers

This package provides access to services that can be used to read and set the value of a clock, obtain the resolution of a clock, create a timer, delete a timer, arm and disarm a timer, and obtain the state of a timer.

```
with POSIX,
     POSIX_Signals;
package POSIX_Timers is
   -- 14.1.1 Clock and Timer Types
   type Clock_ID is private;
   type Timer_ID is private;
   -- 14.1.2 Realtime Clock
   Clock_Realtime: constant Clock_ID;
   -- 14.1.3 Timer State and Timer Options
   type Timer_State is private;
   type Timer_Options is new POSIX.Option_Set;
   Absolute_Timer: constant Timer_Options := implementation-defined;
   procedure Set_Initial
      (State:   in out Timer_State;
       Initial: in POSIX.Timespec);
   function Get_Initial
      (State: Timer_State) return POSIX.Timespec;
   procedure Set_Interval
      (State:    in out Timer_State;
       Interval: in POSIX.Timespec);
   function Get_Interval
      (State: Timer_State) return POSIX.Timespec;
   -- 14.1.4 Clock Operations
   procedure Set_Time
      (Clock: in Clock_ID;
       Value: in POSIX.Timespec);
   function Get_Time (Clock: Clock_ID)
     return POSIX.Timespec;
   function Get_Resolution (Clock: Clock_ID)
     return POSIX.Timespec;
   -- 14.1.5 Create a Timer
   function Create_Timer
      (Clock: Clock_ID;
       Event: POSIX_Signals.Signal_Event)
     return Timer_ID;
   -- 14.1.6 Delete a Timer
   procedure Delete_Timer
      (Timer: in out Timer_ID);
```

```
   -- 14.1.7 Timer Operations
   procedure Arm_Timer
      (Timer:     in  Timer_ID;
       Options:   in  Timer_Options;
       New_State: in  Timer_State;
       Old_State: out Timer_State);
   procedure Arm_Timer
      (Timer:     in  Timer_ID;
       Options:   in  Timer_Options;
       New_State: in  Timer_State);
   function Get_Timer_State
      (Timer: Timer_ID) return Timer_State;
   procedure Disarm_Timer
      (Timer: in  Timer_ID);
   function Get_Timer_Overruns
      (Timer: Timer_ID) return Natural;
private
   implementation-defined
end POSIX_Timers;
```

14.1.1 Clock and Timer Types

14.1.1.1 Synopsis

```
type Clock_ID is private;
type Timer_ID is private;
```

14.1.1.2 Description

Clocks are used to measure the passage of time. Timers are based on clocks and can be used to trigger signals.

Objects of type `Clock_ID` are used to identify clocks in clock and timer operations.

Objects of type `Timer_ID` are used to identify timers in timer operations.

A value of type `Timer_ID` is *valid* if it has been returned by function `Create_Timer` (see 14.1.5) and not yet provided as a parameter to `Delete_Timer` (see 14.1.6).

14.1.2 Realtime Clock

14.1.2.1 Synopsis

```
Clock_Realtime: constant Clock_ID;
```

14.1.2.2 Description

Clocks measure the passage of time. A clock can measure real time or some other implementation-defined concept of time, such as the processor time used by the process.

Every conforming implementation shall provide at least one system-wide realtime clock, having the identifier `Clock_Realtime`. The maximum allowable resolution for the `Clock_Realtime` clock and all timers based on this clock is `Portable_Clock_Resolution_Minimum` (see 2.6). Implementations may support smaller values of resolution for the `Clock_Realtime` clock to provide finer granularity time

bases. The actual resolution supported by an implementation for a specific clock is obtained by use of `Get_Resolution` (see 14.1.4). If the actual resolution supported for timers based on this clock differs from the resolution supported by the clock, the implementation shall document this difference.

The minimum allowable maximum value for the `Clock_Realtime` clock and absolute timers based on it is `Seconds'Last` (see 2.4.8). If the maximum value supported for timers based on this clock differs from the maximum value supported by the clock, the implementation shall document this difference.

NOTE: Conforming POSIX.5 Applications cannot presume the system-wide realtime clock is monotonically increasing.

A conforming implementation may declare additional constants of type `Clock_ID` in this package specification, provided the names begin with "`Clock_`/nobreak".

NOTE: A future revision to this standard is expected to define other values of type `Clock_ID` to represent an externally synchronized block, a monotonically increasing clock, and execution time clocks for processes and tasks.

14.1.3 Timer State and Timer Options

14.1.3.1 Synopsis

```
type Timer_State is private;
type Timer_Options is new POSIX.Option_Set;
Absolute_Timer: constant Timer_Options := implementation-defined;
procedure Set_Initial
   (State:   in out Timer_State;
    Initial: in POSIX.Timespec);
function Get_Initial
   (State: Timer_State) return POSIX.Timespec;
procedure Set_Interval
   (State:    in out Timer_State;
    Interval: in POSIX.Timespec);
function Get_Interval
   (State: Timer_State) return POSIX.Timespec;
```

14.1.3.2 Description

The type `Timer_State` shall represent the state of a timer, including an initial timer value and a repetition interval for use by the timer operations. The state of a timer shall include at least the following attributes.

Initial

If the Initial attribute is nonzero, it specifies the time to or the time of the next timer expiration (for relative and absolute timers, respectively).

Interval

If the Interval attribute is positive, it specifies an interval to be used in reloading the timer when it expires; that is, a periodic timer is specified. If the value of the Interval attribute is zero, the timer shall be disarmed after its next expiration; that is, a "one-shot" timer is specified.

Implementations may add extensions to the state of a timer as permitted in 1.3.1.1, point(2). Adding extensions that might change the behavior of the application with respect to this standard when those members are uninitialized also requires that the extension be enabled as required in 1.3.1.1.

Values of the Initial and Interval attributes are of type `Timespec`, which is defined in 2.4.8.

NOTE: Default values for the Initial and Interval timer state attributes are not specified by this standard.

NOTE: The notes in 14.1.7 imply the existence of another hidden attribute of timer state, namely, whether the option `Absolute_Timer` was specified in the `Arm_Timer` call that most recently armed the timer.

Operations are provided to specify and to interrogate the value of the Initial and Interval attributes. The `Set_Initial` procedure shall set the Initial attribute of the timer state object specified by `State` to the value specified by `Initial`, and the `Set_Interval` procedure shall set the Interval attribute of the timer state object specified by `State` to the value specified by `Interval`. The `Get_Initial` function shall return the Initial attribute of the specified timer state object, and the `Get_Interval` function shall return the Interval attribute of the specified timer state object.

Type `Timer_Options` is derived from type `POSIX .Option_Set` and shall include values for each supported timer option. This standard specifies one constant, `Absolute_Timer`, of type `Timer_Options`. The effect of option `Absolute_Timer` is specified in 14.1.7.

14.1.4 Clock Operations

14.1.4.1 Synopsis

```
procedure Set_Time
   (Clock: in Clock_ID;
    Value: in POSIX.Timespec);
function Get_Time (Clock: Clock_ID)
  return POSIX.Timespec;
function Get_Resolution (Clock: Clock_ID)
  return POSIX.Timespec;
```

14.1.4.2 Description

Operations are provided to set and to read time values of clocks and to interrogate clock resolution.

The `Set_Time` procedure shall set the time of the clock specified by `Clock` to `Value`. Time values between two consecutive nonnegative multiples of the resolution of the specified clock shall be truncated down to the smaller multiple of the resolution.

NOTE: The behavior of `Set_Time` for negative time values is not specified by this standard.

The `Get_Time` function shall return the current time value of the clock specified by `Clock`.

A clock may be system wide, that is, visible to all processes, or it may be per-process, measuring time that is meaningful only within an individual process. For the required clock `Clock_Realtime`, the values returned by `Get_Time` and specified by `Set_Time` shall represent an offset from the Epoch. (See 2.2.2.58).

An implementation may also support additional clocks. The interpretation of time values for these clocks is unspecified.

The effect of setting a clock via `Set_Time` on expiration times of armed timers associated with that clock is implementation defined.

NOTE: A future revision to this standard is expected to require that relative timers be unaffected and that absolute timers expire at the new adjusted time of the clock. See the second note in 14.1.7.

The appropriate privilege to set a particular clock is implementation defined.

The `Get_Resolution` function shall return the resolution of the clock specified by `Clock`. Whether it is possible to set the resolution of any clock and the means of doing so are implementation defined.

14.1.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

`Clock` is not a valid clock identifier.

The `Value` parameter for `Set_Time` is outside the range allowed for the specified `Clock`.

Upon `Set_Time`, an argument of type `Timespec` cannot be interpreted as a valid time. For this error, the implementation may instead raise `Constraint_Error`.

NOTE: For example, an invalid value of type `Timespec` may be obtained via an uninitialized variable.

`Operation_Not_Implemented`

The operations `Set_Time`, `Get_Time`, and `Get_Resolution` are not supported by this implementation.

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

Upon `Set_Time`, the requesting process does not have the appropriate privilege to set the specified clock.

14.1.5 Create a Timer

14.1.5.1 Synopsis

```
function Create_Timer
   (Clock: Clock_ID;
    Event: POSIX_Signals.Signal_Event)
  return Timer_ID;
```

14.1.5.2 Description

An application can create a timer, associated with a specific clock, which later can be armed to provide one-shot or periodic notifications of timer expiration. Notifications of timer expiration are provided to the application as signal events.

Each implementation shall define a set of clocks, which shall include the system-wide realtime clock `Clock_Realtime`, that can be used as timing bases for timers.

The `Create_Timer` function shall create a timer using the specified `Clock` as a timing base and shall return the identifier of the timer, which can be used in later timer operations. This timer identifier shall be unique within the calling process until the timer is deleted. The timer whose identification is returned shall be in a disarmed state upon return from `Create_Timer`.

The `Event` parameter identifies a `Signal_Event` object that defines the asynchronous notification that will occur when the timer expires. If the Notification attribute of `Event` is `Signal_Notification`, then `Event` shall specify the signal number and the application-specific data value to be sent upon timer expiration. The `Event` parameter must not specify a reserved signal of the Ada language implementation. If the Notification attribute of `Event` is `No_Notification`, no notification shall be sent upon timer expiration. The behavior for any other value of Notification is implementation defined.

If the `Event` parameter specifies `Signal_Notification`, the Signal specified by `Event` shall be generated upon timer expiration.

If the Realtime Signals option is supported: If the signal is in the range `Realtime_Signal` and if signal queueing is enabled for the specified signal, the signal shall be queued with the Data value specified by `Event`. Otherwise, it is unspecified whether the signal is queued and what Data value, if any, is queued with it.

Timers shall not be inherited by a child process created by a call to `POSIX_Unsafe_Process_Primitives.Fork`, or `POSIX_Process_Primitives.Start_Process`, `POSIX_Process_Primitives.Start_Process_Search`. Timers shall be disarmed and deleted by a call to `POSIX_Unsafe_Process_Primitives.Exec` or `POSIX_Unsafe_Process_Primitives.Exec_Search`.

The value returned is undefined if the `Create_Timer` function fails.

If the Ada implementation reserves any timers, the value returned by `Timers_Maximum` (see 4.5) shall be reduced to reflect the number of timers actually available to the application.

14.1.5.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Resource_Temporarily_Unavailable`

The system lacks sufficient signal queueing resources to honor the request. The calling process has already created all the timers it is allowed by this implementation.

Invalid_Argument

Clock is not a valid clock identifier.

The application is attempting to create a timer that delivers a signal that is reserved for the Ada implementation (see 2.2.2.155).

Operation_Not_Supported

The function Create_Timer is not supported by this implementation.

14.1.6 Delete a Timer

14.1.6.1 Synopsis

```
procedure Delete_Timer
   (Timer: in out Timer_ID);
```

14.1.6.2 Description

A previously created timer can be deleted.

The Delete_Timer procedure shall delete the timer specified by Timer, previously created by Create_Timer. If the specified timer is armed when Delete_Timer is called, the behavior shall be as if the timer is automatically disarmed before deletion. The disposition of pending signals (if any) generated by the deleted timer is unspecified. An implementation may cause Delete_Timer to set Timer to some invalid value. The effect of using Timer in subsequent timer operations, except in a subsequent call to Create_Timer, is undefined.

14.1.6.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

Timer is not a valid timer identifier.

Operation_Not_Supported

The procedure Delete_Timer is not supported by this implementation.

14.1.7 Timer Operations

14.1.7.1 Synopsis

```
procedure Arm_Timer
   (Timer:     in  Timer_ID;
    Options:   in  Timer_Options;
    New_State: in  Timer_State;
    Old_State: out Timer_State);
procedure Arm_Timer
   (Timer:     in  Timer_ID;
    Options:   in  Timer_Options;
    New_State: in  Timer_State);
function Get_Timer_State
   (Timer: Timer_ID) return Timer_State;
procedure Disarm_Timer
   (Timer: in  Timer_ID);
function Get_Timer_Overruns
   (Timer: Timer_ID) return Natural;
```

14.1.7.2 Description

Once created, a timer must be armed in order for signals to be generated upon timer expiration.

The `Arm_Timer` procedure shall set the time until next expiration of the timer specified by `Timer` from the Initial attribute of `New_State` and shall arm the timer if this value is positive.

If `Absolute_Timer` is specified by `Options`, then the timer specified by `Timer` shall be armed as an absolute timer. The time of the next timer expiration shall be set to the value specified by the Initial attribute of `New_State`, provided this value is not zero. The timer shall expire when the clock on which the timer is based reaches the time specified by the Initial attribute of `New_State`.

If the Initial attribute of `New_State` is zero, the implementation shall detect the condition, and the call to `Arm_Timer` shall fail.

If the Initial attribute of `New_State` is not zero, but is less than or equal to the current value of the clock on which the specified timer is based, the call to `Arm_Timer` shall succeed, and immediate expiration notification shall be made.

If `Absolute_Timer` is not specified by `Options`, then the timer specified by `Timer` shall be armed as a relative timer. The time of the next timer expiration shall be set as specified by the Initial attribute of `New_State`, provided the value is positive. The value of the Initial attribute of `New_State` shall be added to the current value of the clock on which the specified timer is based in order to obtain the future value of time at which the timer shall expire.

If the Initial attribute of `New_State` is zero or negative, the implementation shall detect the condition, and the call to `Arm_Timer` shall fail.

If the timer specified by `Timer` is already armed, `Arm_Timer` shall reset the time until next expiration, as specified by `Options` and the Initial attribute of `New_State`. The effect on pending expiration notifications of resetting the timer is unspecified.

The effect of the Interval attribute of `New_State` is to specify a reload value when the timer expires. When this value is zero, a one-shot timer is armed. Upon expiration, the timer shall not be automatically rearmed. When this value is positive, a periodic timer is specified. Upon timer expiration, the value of the Interval attribute of `New_State` shall be added to the current value of the clock on which the timer is based to obtain the next expiration time of the timer, and the timer shall be rearmed.

Time values that are between two consecutive nonnegative integer multiples of the resolution of the underlying clock shall be rounded up to the next larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded-up time value.

If present, `Old_State` shall return the state of the timer prior to setting its new state, the same as would be returned by a call to `Get_Timer_State`.

The function `Get_Timer_State` shall return the current state of the timer specified by `Timer`. The Initial attribute of the return value shall return a `Timespec` value

representing either zero, if the specified timer is not armed, or the difference between the next expiration time of `Timer` and the current time of the underlying clock, if the timer is armed. The Interval attribute of the return value shall return a `Timespec` value representing either zero, if the specified timer is one-shot, or the interval value previously specified by a call to `Arm_Timer`, if the timer is periodic. The values of the Initial and Interval attributes are subject to the resolution of the underlying clock.

NOTE: The value of the Initial attribute returns a relative amount of time until the next timer expiration, even if the timer was armed as an absolute timer.

The `Disarm_Timer` procedure shall cancel a previous timer arming operation for the timer specified by `Timer`. The effect of `Disarm_Timer` on pending expiration notifications is unspecified.

NOTE: A future revision to this standard is expected to require the following behavior:

- *If the value of the realtime clock is set via `Set_Time`, the new value of the clock shall be used to determine the expiration for any absolute timers based upon the realtime clock.*
- *If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time service shall expire immediately in the same way as if the clock had reached the requested time normally.*
- *Setting the value of the realtime clock via `Set_Time` shall have no effect on tasks that are blocked waiting for a relative time service based on this clock. Consequently, such time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.*
- *The effect of setting a clock via `Set_Time` on armed timers associated with a clock other than the realtime clock is implementation defined.*

Only a single signal shall be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no additional signal shall be queued, and a timer overrun shall occur.

When a timer expiration signal is accepted, if the Realtime Signals option is supported, the `Get_Timer_Overruns` function shall return the timer expiration overrun count for the timer specified by `Timer`. The overrun count returned shall be the number of extra timer expirations that occurred between the time the timer expiration signal was generated and when it was accepted, up to an implementation-definedmaximum of `Timer_Overruns_Maximum`. If the number of such extra timer expirations exceeds `Timer_Overruns_Maximum`, then the overrun count shall be set to `Timer_Overruns_Maximum`. The value returned by `Get_Timer_Overruns` shall apply to the most recent expiration signal acceptance for the specified timer. If no expiration signal has been accepted for `Timer`, or the Realtime Signals option is not supported, the meaning of the overrun count returned is undefined.

14.1.7.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

`Timer` is not a valid timer identifier.

Upon Arm_Timer, Absolute_Timer is specified by Options, and the Initial attribute of New_State is zero; or Absolute_Timer is not specified, and the Initial attribute of New_State is zero or negative.

Upon Arm_Timer, an argument of type Timespec cannot be interpreted as a valid time. For this error, the implementation may instead raise Constraint_Error.

NOTE: For example, an invalid value of type Timespec may be obtained via an uninitialized variable.

Operation_Not_Implemented

The operations Arm_Timer, Get_Timer_State, Disarm_Timer, and Get_Timer_Overruns are not supported by this implementation.

14.2 High Resolution Delay

This standard specifies no new operations for high resolution delay. Rather, it places additional requirements on the implementation of the **delay** statement and, if supported, the **delay until** statement.

If the Timers option is supported: The resolution of delays via the **delay** and **delay until** statements shall be no coarser than for timers created for Clock_Realtime.

Section 15: Message Passing

This section describes the facilities for interprocess communication message passing available under this standard.

The functionality described in this section is optional. If the Message Queues option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this section to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this section.*

15.1 Package POSIX_Message_Queues

This package provides access to services that can be used to create and remove message queue objects, send messages to a message queue, and receive messages from a message queue.

```
with Ada_Streams,
     POSIX,
     POSIX_IO,
     POSIX_Permissions,
     POSIX_Signals;
package POSIX_Message_Queues is
   -- 15.1.1 Message Queue Attributes
   type Message_Queue_Descriptor is private;
   type Attributes is private;
   type Message_Queue_Options is new POSIX.Option_Set;
   Non_Blocking: constant Message_Queue_Options
      := Message_Queue_Options(POSIX_IO.Non_Blocking);
   subtype Message_Priority is Integer range implementation-defined;
   procedure Set_Max_Messages
      (Attrs: in out Attributes;
       Value: Natural);
   function Get_Max_Messages
      (Attrs: Attributes) return Natural;
   procedure Set_Message_Length
      (Attrs: in out Attributes;
       Value: Natural);
   function Get_Message_Length
      (Attrs: Attributes) return Natural;
   procedure Set_Options
      (Attrs: in out Attributes;
       Value: Message_Queue_Options);
   function Get_Options
      (Attrs: Attributes)
     return Message_Queue_Options;
   function Get_Message_Count
      (Attrs: Attributes) return Natural;
   -- 15.1.2 Open a Message Queue
   function Open
      (Name:           POSIX.POSIX_String;
       Mode:           POSIX_IO.File_Mode;
       Options:        POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
     return Message_Queue_Descriptor;
```

```
function Open_Or_Create
   (Name:           POSIX.POSIX_String;
    Mode:           POSIX_IO.File_Mode;
    Permissions:    POSIX_Permissions.Permission_Set;
    Options:        POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
  return Message_Queue_Descriptor;
function Open_Or_Create
   (Name:           POSIX.POSIX_String;
    Mode:           POSIX_IO.File_Mode;
    Permissions:    POSIX_Permissions.Permission_Set;
    Options:        POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
    Attrs:          Attributes;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
  return Message_Queue_Descriptor;
-- 15.1.3 Close a Message Queue
procedure Close (MQ: in out Message_Queue_Descriptor);
-- 15.1.4 Remove a Message Queue
procedure Unlink_Message_Queue (Name:  in POSIX.POSIX_String);
-- 15.1.5 Send a Message to a Message Queue
procedure Send
   (MQ:             in Message_Queue_Descriptor;
    Message:        in Ada_Streams.Stream_Element_Array;
    Priority:       in Message_Priority;
    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
-- 15.1.6 Receive a Message from a Message Queue
procedure Receive
   (MQ:             in  Message_Queue_Descriptor;
    Message:        out Ada_Streams.Stream_Element_Array;
    Last:           out Ada_Streams.Stream_Element_Offset;
    Priority:       out Message_Priority;
    Masked_Signals: in  POSIX.Signal_Masking:= POSIX.RTS_Signals);
-- 15.1.7 Generic Message Passing
generic
  type Message_Type is private;
package Generic_Message_Queues is
   procedure Send
      (MQ:             in Message_Queue_Descriptor;
       Message:        in Message_Type;
       Priority:       in Message_Priority;
       Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
   procedure Receive
      (MQ:             in  Message_Queue_Descriptor;
       Message:        out Message_Type;
       Priority:       out Message_Priority;
       Masked_Signals: in  POSIX.Signal_Masking:= POSIX.RTS_Signals);
   function Get_Error_Buffer return Ada_Streams.Stream_Element_Array;
end Generic_Message_Queues;
-- 15.1.8 Notify Process that a Message is Available
procedure Request_Notify
   (MQ:    in Message_Queue_Descriptor;
    Event: in POSIX_Signals.Signal_Event);
procedure Remove_Notify (MQ: in Message_Queue_Descriptor);
-- 15.1.9 Set/Query Message Queue Attributes
procedure Set_Attributes
   (MQ:        in  Message_Queue_Descriptor;
    New_Attrs: in  Attributes;
    Old_Attrs: out Attributes);
```

```
   procedure Set_Attributes
      (MQ:        in  Message_Queue_Descriptor;
       New_Attrs: in  Attributes);
   function Get_Attributes
      (MQ:        Message_Queue_Descriptor)
       return Attributes;
private
   implementation-defined
end POSIX_Message_Queues;
```

15.1.1 Message Queue Attributes

15.1.1.1 Synopsis

```
   type Message_Queue_Descriptor is private;
   type Attributes is private;
   type Message_Queue_Options is new POSIX.Option_Set;
   Non_Blocking: constant Message_Queue_Options
      := Message_Queue_Options(POSIX_IO.Non_Blocking);
   subtype Message_Priority is Integer range implementation-defined;
   procedure Set_Max_Messages
      (Attrs: in out Attributes;
       Value: Natural);
   function Get_Max_Messages
      (Attrs: Attributes) return Natural;
   procedure Set_Message_Length
      (Attrs: in out Attributes;
       Value: Natural);
   function Get_Message_Length
      (Attrs: Attributes) return Natural;
   procedure Set_Options
      (Attrs: in out Attributes;
       Value: Message_Queue_Options);
   function Get_Options
      (Attrs: Attributes)
     return Message_Queue_Options;
   function Get_Message_Count
      (Attrs: Attributes) return Natural;
```

15.1.1.2 Description

Objects of type `Message_Queue_Descriptor` are used to refer to open message queue descriptions. A value of type `Message_Queue_Descriptor` is *valid* if it was returned by a call to the functions `Open` or `Open_Or_Create` (See 15.1.2) and not yet used as a parameter in a call to `Close` (See 15.1.3).

Message queues may, but need not, be implemented within the file system, and open message queues may use file descriptors. Thus, the use of message queues may reduce the number of file descriptors available for other uses (see Open Files Maximum in 2.6.1).

Objects of type `Attributes` are used to specify attributes of message queues. Message queue attributes are initially set when a message queue is created or opened. Each value of type `Attributes` has at least the following attributes :

Max Messages

The number of messages that can be held in the message queue, without causing `Send` or `Generic_Message_Queues.Send` to fail or wait due to lack of resources. This attribute may be queried, but can only be set at the time the message queue is created.

Message Length

The maximum size in bytes (not bits) of each message in the message queue. This attribute may be queried, but can only be set at the time the message queue is created.

Options

Options specific to message queues that affect message queue operations and the state of the message queue to which they are applied. This attribute may be both queried and set (See 15.1.9).

Message Count

The number of messages currently on the queue. This attribute may be queried, but cannot be set.

Implementations may add other attributes, as permitted in 1.3.1.1 (2). Adding attributes that may change the behavior of applications with respect to this standard when those attributes are uninitialized also requires that the extension be enabled, as required by 1.3.1.1.

A function whose name is of the form `Get_X`, where `X` is the name of one of the attributes of a message queue, shall return the value of that attribute for the specified `Attributes` object.

A procedure whose name is of the form `Set_X`, where `X` is the name of one of the attributes of a message queue, shall set the value of that attribute for the specified `Attributes` object.

NOTE: The operation to set the Message Count attribute of an `Attributes` object is intentionally omitted.

Values of type `Message_Queue_Options` are used to specify the `Options` attribute of an `Attributes` object. The following value of `Message_Queue_Options` shall be defined :

`Non_Blocking`

Calls to `Send`, `Generic_Message_Queues.Send`, `Receive`, and `Generic_Message_Queues.Receive` shall fail if a message queue resource is temporarily unavailable (such as if the message queue is full on a `Send`). If `Non_Blocking` is not specified, calls to these operations shall wait until the resource becomes available.

The constant `POSIX_Message_Queues.Non_Blocking` is a homograph of the constant `POSIX_IO.Non_Blocking` and shall have the same representation, although it is of a different type.

The interpretation of other values of `Message_Queue_Options` is unspecified.

The subtype `Message_Priority` defines the range of message priority levels supported by the implementation.

NOTE: `Message_Priority'Last` may be greater than Message Priority Maximum.

NOTE: Priority is an attribute of a message, not an attribute of a message queue.

15.1.1.3 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Operation_Not_Implemented`

The requested message queue operation is not supported by this implementation.

15.1.2 Open a Message Queue

15.1.2.1 Synopsis

```
function Open
   (Name:           POSIX.POSIX_String;
    Mode:           POSIX_IO.File_Mode;
    Options:        POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
  return Message_Queue_Descriptor;
function Open_Or_Create
   (Name:           POSIX.POSIX_String;
    Mode:           POSIX_IO.File_Mode;
    Permissions:    POSIX_Permissions.Permission_Set;
    Options:        POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
  return Message_Queue_Descriptor;
function Open_Or_Create
   (Name:           POSIX.POSIX_String;
    Mode:           POSIX_IO.File_Mode;
    Permissions:    POSIX_Permissions.Permission_Set;
    Options:        POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
    Attrs:          Attributes;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
  return Message_Queue_Descriptor;
```

15.1.2.2 Description

The `Open` and `Open_Or_Create` functions establish the connection between a process and a message queue via a message queue descriptor. They create an open message queue description that refers to that message queue and return a message queue descriptor that refers to the open message queue description. The open message queue description contains information about the mode in which the process accesses the message queue as well as a reference to the message queue itself. The message queue descriptor is used by other operations to refer to the open message queue description.

The `Open` function requires that the message queue exist at the time of the call. `Open_Or_Create` will create the message queue if it does not exist. The `Name` parameter specifies a string naming a message queue. It is unspecified whether this name appears in the file system and is visible to other functions that take pathnames as arguments. The `Name` parameter shall conform to the construction rules

for a pathname. If Name begins with the slash character, then processes calling Open or Open_Or_Create with the same value of Name shall refer to the same message queue object, as long as that name has not been removed. If Name does not begin with the slash character, the effect is implementation defined. The interpretation of slash characters other than the leading slash character in Name is implementation defined.

The Mode parameter specifies the desired receive and/or send access to the message queue. The requested access permission to receive or send messages is granted if the calling process would be granted read or write access, respectively, to an equivalently protected file. Allowed values for Mode are defined by POSIX_IO.File_Mode with effects as follows:

Read_Only

Open the message queue for receiving messages. The process can use the returned message queue descriptor as a parameter to Receive or Generic_Message_Queues.Receive but not Send or Generic_Message_Queues.Send.

Write_Only

Open the queue for sending messages. The process can use the returned message queue descriptor as a parameter to Send and Generic_Message_Queues.Send but not Receive or Generic_Message_Queues.Receive.

Read_Write

Open the queue for both receiving and sending messages. The process can use the returned message queue descriptor as a parameter to any of the operations allowed for Read_Only and Write_Only.

A message queue may be referenced by multiple open message queue descriptions, with (possibly) different modes, in the same or different processes.

The Open_Or_Create function can be used to create a message queue. It requires an additional parameter, Permissions, which specifies the permissions for the message queue, if it is created. The value of Permissions may be constructed as the union ("+") of values from POSIX_Permissions.Permission_Set. The permissions behave as described in 5.1.1 for file permissions.

If the pathname Name is not the name of an existing message queue on a call to Open, then the call shall fail. If the pathname, Name, has already been used to create a message queue that still exists, the effect of the Open_Or_Create call shall be the same as Open, except as noted under Exclusive below. Otherwise, a message queue is created as follows:

— The new message queue shall have no messages in it.

— The user ID of the message queue shall be set to the effective user ID of the process, and the group ID of the message queue shall be set to the effective group ID of the process.

— If the optional parameter Attrs is specified and the calling process has the appropriate privilege for Name, the Max Messages and Message Length attributes of the message queue shall be set to the values specified in Attrs.

NOTE: The blocking behavior associated with the new open message queue description is taken from the `Options` *parameter. The Options attribute of the* `Attrs` *parameter is ignored by* `Open_Or_Create`*.*

— If the optional parameter `Attrs` is specified and the calling process does not have the appropriate privilege on `Name`, the `Open_Or_Create` function shall fail without creating the message queue.

— If the `Attrs` parameter is not specified, the message queue shall be created with implementation-defined default message queue attributes.

The parameter `Options` shall specify additional options for the message queue. The meanings of the options are as follows :

`Exclusive`

`Open_Or_Create` shall fail if the message queue `Name` exists. The check for the existence of the message queue and the creation of the message queue if it does not exist shall be atomic with respect to other processes executing `Open_Or_Create` naming the same `Name` with `Exclusive` set. If this option is specified for `Open`, the result is undefined.

`Non_Blocking`

This option is associated with the open message queue description and determines whether calls to `Send`, `Generic_Message_Queues.Send`, `Receive`, and `Generic_Message_Queues.Receive` shall fail if a message queue resource is temporarily unavailable (such as if the message queue is full on a `Send`). If `Non_Blocking` is not specified in `Options`, calls to these operations shall wait until the resource becomes available.

The constant `POSIX_IO.Non_Blocking` is a homograph of the constant `POSIX_Message_Queues.Non_Blocking` and shall have the same representation, although it is of a different type.

The effect of setting other values from `POSIX_IO.Open_Option_Set` in the `Options` parameter of `Open` or `Open_Or_Create` is undefined.

`Open` and `Open_Or_Create` may be interruptible by the delivery of a signal. `Masked_Signals` specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

The `Open` and `Open_Or_Create` functions shall not add or remove messages from the message queue.

15.1.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

The message queue exists, and the permissions specified by `Mode` are denied; or the message queue does not exist, and permission to create the message queue is denied.

`File_Exists`

The `Exclusive` option is specified on a call to `Open_Or_Create` and the named message queue already exists.

`Interrupted_Operation`

A signal interrupted the `Open` or `Open_Or_Create` operation.

`Invalid_Argument`

The `Open` or `Open_Or_Create` operation is not supported for the given name. The implementation shall document under what circumstances this error may be returned.

The Max Messages or Message Length attribute of `Attrs` is less than or equal to zero.

`Too_Many_Open_Files`

Too many message queue descriptors or file descriptors are currently in use by this process.

`Filename_Too_Long`

The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

`Too_Many_Open_Files_In_System`

Too many message queues are currently open in the system.

`No_Such_File_Or_Directory`

The `Open` operation was called, and the named message queue does not exist.

`No_Space_Left_On_Device`

There is insufficient space for the creation of the new message queue.

`Operation_Not_Implemented`

The `Open` and `Open_Or_Create` operations are not supported by this implementation.

15.1.3 Close a Message Queue

15.1.3.1 Synopsis

```
procedure Close (MQ: in out Message_Queue_Descriptor);
```

15.1.3.2 Description

The `Close` procedure shall remove the association between the message queue descriptor `MQ` and its associated open message queue description. When all message queue descriptors associated with an open message queue description are closed, the open message queue description shall be freed. The results of using the message queue descriptor value after a successful return from `Close` until the return of this message queue descriptor value from a subsequent call to `Open` or `Open_Or_Create` are undefined.

If the process has successfully attached a notification request to the message queue, this attachment shall be removed and the message queue made available for another process to attach for notification.

15.1.3.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor
: The MQ parameter is not a valid message queue descriptor.

Operation_Not_Implemented
: The Close operation is not supported by this implementation.

15.1.4 Remove a Message Queue

15.1.4.1 Synopsis

```
procedure Unlink_Message_Queue (Name:  in POSIX.POSIX_String);
```

15.1.4.2 Description

The Unlink_Message_Queue procedure shall remove the message queue named by Name. After a successful call to Unlink_Message_Queue with Name, a call to Open with Name shall fail. If one or more processes have the message queue open when Unlink_Message_Queue is called, destruction of the message queue shall be postponed until all references to the message queue have been closed. Calls to Open_Or_Create to recreate the message queue may fail until the message queue is actually removed. However, the Unlink_Message_Queue call need not block until all references have been closed; it may return immediately.

15.1.4.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Permission_Denied
: Permission is denied to unlink the named message queue.

Filename_Too_Long
: The length in POSIX characters of the specified pathname exceeds Pathname Limit; or the length in POSIX characters of a component of the specified pathname is greater than Filename Maximum, and the Filename Truncation option is not supported for the pathname prefix of that component. (See 5.4.2.)

No_Such_File_Or_Directory
: The named message queue does not exist.

Operation_Not_Implemented
: The Unlink_Message_Queue operation is not supported by this implementation.

15.1.5 Send a Message to a Message Queue

15.1.5.1 Synopsis

```
procedure Send
   (MQ:             in Message_Queue_Descriptor;
    Message:        in Ada_Streams.Stream_Element_Array;
    Priority:       in Message_Priority;
    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
```

15.1.5.2 Description

The Send procedure adds the message of type Ada_Streams.Stream_Element_-Array to the message queue specified by MQ. The call shall fail if Message'Length is greater than the Message Length attribute of the message queue.

If the message queue specified by MQ is not full, Send shall behave as if the message is inserted into the message queue at the position indicated by the Priority argument. The value of Priority shall be less than Message Priority Maximum.

A message with a larger numeric value for Priority is inserted before messages with lower values for Priority. A message shall be inserted after other messages in the queue with equal Priority, if any.

If the message queue is full, and Non_Blocking is not specified in the open message queue description, Send shall block until space becomes available to enqueue the message or until interrupted by a signal. If more than one task is waiting to send when space becomes available in the message queue, one task shall be unblocked to send its message.

If the Priority Process Scheduling option is supported: The task to be unblocked shall be selected in a manner appropriate to the scheduling policies and parameters in effect for the blocked tasks.

If the Priority Process Scheduling option is not supported, the selection of which task to unblock is unspecified.

If the specified message queue is full and Non_Blocking is specified in the open message queue description, the message is not queued and POSIX.POSIX_Error is raised.

Send may be interruptible by the delivery of a signal. Masked_Signals specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

15.1.5.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Temporarily_Unavailable
: Non_Blocking is specified in the open message queue description associated with MQ, and the specified message queue is full.

Bad_File_Descriptor
: The MQ argument is not a valid message queue descriptor open for writing.

Interrupted_Operation
: A signal interrupted the call to Send.

Invalid_Argument
: The value of Priority is outside of the valid range.

Message_Too_Long
: Message'Length is greater than the Message Length attribute of the queue.

Operation_Not_Implemented

The Send operation is not supported by this implementation.

15.1.6 Receive a Message from a Message Queue

15.1.6.1 Synopsis

```
procedure Receive
   (MQ:             in  Message_Queue_Descriptor;
    Message:        out Ada_Streams.Stream_Element_Array;
    Last:           out Ada_Streams.Stream_Element_Offset;
    Priority:       out Message_Priority;
    Masked_Signals: in  POSIX.Signal_Masking:= POSIX.RTS_Signals);
```

15.1.6.2 Description

The Receive procedure removes the oldest of the highest priority messages from the message queue specified by MQ. The message is stored in the Message parameter of type Ada_Streams.Stream_Element_Array. The call shall fail if Message'Length is less than the Message Length attribute of the message queue. Otherwise, the message shall be removed from the queue and placed into Message, starting at the first element of Message, and Last shall be set to the index of the last byte of the message.

Priority shall be assigned the priority of the message removed from the message queue.

If the specified message queue is empty and Non_Blocking is not specified in the message queue description associated with MQ, Receive shall block until a message is enqueued on the message queue or until interrupted by a signal.

If the Process Scheduling option is supported: The task to be unblocked shall be selected in a manner appropriate to the scheduling policies and parameters in effect for the blocked tasks.

If the Priority Process Scheduling option is not supported, the selection of which task receives the message in unspecified.

If the specified message queue is empty and Non_Blocking is specified in the open message queue description associated with MQ, Receive shall fail.

Receive may be interruptible by the delivery of a signal. Masked_Signals specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

15.1.6.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Temporarily_Unavailable

Non_Blocking is specified in the open message queue description associated with MQ, and the specified message queue is empty.

Bad_File_Descriptor

The MQ argument is not a valid message queue descriptor open for reading.

Message_Too_Long

Message'Length is less than the Message Length attribute of the message queue.

Interrupted_Operation

A signal interrupted the call to Receive.

Operation_Not_Implemented

The Receive operation is not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_Message

The implementation has detected a data corruption problem with the message.

15.1.7 Generic Message Passing

15.1.7.1 Synopsis

```
generic
  type Message_Type is private;
package Generic_Message_Queues is
   procedure Send
      (MQ:             in Message_Queue_Descriptor;
       Message:        in Message_Type;
       Priority:       in Message_Priority;
       Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
   procedure Receive
      (MQ:             in  Message_Queue_Descriptor;
       Message:        out Message_Type;
       Priority:       out Message_Priority;
       Masked_Signals: in  POSIX.Signal_Masking:= POSIX.RTS_Signals);
   function Get_Error_Buffer return Ada_Streams.Stream_Element_Array;
end Generic_Message_Queues;
```

15.1.7.2 Description

The Generic_Message_Queues package provides a typed interface for sending and receiving messages through POSIX message queues. The package is instantiated with any constrained type and is used to send and receive messages of that type.

The Send and Receive operations shall behave the same as defined for POSIX_Message_Queues.Send and POSIX_Message_Queues.Receive except that the type of the Message parameter is the user specified type that was used to instantiate the Generic_Message_Queues package. Constraint_Error may be raised by a call to Receive if the message received cannot be interpreted as a value of the specified type.

Send and Receive may be interruptible by the delivery of a signal. Masked_Signals specifies the set of signals that shall be added to the signal mask for the duration of this operation, as described in 3.3.6.

The Get_Error_Buffer function shall return the raw message data after a Constraint_Error has been raised during a Receive operation. This behavior is specified only for the period after the Constraint_Error is raised and prior to requesting additional operations on the same instantiation of Generic_Message_Queues. The effect of making concurrent calls to Get_Error_Buffer for the same instantiation of Generic_Message_Queues is undefined.

NOTE: An application that uses the operations Send and Receive from an instantiation of Generic_Message_Queues is responsible for ensuring that

— *For Send, the Message Length attribute of the message queue specified by MQ is equal to or greater than the size in bytes of Message.*

— *For Receive, the size in bytes of Message is equal to or greater than the Message Queue Length attribute of the message queue specified by MQ.*

15.1.7.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Resource_Temporarily_Unavailable

Non_Blocking is specified in the open message queue description and one of the following conditions occurred:

— Send was invoked, and the message queue is full.

— Receive was invoked, and the message queue is empty.

— Nno raw data are available on a call to Get_Error_Buffer. Either Constraint_Error has not been raised from a call to Receive or a subsequent operation involving the instantiation of Generic_Message_Queues that raised Constraint_Error has invalidated the raw data.

Bad_File_Descriptor

The MQ argument is not a valid message queue descriptor open for writing on a call to Send or open for reading on a call to Receive.

Interrupted_Operation

A signal interrupted the call to an instantiation of Send or Receive.

Invalid_Argument

The value of Priority specified for the Send operation is outside of the valid range.

Message_Too_Long

The size of Message in bytes is greater than the Message Length attribute of the queue on a Send or less than the Message Length attribute of the queue on a Receive.

Operation_Not_Implemented

The Send, Receive and Get_Error_Buffer operations are not supported by this implementation.

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_Message

The implementation has detected a data corruption problem with the message on a Receive.

In addition to the error conditions specified above, Constraint_Error may be raised by a call to Receive if the message received cannot be interpreted as a value of the specified type. Whenever a message has been removed from the queue, but was not returned due to Constraint_Error having been raised, Get_Error_Buffer can be called to gain access to the raw message data for debugging and error logging.

15.1.8 Notify Process that a Message is Available

15.1.8.1 Synopsis

```
procedure Request_Notify
   (MQ:    in Message_Queue_Descriptor;
    Event: in POSIX_Signals.Signal_Event);
procedure Remove_Notify (MQ: in Message_Queue_Descriptor);
```

15.1.8.2 Description

The functionality described in this subclause is optional. If the Realtime Signals option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

Request_Notify registers the calling process to be notified of message arrival at an empty message queue specified by MQ. The notification specified by Event shall be sent to the process when the message queue transitions to nonempty. At any time, only one process may be registered to receive notification for the specified message queue. If the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue shall fail.

The Event parameter identifies a Signal_Event object that defines the asynchronous notification that will occur when the message queue transitions to nonempty. If the Notification attribute of Event is No_Notification, no notification shall be sent when the message queue transitions to nonempty. If the Notification attribute of Event is Signal_Notification, then Event shall specify the signal number and the application-specific data value to be sent when the message queue transitions to nonempty. If the signal is in the range POSIX_Signals.Realtime_Signal and signal queueing is enabled for the specified signal, the signal shall be queued along with the Data value specified by Event. Otherwise, it is unspecified whether the signal is queued and what Data value, if any, is queued with it. The Event parameter shall not specify a signal reserved for the Ada language implementation. (See 2.2.2.155.)

Remove_Notify shall have the effect of removing an existing registered notification associated with the message queue. This removal shall make the message queue available for registration.

When notification is sent to the registered process, its notification shall be removed. The message queue shall then be available for registration.

If a process has registered for notification of message arrival at a message queue and a process is blocked in Receive waiting to receive a message when a message arrives at the queue, the arriving message shall satisfy the appropriate pending Receive (See 15.1.6). The resulting behavior is as if the message queue stays empty, and no notification shall be sent.

15.1.8.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor
: The MQ argument is not a valid message queue descriptor.

Invalid_Argument
: The notification signal specified by Event is one of the reserved signals.

Resource_Busy
: A process is already registered for notification by the message queue.

Operation_Not_Implemented
: The Request_Notify and Remove_Notify operations are not supported by this implementation.

15.1.9 Set/Query Message Queue Attributes

15.1.9.1 Synopsis

```
procedure Set_Attributes
   (MQ:        in  Message_Queue_Descriptor;
    New_Attrs: in  Attributes;
    Old_Attrs: out Attributes);
procedure Set_Attributes
   (MQ:        in  Message_Queue_Descriptor;
    New_Attrs: in  Attributes);
function Get_Attributes
   (MQ:        Message_Queue_Descriptor)
    return Attributes;
```

15.1.9.2 Description

Set_Attributes is used to set the attributes associated with the open message queue description referenced by the message queue descriptor specified by the MQ parameter. Set_Attributes is overloaded to allow the previous attributes of the message queue to be returned.

Only the Options attribute of the New_Attrs parameter is relevant to the Set_Attributes call. The values of the other attributes are ignored by Set_Attributes.

Get_Attributes is used to get status information and attributes of the message queue and the open message queue description associated with the message queue descriptor specified by MQ. The Max Messages and Message Length attributes of the returned Attributes object shall have the values that were set when the message queue was created. The Options attribute of the returned Attributes object shall

have the value associated with the open message queue description that was set when the message queue was created or opened and as modified by subsequent calls to `Set_Attributes`. The Message Count attribute of the returned `Attributes` object shall have a value that reflects the current state of the specified message queue.

15.1.9.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`

The `MQ` argument is not a valid message queue descriptor.

`Operation_Not_Implemented`

The requested message queue operation is not supported by this implementation.

Section 16: Task Management

This section describes the services that this standard provides for task identification.

16.1 Package Ada_Task_Identification

```
package Ada_Task_Identification renames Ada.Task_Identification;
```

During transition from Ada 83 to Ada 95 the implementation is permitted to replace this renaming declaration with the following package specification.

```
package Ada_Task_Identification is
   -- Task Identification Type
   type Task_ID is private;
   Null_Task_ID: constant Task_ID;
   function  Equals       (Left, Right: Task_ID) return Boolean;
   function  Image        (T: Task_ID) return String;
   -- Identifying the Current Task
   function  Current_Task return Task_ID;
   -- Aborting a Task
   procedure Abort_Task   (T: in out Task_ID);
   -- Inquiring the State of a Task
   function  Is_Terminated (T: Task_ID) return Boolean;
   function  Is_Callable  (T: Task_ID) return Boolean;
private
   -- implementation-defined;
end Ada_Task_Identification;
```

16.1.1 Description

The semantics of the types and operations defined in this package shall satisfy the requirements of Ada 95.

Section 17: Detailed Network Interface - XTI

The operations in this section provide the XTI protocol-independent process- to-process communications interface. If the implementation supports the XTI Detailed Network Interface option, then the operations described in this section shall perform as specified.

17.1 Introduction

17.1.1 Communications Providers

An implementation can provide one or more communications providers at the same time. The identifier parameter of the communications provider (or pathname) passed to the `Open` procedure determines the required communications provider.

NOTE: To keep the applications portable, the identifier parameter (or pathname) of the communications provider should not be hard-coded into the application source code.

To manage multiple communications providers, an application shall call `Open` for each provider.

NOTE: For example, a server application that is waiting for incoming connection indications from several communications providers shall open a communications endpoint for each provider and listen for connection indications on each of the associated file descriptors.

17.1.2 Communications Endpoints

All requests to a communications provider pass through a communications endpoint. One or more communications endpoints are provided by a communication provider. When an application calls `Open`, a file descriptor that identifies a communications endpoint is returned. This file descriptor is used as a parameter to identify the endpoint in subsequent operations.

The `Open` (see 17.4.19) procedure shall succeed only if the implementation supports pathnames that identify character special files for use with XTI calls, and the opened character string is such a pathname. The only operations that a Conforming POSIX.5 Application shall perform on any file descriptor returned by `Open` are those defined in this section and `Duplicate`, `Duplicate_And_Close`, `Close`, `Get_File_Status`, `Poll`, and `Select_File`. Other operations have undefined results.

To be active, a communications endpoint shall have a communications address associated with it by the `Bind` procedure.

A communications endpoint can support only one established connection at a time. A connection is characterized by the association of two active endpoints, made by using the connection establishment operations (`Connect`, `Confirm_Connection`, `Listen`, and `Accept_Connection`). A communications endpoint can be associated with another endpoint in a connection or can be used in connectionless mode.

17.1.3 Association of a Process to an Endpoint

One process can simultaneously open several file descriptors. However, when `POSIX_IO.Non_Blocking` is not set, the process needs to manage the different

actions of the associated connections sequentially. Conversely, several processes can share the same file descriptor (by POSIX_Process_Primitives.Start_Process operations) but they have to synchronize themselves so as not to invoke an operation that is unsuitable to the current state of the communications endpoint.

The communications provider shall treat all applications of a communications endpoint as a single application. If multiple processes are using the same endpoint, they need to coordinate their activities so they do not violate the state of the endpoint. The Synchronize_Endpoint function returns the current state of the endpoint to the application, thereby enabling the application to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the state of the endpoint after a Synchronize_Endpoint is issued.

A process can listen for an incoming connection indication on one file descriptor and accept the connection on a different file descriptor that has been bound with the Request_Queue_Length parameter set to zero or omitted (see 17.4.5).

NOTE: The communication model described here facilitates the writing of a listener application where the listener waits for all incoming connection indications on a given communications endpoint. The listener accepts the connection on a new file descriptor and can (using POSIX_Process_Primitives.Start_Process) create a child process to service the request without blocking other incoming connection indications.

17.1.4 Use of the Same Protocol Address

If several endpoints are bound to the same protocol address, only one at a time shall be listening for incoming connections. However, others can be in data transfer state or establishing a communications connection as initiators.

17.1.5 Addressing

All network addresses are designated using an values of type XTI_Address_Pointer, described in 17.4.1.3. The protocol-specific versions of network addresses, along with functions to convert access values designating objects of protocol-specific address types to and from the XTI_Address_Pointer type, are described in D.2.

17.1.6 Modes of Service

The communications service interface supports two modes of service: connection mode and connectionless mode. A single communications endpoint shall not support both modes of service simultaneously.

17.1.7 Error Handling

Each XTI operation has one or more error returns. Failures shall be indicated by raising the exception POSIX_Error. The XTI error code shall be returned by the POSIX.Get_Error_Code function. The diagnostic function, POSIX.Image, shall return error message text information for the current XTI error. The state of the endpoint can change if an error occurs.

When an operating system service routine fails or a general error occurs, the system error shall also be indicated by raising the exception `POSIX_Error`. The specific system error can be accessed through the `Get_Error_Code` function.

A system error can be generated by the communications provider when a protocol error has occurred. If the error is severe, it can cause the file descriptor and communications endpoint to be unusable. To continue in this case, all applications of the file descriptor need to close the endpoint. Then the communications endpoint can be reopened and initialized.

17.1.8 Execution Modes

The communications service interface is inherently asynchronous; various events may occur that are independent of the actions of an application. For example, an application may be sending data over a connection when a disconnection indication arrives. The application needs to somehow be informed that the connection has been broken.

The communications service interface supports two execution modes for handling events.

(1) When `POSIX_IO.Non_Blocking` is not set, the following operations shall block until the processing associated with them has been completed or they are interrupted by an event: `Connect`, `Listen`, `Receive`, `Confirm_Connection`, `Receive_Data_Unit`, `Receive_And_Scatter_Data`, `Receive_And_Scatter_Data_Unit`, `Send`, `Send_Data_Unit`, `Gather_And_Send_Data`, and `Gather_And_Send_Data_Unit`.

NOTE: This mode of execution is the default. It is useful for applications that wait for events to occur, or for application processes that maintain only a single connection. For example, `Receive` is required to wait until data have arrived before returning control.

(2) When `POSIX_IO.Non_Blocking` is set, the above operations shall not block, but shall initiate the processing associated with them and then return control.

NOTE: In general, the results of such processing can be obtained by a later call to another operation. For example, the result of a call to `Connect` with `POSIX_IO.Non_Blocking` set can be obtained by a call to `Confirm_Connection`. The handling of networking events in this manner is seen as a desirable capability of the communications interface. It enables applications to perform useful work while expecting a particular event. For example, an operation that attempts to receive data returns control immediately if `POSIX_IO.Non_Blocking` is set and no data are available. The application can then periodically poll for incoming data until it arrives. Polling is useful for applications that expect long delays between events and have other tasks that they can perform in the meantime and for applications that handle multiple connections concurrently.

The two execution modes are not provided through separate interfaces or different operations. The desired mode is specified through the `POSIX_IO.Non_Blocking` flag, which may be set when the communications provider is initially opened, or before any specific operation or group of operations is executed using the `POSIX_IO.Set_File_Control` procedure.

17.1.8.1 Service Modes

Nine (only eight if orderly release is not supported) events are defined in the communications service interface to cover both connection mode and connectionless mode

service. They are represented as a set of symbols of type `POSIX.Option_Set`, using the following names:

`Connect_Request_Received`

In connection mode only, a connection request from a remote application was received by a communications provider. For a connection request to be received both of the following conditions shall be met:

(1) The file descriptor is bound to a valid address.

(2) No connection is established at this time.

`Connect_Response_Received`

In connection mode only, a connection response was received by the communications provider. Receipt of a connection response can occur after a `Connect` has been issued.

`Normal_Data_Received`

Normal data (all or part of a SDU) were received by the communications provider.

`Expedited_Data_Received`

Expedited data were received by the communications provider.

`Disconnect_Request_Received`

In connection mode only, a disconnection request was received by the communications provider.

`Orderly_Release_Request_Received`

An orderly release request was received by a communications provider. Receipt of a orderly release request can only occur in connection mode with orderly release.

`Error_In_Previously_Sent_Datagram`

In connectionless mode only, an error was found in a previously sent data unit (datagram).

`Okay_To_Send_Normal_Data`

Flow control restrictions on normal data flow that led to a `Flow_Control_Error` error have been lifted. Normal data can be sent again.

`Okay_To_Send_Expedited_Data`

Flow control restrictions on expedited data flow that led to a `Flow_Control_Error` error have been lifted. Expedited data may be sent again.

A process that issues operations with `POSIX_IO.Non_Blocking` clear needs to still be able to recognize certain events and act on them if necessary. Recognition of events that require attention is handled through a special communications error `Event_Requires_Attention`, which is returned by an operation when an event is outstanding. The `Look` function is then invoked to identify the specific event that has occurred when this error is returned.

Another means to notify a process that an event has occurred is polling. The polling capability enables processes to do useful work and periodically poll for one of the events defined in this subclause (see the list above, and also Table 17.1). This facility is provided by setting `POSIX_IO.Non_Blocking` for the appropriate primitive(s).

17.1.8.2 Events and Look

All events that occur at a communications endpoint are stored by XTI. These events are retrievable one at a time via the `Look` function. If multiple events occur, it is implementation-defined in what order `Look` shall return the events. An event is outstanding on a communications endpoint until it is consumed. Every event has a corresponding consuming operation that handles the event and clears it. Both `Normal_Data_Received` and `Expedited_Data_Received` events are consumed when the corresponding consuming operation has read all the corresponding data associated with that event. As a consequence, `Normal_Data_Received` always indicates that there are data to receive. Two events, `Okay_To_Send_Normal_Data` and `Okay_To_Send_Expedited_Data`, are also cleared as they are returned by `Look`. Table 17.1 summarizes the relationships of the events to `Look`.

Table 17.1 – Events and Look

Event	Cleared on Look?	Consuming XTI operations
Connect_Request_Received	No	`Listen`
Connect_Response_Received	No	`Connect(1), Confirm_Connection`
Normal_Data_Received	No	`Receive, Receive_Data_Unit, Receive_And_Scatter_Data, Receive_And_Scatter_Data_Unit`
Expedited_Data_Received	No	`Receive, Receive_And_Scatter_Data`
Disconnect_Request_Received	No	`Retrieve_Disconnect_Info`
Error_In_Previously_Sent_Datagram	No	`Retrieve_Data_Unit_Error`
Orderly_Release_Request_Received	No	`Acknowledge_Orderly_Release`
Orderly_Release_Request_Received	No	`Acknowledge_Orderly_Release_-With_Data`
Okay_To_Send_Normal_Data	Yes	`Send, Gather_And_Send_Data, Gather_And_Send_Data_Unit`
Okay_To_Send_Expedited_Data	Yes	`Send, Gather_And_Send_Data`

NOTE:

In the case of the `Connect` procedure, the `Connect_Response_Received` event is both generated and consumed by the execution of the procedure and is, therefore, not visible to the application.

17.1.9 Effect of Signals

With `POSIX_IO.Non_Blocking` either set or clear, XTI calls may be affected by signals. Unless the descriptions of the operations specify otherwise, they behave as described in this subclause.

If an XTI operation is interrupted by a signal while blocked under circumstances where, if `POSIX_IO.Non_Blocking` were set, it would have returned because no event was available, then it shall raise `POSIX_Error` with error code `Interrupted_Operation`, and the state of the communications endpoint shall be unchanged.

In addition, any XTI operation may under implementation-defined conditions raise the exception `POSIX_Error` with error code `Interrupted_Operation`. In such a

case, the state of the endpoint shall not have been changed, no data shall have been sent or received, and any buffers provided by the application for return values may have been overwritten.

17.1.10 Event Management

The XTI operations deal with one communications endpoint at a time. They do not enable the application simultaneously to wait for several events from different sources, in particular from several connections. An application can, however, wait for events from different sources by using an event management service in conjunction with XTI operations.

For an event management service to be usable in this way, it needs to be able to notify a process of the events shown in 17.1.8.

Event management services that can be used in conjunction with XTI include `Poll` (see 19.1.1) and `Select_File` (see 19.1.2).

17.1.11 Classification of Operations

The following operations correspond to the subset of XTI operations common to connection mode and connectionless mode services; `Bind`, `Close`, `Look`, `Open`, `Synchronize_Endpoint`, and `Unbind`.

The following operations are provided as part of the XTI connection mode service; `Accept_Connection`, `Connect`, `Listen`, `Receive`, `Confirm_Connection`, `Retrieve_Disconnect_Info`, `Receive_And_Scatter_Data`, `Send`, `Send_Disconnect_Request`, and `Gather_And_Send_Data`.

The following XTI operations support connectionless mode service; `Receive_Data_Unit`, `Retrieve_Data_Unit_Error`, `Receive_And_Scatter_Data_Unit`, `Send_Data_Unit`, and `Gather_And_Send_Data_Unit`.

The following utility operations are also provided; `Get_Protocol_Address`, `Get_Info`, `Get_Current_State`, and `Manage_Options`.

The orderly release mechanism (using `Initiate_Orderly_Release` or `Initiate_Orderly_Release_With_Data` and `Acknowledge_Orderly_Release` or `Acknowledge_Orderly_Release_With_Data`) shall be supported only for `Connection_Mode_With_Orderly_Release` type providers. Use with other providers shall cause the `XTI_Operation_Not_Supported` error to be returned.

NOTE: Applications that depend on orderly release will, therefore, not be portable to systems that use the ISO Transport Protocol.

Optional mechanisms include

— The ability to manage (enqueue) more than one incoming connection indication at any one time

— The ability to check the address of the caller passed with `Accept_Connection`

Table 17.2 presents all the operations defined in XTI. The character √ indicates that the mapping of that operation is possible onto a connection mode or connectionless mode service. Utility operations are classified as “General”.

17.2 States and Events

This clause describes the possible states of the communications endpoint as seen by the application, describe the incoming and outgoing events that may occur on any connection, and identify the allowable sequence of subprogram calls. Given a current state and event, the transition to the next state is shown as well as any actions that shall be taken in the course of the transition.

The allowable sequence of operations is described in Table 17.4, Table 17.5, Table 17.6, and Figure 17.1. The operations, `Get_Protocol_Address`, `Get_Current_State`, `Get_Info`, `Look`, `Manage_Options`, and `Synchronize_Endpoint` are excluded from the state tables because they do not affect the state of the interface. Each of these operations may be issued from any state except the Uninitialized state.

NOTE: Although the `Synchronize_Endpoint` function does not affect the state of the interface, it may affect the visibility of that state for the calling application when multiple processes are using the same endpoint (see 17.4.29).

17.2.1 Communications Interface States

The possible states of the communications endpoint as seen by the application are summarized in Table 17.3. States Outgoing Release and Incoming Release are significant only if the optional orderly release mechanism is both supported and used. The service type may be connection mode, connection mode with orderly release, or connectionless mode.

17.2.2 Outgoing Events

Some of the outgoing events (*e.g.*, Accept Connection1, Accept Connection2, and Accept Connection3) are distinguished by the context in which they occur. The context is based on the values of the following:

Outstanding Connection Count

: Count of outstanding connection indications (connection indications passed to the application, but not accepted or rejected).

Listening Communications Endpoint

: File descriptor of the listening communications endpoint (*i.e.*, the `Listening_Endpoint` parameter passed to `Accept_Connection`).

Responding Communications Endpoint

: File descriptor of the communications endpoint where a connection shall be accepted (*i.e.*, the `Responding_Endpoint` parameter passed to `Accept_Connection`).

The following outgoing events correspond to the successful return or error return of the specified communications operations causing XTI to change state, where these operations send a request or response to the communications provider.

1) Outstanding Connection Count is only meaningful for the listening communications endpoint (Listening Communications Endpoint).

Table 17.2 – Classification of the XTI Functions

Function	Connection	Connectionless	General
Accept_Connection	√		
Acknowledge_Orderly_Release	√		
Acknowledge_Orderly_Release_With_Data	√		
Bind	√	√	
Close	√	√	
Confirm_Connection	√		
Connect	√		
Gather_And_Send_Data	√		
Gather_And_Send_Data_Unit		√	
Get_Protocol_Address			√
Get_Info			√
Get_Current_State			√
Initiate_Orderly_Release	√		
Initiate_Orderly_Release_With_Data	√		
Listen	√		
Look	√	√	
Open	√	√	
Manage_Options			√
Receive	√		
Receive_Data_Unit		√	
Receive_And_Scatter_Data	√		
Receive_And_Scatter_Data_Unit		√	
Retrieve_Data_Unit_Error		√	
Retrieve_Disconnect_Info	√		
Send	√		
Send_Disconnect_Request	√		
Send_Data_Unit		√	
Synchronize_Endpoint			√
Unbind	√	√	

Accept Connection1

Successful return of `Accept_Connection` with Outstanding Connection Count[1)] = 1, and Listening Communications Endpoint = Responding Communications Endpoint. Service types: `Connection_Mode`, `Connection_Mode_With_Orderly_Release`

Accept Connection2

Successful return of `Accept_Connection` with Outstanding Connection Count[1)] = 1, and Listening Communications Endpoint / = Responding Communications Endpoint. Service types: `Connection_Mode`, `Connection_Mode_With_Orderly_Release`

Accept Connection3

Successful return of `Accept_Connection` with Outstanding Connection Count[2)] > 1. Service types: `Connection_Mode`, `Connection_Mode_With_Orderly_Release`

Bind

Successful return of Bind. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode

Connect

Successful return of Connect with POSIX_IO.Non_Blocking clear. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Connect Error

No_Data_Available error on Connect (POSIX_IO.Non_Blocking set), or Event_Requires_Attention error due to a disconnection indication arriving on the communications endpoint. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Close

Successful return of Close. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode

Manage Options

Successful return of Manage_Options. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode

Open

Successful return of Open. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode

Release

Successful return of Initiate_Orderly_Release or Initiate_Orderly_Release_With_Data. Service types: Connection_Mode_With_Orderly_Release

Send

Successful return of Send or Gather_And_Send_Data. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Send Data Unit

Successful return of Send_Data_Unit or Gather_And_Send_Data_Unit. Service types: Connectionless_Mode

Send Disconnect1

Successful return of Send_Disconnect_Request with Outstanding Connection Count $<=1$. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Send Disconnect2

Successful return of Send_Disconnect_Request with Outstanding Connection Count >1. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Unbind

Successful return of Unbind. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode

Table 17.3 – Communication Interface States

State	Description	Service Type
Data Transfer	Data transfer	Connection_Mode, Connection_Mode_With_Orderly_Release
Idle	No connection established	Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode
Incoming Connect	Incoming connection pending for passive application	Connection_Mode, Connection_Mode_With_Orderly_Release
Incoming Release	Incoming orderly release pending (waiting to send orderly release request)	Connection_Mode_With_Orderly_Release
Outgoing Connect	Outgoing connection pending for active application	Connection_Mode, Connection_Mode_With_Orderly_Release
Outgoing Release	Outgoing orderly release pending (waiting for orderly release indication)	Connection_Mode_With_Orderly_Release
Unbound	Unbound	Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode
Uninitialized	Uninitialized— initial and final state of interface	Connection_Mode, Connection_Mode_With_Orderly_Release, Connectionless_Mode

17.2.3 Incoming Events

The following incoming events correspond to the successful return of the specified communications operations, where these operations retrieve data or event information from the communications provider. One incoming event is not associated directly with the return of an operation on a given communications endpoint.

Acknowledge Release

Successful return of Acknowledge_Orderly_Release or Acknowledge_Orderly_Release_With_Data. Service types: Connection_Mode_With_Orderly_Release

Confirm Connection

Successful return of Confirm_Connection. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Listen

Successful return of Listen. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Pass Connection

Receive a passed connection. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Receive

Successful return of Receive or Receive_And_Scatter_Data. Service types: Connection_Mode, Connection_Mode_With_Orderly_Release

Receive Data Unit

Successful return of `Receive_Data_Unit` or `Receive_And_Scatter_Data_Unit`. Service types: `Connectionless_Mode`

Receive Data Error

Successful return of `Retrieve_Data_Unit_Error`. Service types: `Connectionless_Mode`

Receive Disconnect1

Successful return of `Retrieve_Disconnect_Info` with Outstanding Connection Count = `0`. Service types: `Connection_Mode`, `Connection_Mode_With_Orderly_Release`

Receive Disconnect2

Successful return of `Retrieve_Disconnect_Info` with Outstanding Connection Count = `1`. Service types: `Connection_Mode`, `Connection_Mode_With_Orderly_Release`

Receive Disconnect3

Successful return of `Retrieve_Disconnect_Info` with Outstanding Connection Count > `1`. Service types: `Connection_Mode`, `Connection_Mode_With_Orderly_Release`

Pass Connection occurs when an application transfers a connection to another communications endpoint. This event occurs on the endpoint that is being passed the connection, despite the fact that no operation is issued on that endpoint. Pass Connection event is included in the state tables to describe what happens when an application accepts a connection on another communications endpoint.

Receive Disconnect1, Receive Disconnect2 and Receive Disconnect3 are distinguished from the other incoming events by the context in which they occur. The context is based on the value of Outstanding Connection Count, which is the count of outstanding connection indications on the listening communications endpoint.

17.2.4 Transition Actions

Some state transitions listed in Table 17.4 and Table 17.6 are accompanied by a list of actions that the implementation shall take. These actions are represented by the notation (n), where n is the number of the specific action as follows:

(1) Set Outstanding Connection Count to zero.

(2) Increment Outstanding Connection Count.

(3) Decrement Outstanding Connection Count.

(4) Pass a connection to another communications endpoint as indicated in `Accept_Connection`.

17.2.5 State Tables

The possible next states, given the current state and event, are shown in tabular form in Table 17.4, Table 17.5,k and Table 17.6, and in diagrammatic form in Figure 17.1. The state is that of the communications endpoint as seen by the application.

The contents of each cell represent the next state given the current state (column) and the current incoming or outgoing event (row). An empty cell represents a state-event combination that is invalid. Along with the next state, each cell may include an action list (as specified in 17.2.4). The implementation shall take the specific actions in the order specified in the state table.

A separate table is shown for initialization/de-initialization, data transfer in connectionless mode and connection/release/data transfer in connection mode.

Table 17.4 – Initialization/De-initialization State Table

Event	State		
	Uninitialized	Unbound	Idle
Open	Unbound		
Bind		Idle(1)	
Unbind			Unbound
Close		Uninitialized	Uninitialized

NOTE:

For explanation of state transition action (1) see 17.2.4.

Table 17.5 – Data Transfer State Table for Connectionless-Mode Service

Event	State
	Idle
Send_Data_Unit	Idle
Receive_Data_Unit	Idle
Receive_Data_Error	Idle

17.2.6 Events and `Event_Requires_Attention` Error Indication

Table 17.7 describes the events that cause an XTI call to return with an `Event_Requires_Attention` error.

Table 17.6 – Connection/Release/Data Transfer State Table for Connection-Mode Service

Event	State						
	Idle	**Outgoing Connect**	**Incoming Connect**	**Data Transfer**	**Outgoing Release**	**Incoming Release**	**Unbound**
Accept Connection1			Data Transfer(3)				
Accept Connection2			Idle(3)(4)				
Accept Connection3			Incoming Connect(3)(4)				
Acknowledge Release				Incoming Release	Idle		
Close	Uninitialized	Uninitialized	Uninitialized	Uninitialized	Uninitialized	Uninitialized	Uninitialized
Confirm Connection		Data Transfer					
Connect	Data Transfer						
Connect Error	Outgoing Connect						
Listen	Incoming Connect (2)		Incoming Connect (2)				
Manage Options	Idle	Outgoing Connect	Incoming Connect	Data Transfer	Outgoing Release	Incoming Release	Unbound
Pass Connection	Data Transfer						Data Transfer
Receive				Data Transfer	Outgoing Release		
Receive Disconnect1		Idle		Idle	Idle	Idle	
Receive Disconnect2			Idle (3)				
Receive Disconnect3			Incoming Connect (3)				
Release				Outgoing Release		Idle	
Send				Data Transfer		Incoming Release	
Send Disconnect1		Idle	Idle(3)	Idle	Idle	Idle	
Send Disconnect2			Incoming Connect(3)				

NOTE:

For explanations of state transition actions (2), (3), and (4) see 17.2.4.

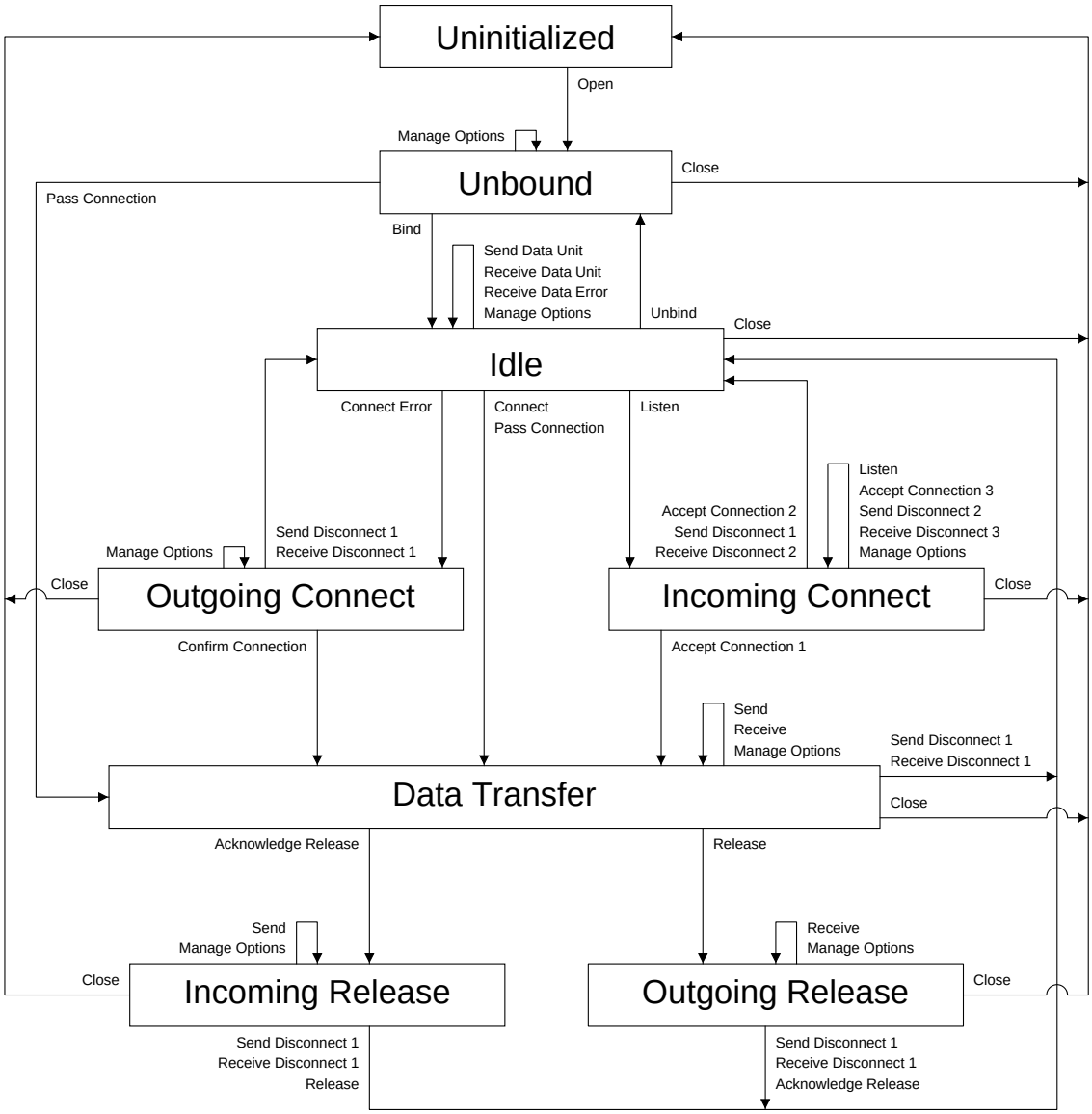

Figure 17.1 – XTI State Diagram

Once an `Event_Requires_Attention` error has been received on a communications endpoint via an XTI operation, subsequent calls to that and other XTI operations, to which the same `Event_Requires_Attention` error applies, shall continue to return `Event_Requires_Attention` until the event is consumed. An event causing the `Event_Requires_Attention` error can be determined by calling `Look` and then can be consumed by calling the corresponding consuming XTI operation as defined in Table 17.1.

17.3 The Use of Options

17.3.1 Generalities

The operations Accept_Connection, Connect, Listen, Manage_Options, Confirm_Connection, Receive_Data_Unit, Retrieve_Data_Unit_Error, Receive_And_Scatter_Data_Unit, Send_Data_Unit, and Gather_And_Send_Data_Unit use an Options argument of type Protocol_Option_List as an explicit parameter or included in a complex argument. This object is used to convey options between the application and the communications provider.

Table 17.7 – Event_Requires_Attention Error Indications

Function Call	Event(s)
Accept_Connection	Disconnect_Request_Received, Connect_Request_Received
Connect	Disconnect_Request_Received, Connect_Request_Received(1)
Listen	Disconnect_Request_Received(2)
Receive	Disconnect_Request_Received, Orderly_Release_Request_Received(3)
Receive_And_Scatter_Data	Disconnect_Request_Received, Orderly_Release_Request_Received(3)
Confirm_Connection	Disconnect_Request_Received
Acknowledge_Orderly_Release	Disconnect_Request_Received
Acknowledge_Orderly_Release_With_Data	Disconnect_Request_Received
Receive_Data_Unit	Error_In_Previously_Sent_Datagram
Receive_And_Scatter_Data_Unit	Error_In_Previously_Sent_Datagram
Send	Disconnect_Request_Received, Orderly_Release_Request_Received
Gather_And_Send_Data	Disconnect_Request_Received, Orderly_Release_Request_Received
Send_Data_Unit	Error_In_Previously_Sent_Datagram
Gather_And_Send_Data_Unit	Error_In_Previously_Sent_Datagram
Unbind	Connect_Request_Received, Normal_Data_Received(4), Error_In_Previously_Sent_Datagram
Initiate_Orderly_Release	Disconnect_Request_Received
Initiate_Orderly_Release_With_Data	Disconnect_Request_Received
Send_Disconnect_Request	Disconnect_Request_Received

NOTES:

(1) This event occurs only when a Connect is done on an endpoint that has been bound with a Request_Queue_Length>0 and for which a connection indication is pending.

(2) This event indicates a disconnection on an outstanding connection indication.

(3) This event occurs only when all pending data are read.

(4) Normal_Data_Received may only occur for the connectionless mode.

There is no general definition about the possible contents of options. There are general XTI options and those that are specific for each communications provider. Some

options allow the application to tailor its communication needs, for instance by asking for high throughput or low delay. Others allow the fine-tuning of the protocol behavior so that communication with unusual characteristics can be handled more effectively. Other options are for debugging.

All options have default values. Their values have meaning to and are defined by the protocol level in which they apply. However, their values can be negotiated by an application, the trivial form of such negotiation being when an application simply enforces the use of a specific value for an option. Often, the communications provider or even the remote application may have the right to negotiate a value of lesser quality than the proposed one; *i.e.*, a delay may become longer, or a throughput may become lower.

It is useful to differentiate between options that have end-to-end significance and those that do not. Options with end-to-end significance are intimately related to the particular connection or data unit (datagram) transmission. If the calling application specifies such an option, some ancillary information is transferred across the network in most cases. The interpretation and further processing of this information is protocol-specific. For instance, in an ISO connection mode communication, the calling application may specify quality-of-service parameters on connection establishment. Quality of service parameters are first processed and possibly lowered by the local communications provider, then sent to the remote communications provider that may degrade them again, and finally conveyed to the called application that makes the final selection and transmits the selected values back to the caller. Options that do not have end-to-end significance do not contain information destined for the remote application. Some have purely local relevance, *e.g.*, an option that enables debugging.

Other options influence the transmission, for instance the option that sets the IP Time To Live field or TCP No Delay (see D.2.3). Local options are negotiated solely between the application and the local communications provider. The distinction between these two categories of option is as follows. On output, the operations `Listen`, `Receive_Data_Unit`, and `Receive_And_Scatter_Data_Unit` return options with end-to-end significance only. The operations `Confirm_Connection` and `Retrieve_Data_Unit_Error` may return options of both categories. On input, options of both categories may be specified with `Accept_Connection`, `Send_Data_Unit`, and `Gather_And_Send_Data_Unit`. The operations `Connect` and `Manage_Options` can process and return both categories of options.

The communications provider shall have a default value for each option it supports. These defaults are sufficient for the majority of communication relations. Hence, an application should only request options actually needed to perform the task and leave all others at their default value.

This section describes the general framework for the use of options. This framework is obligatory for all communications providers. The specific options that are legal for use with a specific communications provider are described in the provider-specific annexes (see D.2). General XTI options are described in 17.4.18.

17.3.2 Options Format

Options are conveyed via a `Protocol_Option_List` object. Several options can be concatenated in a protocol option list. Each option in the list is a `Protocol_Option` object. Each option may have an associated `Option_Value`, `Option_Value_Array`, `Linger_Info`, or protocol-specific object.

The `Option_Value`, `Option_Value_Array`, `Linger_Info`, or protocol-specific object contains the actual option value. The `Protocol_Option` object is described in 17.4.1.5.

A communications provider embodies a stack of protocols. The Level attribute of the `Protocol_Option` object identifies the XTI level or a protocol of the communications provider such as TCP or ISO/IEC 8073 {4}. The Name attribute identifies the option within the level. The Status attribute is used by the XTI level or the communications provider to indicate success or failure of a negotiation (see 17.3.2.5 and 17.4.18).

This subclause describes the general rules governing the passing and retrieving of options and the error conditions that can occur. Unless explicitly restricted, these rules apply to all operations that allow the exchange of options.

17.3.2.1 Multiple Options and Options Levels

When multiple options are specified in a protocol option list on input, different rules apply to the levels that may be specified, depending on the procedure call. Multiple options specified on input to `Manage_Options` shall address the same option level. Options specified on input to `Connect`, `Accept_Connection`, `Send_Data_Unit`, and `Gather_And_Send_Data_Unit` can address different levels.

17.3.2.2 Illegal Options

Only legal options can be negotiated; illegal options cause failure. The legal values are defined for each option. (See 17.4.18 and D.2.)

If an illegal option is passed to `Manage_Options`, then the operation shall raise the exception `POSIX_Error` with error code `Incorrect_Or_Illegal_Option`. If an illegal option is passed to `Accept_Connection` or `Connect`, then either the operation shall fail with error code `Incorrect_Or_Illegal_Option` or the connection establishment shall fail at a later stage, depending on when the implementation detects the illegal option. A call to `Send_Data_Unit` or `Gather_And_Send_Data_Unit` shall either fail with `Incorrect_Or_Illegal_Option` or shall successfully return, but a `Error_In_Previously_Sent_Datagram` event shall be generated to indicate that the data unit (datagram) was not sent.

If the application passes multiple options in one call and one of them is illegal, the operation shall fail as described above. It is, however, possible that some or even all of the submitted legal options were successfully negotiated. The application can check the current status by a call to `Manage_Options` with the `Get_Current_Options` flag set (see 17.4.18).

Specifying an option level unknown to the communications provider does not cause failure in calls to `Connect`, `Accept_Connection`, `Send_Data_Unit`, or `Gather_And_Send_Data_Unit`; the option is discarded in these cases. The operation `Man-`

age_Options fails with error code Incorrect_Or_Illegal_Option. Specifying an option name that is unknown to or not supported by the protocol selected by the option level does not cause failure. The option is discarded in calls to Connect, Accept_Connection, Send_Data_Unit, or Gather_And_Send_Data_Unit. The operation Manage_Options returns Not_Supported as the Status attribute for the Protocol_Option object.

17.3.2.3 Initiating an Option Negotiation

An application initiates an option negotiation when calling Connect, Send_Data_Unit, Gather_And_Send_Data_Unit, or Manage_Options with the flag Negotiate_Options set.

The negotiation rules for these operations depend on whether an option request is an absolute requirement. Whether an option request is an absolute requirement is explicitly defined for each option (see 17.4.18 and D.2). In case of an ISO communications provider, for example, the option that requests use of expedited data is not an absolute requirement. On the other hand, the option that requests protection could be an absolute requirement.

NOTE: The notion absolute requirement originates from the quality-of-service parameters in ISO/IEC 8072 {3}. Its use is extended in this standard to all XTI protocol options.

If the proposed option value is an absolute requirement, three outcomes are possible:

— The negotiated value is the same as the proposed one. When the result of the negotiation is retrieved, the Status attribute for the Protocol_Option object is set to Success.

— The negotiation is rejected if the option is supported but the proposed value cannot be negotiated. Rejection of the negotiation leads to the following behavior:
  — Manage_Options successfully returns, but the returned Protocol_Option object has its Status attribute set to Failure.
  — Any attempt to establish a connection aborts; a Disconnect_Request_Received event occurs, and a call to Connect with POSIX_IO.Non_Blocking clear fails with error code Event_Requires_Attention.
  — Send_Data_Unit or Gather_And_Send_Data_Unit fails with Event_Requires_Attention or successfully returns, but an Error_In_Previously_Sent_Datagram event occurs to indicate that the data unit (datagram) was not sent.

  If multiple options are submitted in one call and one of them is rejected, XTI behaves as just described. Although the connection establishment or the data unit (datagram) transmission fails, options successfully negotiated before an option was rejected retain their negotiated values. There is no rollback mechanism (see 17.3.3).

  The operation Manage_Options attempts to negotiate each option. The Status attribute for the returned Protocol_Option objects indicate Success or Failure.

— If the local communications provider does not support the option at all, Manage_Options reports Not_Supported in the Status attribute. The operations Connect, Send_Data_Unit, and Gather_And_Send_Data_Unit ignore this option.

If the proposed option value is not an absolute requirement, two outcomes are possible:

— The negotiated value is of equal or lesser quality than the proposed one (*e.g.*, a delay may become longer).

When the result of the negotiation is retrieved, the Status attribute in `Protocol_Option` is set to `Success` if the negotiated value equals the proposed one or set to `Partial_Success` otherwise.

— If the local communications provider does not support the option at all, `Manage_Options` reports `Not_Supported` in the Status attribute. The operations `Connect`, `Send_Data_Unit`, and `Gather_And_Send_Data_Unit` ignore this option.

Unsupported options do not cause operations to fail or a connection to abort since different vendors possibly implement different subsets of options. Furthermore, future enhancements of XTI might encompass additional options that are unknown to earlier implementations of communications providers. The decision whether the missing support of an option is acceptable for the communication is left to the application.

The communications provider does not check for multiple occurrences of the same option, possibly with different option values. It simply processes the options in the protocol option list one after the other. However, the application shall not make any assumption about the order of processing.

Not all options are independent of one another. A requested option value might conflict with the value of another option that was specified in the same call or is currently effective (see 17.3.3). These conflicts may not be detected at once, but later they might lead to unpredictable results. If detected at negotiation time, these conflicts are resolved within the rules stated above. The outcomes may thus be quite different and depend on whether absolute or nonabsolute requests are involved in the conflict.

Conflicts should be detected at the time a connection is established or a data unit (datagram) is sent. If options are negotiated with `Manage_Options`, conflicts should not be detected at this time since independent processing of the requested options needs to allow for temporary inconsistencies.

When called, the operations `Connect`, `Send_Data_Unit`, and `Gather_And_Send_Data_Unit` initiate a negotiation of all options with end-to-end significance according to the rules of this section. Options not explicitly specified in the subprogram calls themselves are taken from an internal protocol option list that contains the values of a previous negotiation (see 17.3.3).

17.3.2.4 Responding to a Negotiation Proposal

In connection mode communication, some protocols give the peer communications applications the opportunity to negotiate characteristics of the communications connection to be established. These characteristics are options with end-to-end significance. With the connection indication, the called application receives (via `Listen`) a proposal about the option values that should be effective for this connection. The

called application can accept this proposal or weaken it by choosing values of lower quality (*e.g.*, longer delays than proposed). The called application can, of course, refuse the connection altogether.

The called application responds to a negotiation proposal via `Accept_Connection`. If the called application tries to negotiate an option of higher quality than proposed, the outcome depends on the protocol to which that option applies. Some protocols may reject the option. Some protocols take other appropriate action described in D.2. If an option is rejected, the connection fails and a `Disconnect_Request_Received` event occurs; it depends on timing and implementation conditions whether `Accept_Connection` still succeeds or fails with error code `Event_Requires_Attention`.

If multiple options are submitted with `Accept_Connection` and one of them is rejected, the connection fails as described in the paragraph above. Options that could be successfully negotiated before the erroneous option was processed retain their negotiated value. There is no roll-back mechanism (see 17.3.3).

The response options can be specified with the `Accept_Connection` call. Alternatively, they can be specified by calling `Manage_Options` with the file descriptor (endpoint) that will subsequently be passed as `Responding_Endpoint` to `Accept_Connection` to identify the responding endpoint (see 17.3.3). In case of conflict between option settings made by calls to `Manage_Options` and `Accept_Connection` at different times, the latest settings when `Accept_Connection` is called shall prevail. The response to a negotiation proposal is activated when `Accept_Connection` is called.

A `Manage_Options` call with erroneous option values as described above shall succeed; the connection aborts at the time `Accept_Connection` is called.

The connection also fails if the selected option values lead to contradictions.

The procedure `Accept_Connection` does not check for multiple specification of an option (see 17.3.2.3). Unsupported options are ignored.

17.3.2.5 Retrieving Information about Options

This subclause describes how a communications application can retrieve information about options. An application needs to be able to

— Know the result of a negotiation (*e.g.*, at the end of a connection establishment).

— Know the proposed option values under negotiation (during connection establishment).

— Retrieve option values sent by the remote application for notification only (*e.g.*, IP options).

— Check option values currently effective for the communications endpoint.

To this end, `Confirm_Connection`, `Connect`, and `Listen` include a `Connection_Info` argument with an Options attribute of type `Protocol_Option_List`, and operations `Manage_Options`, `Retrieve_Data_Unit_Error`, `Receive_Data_Unit`,

and Receive_And_Scatter_Data_Unit include a Protocol_Option_List parameter. The application shall supply the buffer where the options shall be written (see 17.4.1.6).

Which options are returned depend on the subprogram call involved:

Connect

With POSIX_IO.Non_Blocking clear, the operation returns the values of all options with end-to-end significance that were received with the connection response and the negotiated values of those options with end-to-end significance that had been specified on input. However, options specified on input in the Connect call that are not supported or refer to an unknown option level are discarded and not returned on output.

The Status attribute of each option returned indicates whether the proposed value (Success) or a degraded value (Partial_Success) has been negotiated. The Status attribute of received ancillary information (*e.g.*, IP options) that is not subject to negotiation is always set to Success.

Confirm_Connection

(same as Connect)

Listen

The received options with end-to-end significance are related to the incoming connection (identified by the sequence number), not to the listening endpoint.

NOTE: A call to Manage_Options with Request_Flags set to Get_Current_Options will thus often return different option values than a call to Listen will since Manage_Options returns the options that are related to the endpoint. The option values currently effective for the listening endpoint can, however, affect the values retrieved by Listen since the communications provider might be involved in the process of negotiating the options that are related to the connection.

The number of received options may be variable for subsequent connection indications, since many options with end-to-end significance are only transmitted on explicit demand by the calling application (*e.g.*, IP options or ISO/IEC 8072 {3} throughput). It is even possible that no options at all are returned.

The Status attribute is irrelevant.

Receive_Data_Unit

The received options with end-to-end significance are related to the incoming data unit (datagram), not to the communications endpoint. Thus, if the same options are specified in a call to Manage_Options with Request_Flags set to Get_Current_Options, Manage_Options usually does not return the same values.

The number of options received may vary from call to call.

The Status attribute is irrelevant.

Receive_And_Scatter_Data_Unit

(same as Receive_Data_Unit)

Retrieve_Data_Unit_Error

The returned options are related to the options input at the previous Send_Data_Unit or Gather_And_Send_Data_Unit call that produced the error.

Which options are returned and which values they have depend on the specific error condition.

The Status attribute is irrelevant.

`Manage_Options`

This operation processes and returns both categories of options. It acts on options related to the specified communications endpoint, not on options related to a connection indication or an incoming data unit (datagram). A detailed description is given in 17.4.18.

17.3.2.6 Privileged and Read-Only Options

Privileged options or option values are those that may be requested by privileged applications only. The meaning of privilege is implementation defined.

Read-only options serve for information only. The application may be allowed to read the option value, but not to change it. For instance, to select the value of a protocol timer or the maximum length of a protocol data unit may be too subtle to leave to the application, although the knowledge about this value might be of some interest. An option might be read-only for all applications or solely for nonprivileged applications. A privileged option might be inaccessible or read-only for nonprivileged applications.

An option might be negotiable in some XTI states and read-only in other XTI states. The XTI states are defined in 17.2. For instance, the ISO quality-of-service options are negotiable in the states Idle and Incoming Connect and read-only in all other states (except Uninitialized).

If an application requests negotiation of a read-only option or a nonprivileged application requests illegal access to a privileged option, the following outcomes are possible:

— `Manage_Options` successfully returns, but the returned option has its Status attribute set to `Not_Supported` if a privileged option was requested illegally and to `Read_Only` if modification of a read-only option was requested.

— If negotiation of a read-only option is requested, `Accept_Connection` or `Connect` fail with error code `Insufficient_Permission`; or the connection establishment aborts, and a `Disconnect_Request_Received` event occurs. If the connection aborts, a call to `Connect` with `POSIX_IO.Non_Blocking` clear fails with `Event_Requires_Attention`. It depends on timing and implementation conditions whether a `Accept_Connection` call still succeeds or fails with `Event_Requires_Attention`.

If a privileged option is illegally requested, the request shall be ignored but no error indication shall be given. (A nonprivileged application shall not be able to select an option that is privileged or unsupported.)

— If negotiation of a read-only option is requested, `Send_Data_Unit` or `Gather_And_Send_Data_Unit` may generate `Event_Requires_Attention` or successfully return, but an `Error_In_Previously_Sent_Datagram` event occurs to indicate that the data unit (datagram) was not sent.

If a privileged option is illegally requested, the option is quietly ignored. (A nonprivileged application shall not be able to select an option that is privileged or unsupported.)

If multiple options are submitted to Connect, Accept_Connection, Send_Data_Unit, or Gather_And_Send_Data_Unit and a read-only option is rejected, the connection or the data unit (datagram) transmission fails as described. Options that could be successfully negotiated before the erroneous option was processed retain their negotiated values. There is no roll-back mechanism (see also 17.3.3).

17.3.3 Option Management of a Communication Endpoint

This subclause describes how option management works during the lifetime of a communications endpoint.

Each communications endpoint is (logically) associated with an internal protocol option list. When a communications endpoint is created, this list is filled with a system default value for each supported option. Depending on the option, the default may be "OPTION ENABLED," "OPTION DISABLED," or denote a time span, *etc.* These default settings are appropriate for most uses. Whenever an option value is modified in the course of an option negotiation, the modified value is written to this list and overwrites the previous one. At any time, the protocol option list contains all option values that are currently effective for this communications endpoint.

The current value of an option can be retrieved at any time by calling Manage_Options with Request_Flags set to Get_Current_Options. Calling Manage_Options with the Request_Flags parameter set to Get_Default_Options yields the system default for the specified option.

An application can negotiate new option values by calling Manage_Options with Request_Flags set to Negotiate_Options. The negotiation follows the rules in 17.3.2.

Some options may be modified only in specific XTI states and are read-only in other XTI states. Many options with end-to-end significance, for instance, may not be changed in the Data Transfer state, and an attempt to do so shall fail (see 17.3.2.6).

The legal states for each option are specified with its definition (see 17.4.18 and D.2).

Options with end-to-end significance take effect at the time a connection is established or a data unit (datagram) is transmitted, for example, if they contain information that is transmitted across the network or determines specific transmission characteristics. If such an option is modified by a call to Manage_Options, the communications provider checks whether the option is supported and negotiates a value according to its current status. This value is written to the internal protocol option list. The final negotiation takes place if the connection is established or the data unit (datagram) is transmitted. This final negotiation can result in a degradation of the option value or even in a negotiation failure. The negotiated values are written to the internal protocol option list.

Some options may be changed in the Data Transfer state, *e.g.*, those specifying buffer sizes. Such changes might affect the transmission characteristics and lead to unexpected side effects (*e.g.*, data loss if a buffer size was shortened).

The application can explicitly specify options in both categories discussed above on input when calling Connect, Accept_Connection, Send_Data_Unit, or Gather_And_Send_Data_Unit. The options are at first locally negotiated option-by-option,

and the resulting values written to the internal protocol option list. The modified protocol option list is then used if a further negotiation step across the network is required, as for instance in ISO COTS communication. The newly negotiated values are then written to the internal protocol option list.

At any stage, a negotiation failure can lead to an abort of the transmission. If a transmission aborts, the protocol option list shall preserve the content it had at the time the failure occurred. Options that could be negotiated just before the error occurred are written back to the protocol option list, whether the XTI operation fails or succeeds.

It is up to the application to decide which options it explicitly specifies on input when calling `Connect`, `Accept_Connection`, `Send_Data_Unit` or `Gather_And_Send_Data_Unit`. The application need not pass options at all, by omitting the `Options` parameter to the procedure or by providing an empty list (*i.e.*, using the `Make_Empty` procedure on the `Protocol_Option_List` object). In this case the current content of the internal protocol option list is used for negotiation without prior modification.

The negotiation procedure for options at the time of a `Connect`, `Accept_Connection`, `Send_Data_Unit` or `Gather_And_Send_Data_Unit` operation shall obey the rules of 17.3.2.3 and 17.3.2.4, whether the options were explicitly specified during the call or implicitly taken from the internal protocol option list.

The application shall not make assumptions about the order in which options are processed during negotiation.

A value in the protocol option list is only modified as a result of a successful negotiation of this option. It is, in particular, not changed by a connection release. There is no history mechanism that would restore the buffer state existing prior to the connection establishment or the data unit (datagram) transmission. The application needs to be aware that a connection establishment or a data unit (datagram) transmission may change the internal protocol option list, even if each option was originally initialized to its default value.

17.3.4 Supplements

This section contains supplementary remarks and a short summary.

17.3.4.1 The Option Value `Unspecified`

Some options may not have a fully specified value all the time. An ISO COTS communications provider, for instance, that supports several protocol classes, might not have a preselected preferred class before a connection establishment is initiated. At the time of the connection request, the communications provider may conclude from the destination address, quality-of-service parameters, and other locally available information which preferred class it should use. An application asking for the default value of the preferred class option in the `Idle` state would get the value `Unspecified`. This value indicates that the communications provider did not yet select a value. The application could negotiate a different value as the preferred class, *e.g.*, `Transport_Class_2`. The communications provider would then be forced to initiate a connection request with class 2 as the preferred class.

An XTI implementation may also return the value `Unspecified` if it can currently not access the option value, for example, in the Unbound state in systems where the protocol stacks reside on separate controller cards and not in the host. The implementation shall never return `Unspecified` if the option is not supported at all.

If `Unspecified` is a legal value for a specific option, it may also be used by the application on input. It is used to indicate that choosing an appropriate value is left to the provider. `Unspecified` is especially useful in complex options like ISO throughput, where the option value has an internal structure (see the Throughput XTI option in D.2). The application may leave some attributes unspecified by selecting this value. If the application proposes `Unspecified`, the communications provider is free to select an appropriate value. The value chosen by the provider might be the default value, some other explicit value, or `Unspecified`.

For each option, it is specified whether `Unspecified` is a legal value for negotiation purposes.

17.3.4.2 The `Communications_Provider_Info` Argument

The operations `Open` and `Get_Info` return values representing characteristics of the communications provider in an object of type `Communications_Provider_Info` as the parameter `Info`. The Max Size Protocol Options attribute of `Info` is used to allocate storage for a protocol option list to be used in an XTI operation.

In general, the Max Size Protocol Options attribute also includes the size of privileged options, even if these are not read-only for nonprivileged applications. Alternatively, an implementation can choose to return different values in the Max Size Protocol Options attribute for privileged and nonprivileged applications. The values of the Max Size Connect Data, Max Size Disconnect Data, Max Size SDU, and Max Size SEDU attributes possibly diminish as soon as the Data Transfer state is entered. Calling `Manage_Options` does not influence these values.

17.3.4.3 Summary

— The format of an option is defined by a `Protocol_Option` object. The `Protocol_Option` may have optionally associated with it an `Option_Value`, `Option_Value_Array`, `Linger_Info`, or protocol-specific object.

— On input, several options can be specified in an input parameter. Functions and procedures are provided for manipulating the options within a protocol option list.

— There are options that have end-to-end significance and options that do not. On output, the operations `Listen`, `Receive_Data_Unit`, and `Receive_And_Scatter_Data_Unit` return options with end-to-end significance only. The operations `Confirm_Connection` and `Retrieve_Data_Unit_Error` may return options of both categories. On input, options of both categories may be specified with `Accept_Connection`, `Send_Data_Unit`, and `Gather_And_Send_Data_Unit`. The operations `Connect` and `Manage_Options` can process and return both categories of options.

— A communications endpoint is (logically) associated with an internal protocol option list where the currently effective values are stored. Each successful ne-

gotiation of an option modifies this list, regardless of whether the call initiating the negotiation succeeds.

— When calling Connect, Accept_Connection, Send_Data_Unit, or Gather_-And_Send_Data_Unit, the application can choose to submit the currently effective option values by omitting the Options parameter to the procedure or by providing an empty list (*i.e.*, using the Make_Empty procedure on the Protocol_Option_List object).

— If a connection is accepted via Accept_Connection, the explicitly specified option values together with the currently effective option values of the Responding_Endpoint, not of the Listening_Endpoint, matter in this negotiation step.

— The options returned by Retrieve_Data_Unit_Error are those negotiated with the outgoing data unit (datagram) that produced the error. If the error occurred during option negotiation, the returned option might represent some mixture of partly negotiated and not-yet negotiated options.

17.4 Package POSIX_XTI

The package POSIX_XTI provides the XTI protocol independent process to process communications interface.

The functionality described in this clause is optional. If the XTI Detailed Network Interface option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this clause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this clause.

```
with POSIX,
     POSIX_IO,
     POSIX_Limits,
     System;
package POSIX_XTI is
   -- 17.4.1 Common Data Types and Constants
   -- 17.4.1.1 Flags
   type XTI_Flags is new POSIX.Option_Set;
   Expedited_Data      : constant XTI_Flags := implementation-defined;
   More_Data           : constant XTI_Flags := implementation-defined;
   Push_Data           : constant XTI_Flags := implementation-defined;
   type Options_Flags is private;
   Check_Options       : constant Options_Flags;
   Get_Current_Options : constant Options_Flags;
   Get_Default_Options : constant Options_Flags;
   Negotiate_Options   : constant Options_Flags;
   -- 17.4.1.2 Protocol-Specific Service Limits
   type Communications_Provider_Info is private;
   type CP_Flags is new POSIX.Option_Set;
   Orderly_Release_Data_Supported : constant CP_Flags := implementation-defined;
   Zero_Length_SDU_Supported      : constant CP_Flags := implementation-defined;
   type Service_Type is private;
   Connection_Mode                     : constant Service_Type;
   Connection_Mode_With_Orderly_Release : constant Service_Type;
   Connectionless_Mode                 : constant Service_Type;
```

```
function Protocol_Addresses_Are_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Protocol_Address
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Protocol_Options_Are_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Protocol_Options
   (Info_Item : Communications_Provider_Info)
  return Positive;
function SDU_Is_Supported
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function SDU_Is_Infinite
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function SDU_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_SDU
   (Info_Item : Communications_Provider_Info)
  return Positive;
function SEDU_Is_Supported
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function SEDU_Is_Infinite
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function SEDU_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_SEDU
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Connect_Data_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Connect_Data
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Disconnect_Data_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Disconnect_Data
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Get_CP_Flags
   (Info_Item : Communications_Provider_Info)
  return CP_Flags;
function Get_Service_Type
   (Info_Item : Communications_Provider_Info)
  return Service_Type;
-- 17.4.1.3 XTI Addresses
type XTI_Address_Pointer is private;
Null_XTI_Address : constant XTI_Address_Pointer;
```

```
-- 17.4.1.4 Linger Information Objects
type Linger_Info is private;
subtype Linger_Time is POSIX.Seconds range 1 .. POSIX.Seconds'Last;
type Linger_Option is (Linger_Off, Linger_On);
function Get_Status (Item : Linger_Info)
  return Linger_Option;
procedure Set_Status
   (Item   : in out Linger_Info;
    Linger : in     Linger_Option);
function Period_Is_Infinite (Item : Linger_Info)
  return Boolean;
function Period_Is_Unspecified (Item : Linger_Info)
  return Boolean;
function Get_Period (Item : Linger_Info)
  return Linger_Time;
procedure Set_Period_Infinite
   (Item : in out Linger_Info);
procedure Set_Period_Unspecified
   (Item : in out Linger_Info);
procedure Set_Period
   (Item : in out Linger_Info;
    Time : in     Linger_Time);
-- 17.4.1.5 Protocol Option Objects
type Protocol_Option is private;
type Option_Value is range implementation-defined;
type Option_Level is range implementation-defined;
type Option_Name is range implementation-defined;
type Option_Status is private;
Success         : constant Option_Status := implementation-defined;
Partial_Success : constant Option_Status := implementation-defined;
Failure         : constant Option_Status := implementation-defined;
Read_Only       : constant Option_Status := implementation-defined;
Not_Supported   : constant Option_Status := implementation-defined;
type Option_Value_Array is array (Positive range <>) of Option_Value;
function Get_Level (Option_Item : Protocol_Option)
  return Option_Level;
function Get_Name (Option_Item : Protocol_Option)
  return Option_Name;
function Get_Status (Option_Item : Protocol_Option)
  return Option_Status;
function Get_Value (Option_Item : Protocol_Option)
  return Option_Value;
function Get_Value (Option_Item : Protocol_Option)
  return Option_Value_Array;
function Get_Value (Option_Item : Protocol_Option)
  return Linger_Info;
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name);
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name;
    Value       : in      Option_Value);
```

```
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name;
    Value       : in      Option_Value_Array);
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name;
    Value       : in      Linger_Info);
-- 17.4.1.6 Protocol Option List Objects
type Protocol_Option_List is private;
type Protocol_Option_List_Pointer is
  access all Protocol_Option_List;
procedure Make_Empty
   (Info_Item : in out Protocol_Option_List);
procedure Append
   (Info_Item : in out Protocol_Option_List;
    Option    : in     Protocol_Option);
type Octet_Buffer_Pointer is access all POSIX.Octet_Array;
procedure Set_Buffer
   (Info_Item      : in out Protocol_Option_List;
    Options_Buffer : in     Octet_Buffer_Pointer);
generic
   with procedure Action
     (Info : in     Protocol_Option;
      Quit : in out Boolean);
procedure For_Every_Item (Info_Item : in Protocol_Option_List);
function Number_Of_Options (Info_Item : Protocol_Option_List)
  return Natural;
procedure Get_Option
   (Info_Item     : in  Protocol_Option_List;
    Option_Number : in  Positive;
    Option        : out Protocol_Option);
-- 17.4.1.7 Disconnect Reason Codes
type Reason_Code is range implementation-defined;
-- 17.4.1.8 Connection Information Objects
type Connection_Info is limited private;
procedure Set_Address
   (Info_Item  : in out Connection_Info;
    Address    : in     XTI_Address_Pointer);
function Get_Options (Info_Item : Connection_Info)
  return Protocol_Option_List;
procedure Set_Options
   (Info_Item : in out Connection_Info;
    Options   : in     Protocol_Option_List_Pointer);
procedure Set_User_Data
   (Info_Item  : in out Connection_Info;
    User_Data  : in     System.Address;
    Max_Length : in     POSIX.IO_Count);
procedure Set_User_Data_Length
   (Info_Item  : in out Connection_Info;
    Length     : in     POSIX.IO_Count);
function Get_User_Data_Length
   (Info_Item : Connection_Info)
  return POSIX.IO_Count;
function Get_Sequence_Number (Info_Item : Connection_Info)
  return Integer;
```

```
procedure Set_Sequence_Number
   (Info_Item : in out Connection_Info;
    Number    : in     Integer);
-- 17.4.1.9 Scatter/Gather Vector Objects
subtype IO_Vector_Range is Positive range
   1 .. POSIX_Limits.XTI_IO_Vector_Maxima'Last;
type IO_Vector_Array is array
   (IO_Vector_Range range <>) of POSIX_IO.IO_Vector;
-- 17.4.1.10 Communications Interface States
type Interface_State is private;
Uninitialized    : constant Interface_State;
Unbound          : constant Interface_State;
Idle             : constant Interface_State;
Outgoing_Connect : constant Interface_State;
Incoming_Connect : constant Interface_State;
Data_Transfer    : constant Interface_State;
Outgoing_Release : constant Interface_State;
Incoming_Release : constant Interface_State;
-- 17.4.2 Accept a Connection Request
procedure Accept_Connection
   (Listening_Endpoint  : in POSIX_IO.File_Descriptor;
    Responding_Endpoint : in POSIX_IO.File_Descriptor;
    Call                : in Connection_Info);
procedure Accept_Connection
   (Listening_Endpoint  : in POSIX_IO.File_Descriptor;
    Responding_Endpoint : in POSIX_IO.File_Descriptor);
-- 17.4.3 Acknowledge Receipt of an Orderly Release Indication
procedure Acknowledge_Orderly_Release
   (Endpoint : in POSIX_IO.File_Descriptor);
procedure Acknowledge_Orderly_Release
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Reason   : out Reason_Code);
-- 17.4.4 Acknowledge Receipt of an Orderly Release Indication with Data
procedure Acknowledge_Orderly_Release_With_Data
   (Endpoint         : in  POSIX_IO.File_Descriptor;
    Reason           : out Reason_Code;
    User_Data        : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count);
-- 17.4.5 Bind an Address to a Communications Endpoint
procedure Bind
   (Endpoint              : in  POSIX_IO.File_Descriptor;
    Request_Address       : in  XTI_Address_Pointer;
    Request_Queue_Length  : in  Natural;
    Response_Address      : in  XTI_Address_Pointer;
    Response_Queue_Length : out Natural);
procedure Bind
   (Endpoint              : in  POSIX_IO.File_Descriptor;
    Request_Queue_Length  : in  Natural;
    Response_Address      : in  XTI_Address_Pointer;
    Response_Queue_Length : out Natural);
procedure Bind
   (Endpoint             : in POSIX_IO.File_Descriptor;
    Request_Address      : in XTI_Address_Pointer;
    Request_Queue_Length : in Natural);
procedure Bind
   (Endpoint         : in POSIX_IO.File_Descriptor;
    Response_Address : in XTI_Address_Pointer);
```

```
procedure Bind
   (Endpoint : in POSIX_IO.File_Descriptor);
-- 17.4.6 Close a Communications Endpoint
procedure Close
   (Endpoint : in POSIX_IO.File_Descriptor);
-- 17.4.7 Receive the Confirmation from a Connection Request
procedure Confirm_Connection
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Call     : in out Connection_Info);
procedure Confirm_Connection
   (Endpoint : in POSIX_IO.File_Descriptor);
-- 17.4.8 Establish a Connection with Peer
procedure Connect
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Send     : in     Connection_Info;
    Receive  : in out Connection_Info);
procedure Connect
   (Endpoint : in POSIX_IO.File_Descriptor;
    Send     : in Connection_Info);
-- 17.4.9 Gather and send data or expedited data over a connection
procedure Gather_And_Send_Data
   (Endpoint        : in  POSIX_IO.File_Descriptor;
    Vector          : in  IO_Vector_Array;
    Flags           : in  XTI_Flags;
    Octets_Sent     : out POSIX.IO_Count);
-- 17.4.10 Gather and Send a Data Unit
procedure Gather_And_Send_Data_Unit
   (Endpoint        : in POSIX_IO.File_Descriptor;
    Address         : in XTI_Address_Pointer;
    Vector          : in IO_Vector_Array);
procedure Gather_And_Send_Data_Unit
   (Endpoint        : in POSIX_IO.File_Descriptor;
    Address         : in XTI_Address_Pointer;
    Vector          : in IO_Vector_Array;
    Options         : in Protocol_Option_List);
-- 17.4.11 Get the Current State
function Get_Current_State (Endpoint : POSIX_IO.File_Descriptor)
  return Interface_State;
-- 17.4.12 Get Protocol-Specific Service Information
procedure Get_Info
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Info     : out Communications_Provider_Info);
-- 17.4.13 Get the Protocol Address
procedure Get_Protocol_Address
   (Endpoint      : in POSIX_IO.File_Descriptor;
    Bound_Address : in XTI_Address_Pointer;
    Peer_Address  : in XTI_Address_Pointer);
-- 17.4.14 Initiate an Orderly Release
procedure Initiate_Orderly_Release
   (Endpoint : in POSIX_IO.File_Descriptor);
procedure Initiate_Orderly_Release
   (Endpoint : in POSIX_IO.File_Descriptor;
    Reason   : in Reason_Code);
```

```
-- 17.4.15 Initiate an Orderly Release with Application Data
procedure Initiate_Orderly_Release_With_Data
   (Endpoint       : in POSIX_IO.File_Descriptor;
    Reason         : in Reason_Code;
    User_Data      : in System.Address;
    Octets_To_Send : in POSIX.IO_Count);
-- 17.4.16 Listen for a Connection Indication
procedure Listen
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Call     : in out Connection_Info);
-- 17.4.17 Look at the Current Event on a Communication Endpoint
type XTI_Events is new POSIX.Option_Set;
Connect_Request_Received    : constant XTI_Events := implementation-defined;
Connect_Response_Received   : constant XTI_Events := implementation-defined;
Disconnect_Request_Received : constant XTI_Events := implementation-defined;
Error_In_Previously_Sent_Datagram
                            : constant XTI_Events := implementation-defined;
Expedited_Data_Received     : constant XTI_Events := implementation-defined;
Normal_Data_Received        : constant XTI_Events := implementation-defined;
Okay_To_Send_Expedited_Data : constant XTI_Events := implementation-defined;
Okay_To_Send_Normal_Data    : constant XTI_Events := implementation-defined;
Orderly_Release_Request_Received
                            : constant XTI_Events := implementation-defined;
function Look (Endpoint : POSIX_IO.File_Descriptor)
  return XTI_Events;
-- 17.4.18 Manage options for a communication endpoint
XTI_Protocol_Level     : constant Option_Level := implementation-defined;
Unspecified            : constant Option_Value := implementation-defined;
All_Options            : constant Option_Name := implementation-defined;
Enable_Debugging       : constant Option_Name := implementation-defined;
Linger_On_Close_If_Data_Present
                       : constant Option_Name := implementation-defined;
Receive_Buffer_Size    : constant Option_Name := implementation-defined;
Receive_Low_Water_Mark : constant Option_Name := implementation-defined;
Send_Buffer_Size       : constant Option_Name := implementation-defined;
Send_Low_Water_Mark    : constant Option_Name := implementation-defined;
procedure Manage_Options
   (Endpoint       : in     POSIX_IO.File_Descriptor;
    Request        : in     Protocol_Option_List;
    Request_Flag   : in     Options_Flags;
    Response       : in out Protocol_Option_List;
    Response_Flags : out    Option_Status);
-- 17.4.19 Establish a Communication Endpoint
procedure Open
   (Endpoint : out    POSIX_IO.File_Descriptor;
    Name     : in     POSIX.POSIX_String;
    Mode     : in     POSIX_IO.File_Mode;
    Options  : in     POSIX_IO.Open_Option_Set;
    Info     : in out Communications_Provider_Info);
procedure Open
   (Endpoint : out POSIX_IO.File_Descriptor;
    Name     : in  POSIX.POSIX_String;
    Mode     : in  POSIX_IO.File_Mode;
    Options  : in  POSIX_IO.Open_Option_Set);
```

```
-- 17.4.20 Receive Data or Expedited Data Sent Over a Connection
procedure Receive
   (Endpoint         : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Flags            : out XTI_Flags);
-- 17.4.21 Receive and Scatter Data or Expedited Data Sent Over a Connection
procedure Receive_And_Scatter_Data
   (Endpoint          : in  POSIX_IO.File_Descriptor;
    Vector            : in  IO_Vector_Array;
    Octets_Received   : out POSIX.IO_Count;
    Flags             : out XTI_Flags);
-- 17.4.22 Receive and Scatter a Data Unit
procedure Receive_And_Scatter_Data_Unit
   (Endpoint          : in     POSIX_IO.File_Descriptor;
    Address           : in     XTI_Address_Pointer;
    Options           : in out Protocol_Option_List;
    Vector            : in     IO_Vector_Array;
    Octets_Received   : out POSIX.IO_Count;
    Flags             : out XTI_Flags);
-- 17.4.23 Receive a Data Unit
procedure Receive_Data_Unit
   (Endpoint         : in     POSIX_IO.File_Descriptor;
    User_Data        : in     System.Address;
    Octets_Requested : in     POSIX.IO_Count;
    Octets_Received  : out    POSIX.IO_Count;
    Address          : in     XTI_Address_Pointer;
    Options          : in out Protocol_Option_List;
    Flags            : out XTI_Flags);
procedure Receive_Data_Unit
   (Endpoint         : in  POSIX_IO.File_Descriptor;
    User_Data        : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Address          : in  XTI_Address_Pointer;
    Flags            : out XTI_Flags);
-- 17.4.24 Retrieve a Unit Data Error Indication
type Unit_Data_Error_Code is implementation-defined-integer;
procedure Retrieve_Data_Unit_Error
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Address  : in     XTI_Address_Pointer;
    Options  : in out Protocol_Option_List;
    Error    : out Unit_Data_Error_Code);
procedure Retrieve_Data_Unit_Error
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Address  : in  XTI_Address_Pointer;
    Error    : out Unit_Data_Error_Code);
procedure Retrieve_Data_Unit_Error
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Options  : in out Protocol_Option_List;
    Error    : out    Unit_Data_Error_Code);
procedure Retrieve_Data_Unit_Error
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Error    : out Unit_Data_Error_Code);
```

```
   -- 17.4.25 Retrieve Information from Disconnect
   procedure Retrieve_Disconnect_Info
      (Endpoint         : in  POSIX_IO.File_Descriptor;
       User_Data        : in  System.Address;
       Octets_Requested : in  POSIX.IO_Count;
       Octets_Retrieved : out POSIX.IO_Count;
       Reason           : out Reason_Code;
       Sequence_Number  : out Natural);
   procedure Clear_Disconnect_Info
      (Endpoint : in POSIX_IO.File_Descriptor);
   -- 17.4.26 Send Data or Expedited Data Over a Connection
   procedure Send
      (Endpoint       : in  POSIX_IO.File_Descriptor;
       Buffer         : in  System.Address;
       Octets_To_Send : in  POSIX.IO_Count;
       Flags          : in  XTI_Flags;
       Octets_Sent    : out POSIX.IO_Count);
   -- 17.4.27 Send a Data Unit
   procedure Send_Data_Unit
      (Endpoint       : in POSIX_IO.File_Descriptor;
       User_Data      : in System.Address;
       Octets_To_Send : in POSIX.IO_Count;
       Address        : in XTI_Address_Pointer;
       Options        : in Protocol_Option_List);
   procedure Send_Data_Unit
      (Endpoint       : in POSIX_IO.File_Descriptor;
       User_Data      : in System.Address;
       Octets_To_Send : in POSIX.IO_Count;
       Address        : in XTI_Address_Pointer);
   -- 17.4.28 Send Application-Initiated Disconnection Request
   procedure Send_Disconnect_Request
      (Endpoint : in POSIX_IO.File_Descriptor;
       Call     : in Connection_Info);
   procedure Send_Disconnect_Request
      (Endpoint : in POSIX_IO.File_Descriptor);
   procedure Send_Disconnect_Request
      (Endpoint        : in POSIX_IO.File_Descriptor;
       Sequence_Number : in Natural);
   procedure Send_Disconnect_Request
      (Endpoint       : in POSIX_IO.File_Descriptor;
       User_Data      : in System.Address;
       Octets_To_Send : in POSIX.IO_Count);
   -- 17.4.29 Synchronize Communications Endpoint
   function Synchronize_Endpoint
      (Endpoint : in POSIX_IO.File_Descriptor)
     return Interface_State;
   -- 17.4.30 Disable a Communications Endpoint
   procedure Unbind
      (Endpoint : in POSIX_IO.File_Descriptor);

private
implementation-defined
end POSIX_XTI;
```

17.4.1 Common Data Types and Constants

17.4.1.1 Flags

17.4.1.1.1 Synopsis

```
type XTI_Flags is new POSIX.Option_Set;
Expedited_Data      : constant XTI_Flags := implementation-defined;
More_Data           : constant XTI_Flags := implementation-defined;
Push_Data           : constant XTI_Flags := implementation-defined;
type Options_Flags is private;
Check_Options       : constant Options_Flags;
Get_Current_Options : constant Options_Flags;
Get_Default_Options : constant Options_Flags;
Negotiate_Options   : constant Options_Flags;
```

17.4.1.1.2 Description

The type `XTI_Flags` shall denote a set of XTI flags. The operations `"+"`, `"-"`, `">"`, `"<"`, `">="`, `"<="`, and `Empty_Set` are available on the type `XTI_Flags` via the derived type semantics of Ada, from the operations available for `POSIX.Option_Set`. The appropriate operations can be used to create and examine a set containing the required flags. The XTI flags are used to influence the action of various function and procedure calls. Specific usage is detailed in each operation and procedure described in this subclause.

The type `Options_Flags` is used with the `Manage_Options` procedure to indicate what action should be taken by the call.

17.4.1.2 Protocol-Specific Service Limits

17.4.1.2.1 Synopsis

```
type Communications_Provider_Info is private;
type CP_Flags is new POSIX.Option_Set;
Orderly_Release_Data_Supported : constant CP_Flags := implementation-defined;
Zero_Length_SDU_Supported      : constant CP_Flags := implementation-defined;
type Service_Type is private;
Connection_Mode                     : constant Service_Type;
Connection_Mode_With_Orderly_Release : constant Service_Type;
Connectionless_Mode                 : constant Service_Type;
function Protocol_Addresses_Are_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Protocol_Address
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Protocol_Options_Are_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Protocol_Options
   (Info_Item : Communications_Provider_Info)
  return Positive;
function SDU_Is_Supported
   (Info_Item : Communications_Provider_Info)
  return Boolean;
```

```
function SDU_Is_Infinite
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function SDU_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_SDU
   (Info_Item : Communications_Provider_Info)
  return Positive;
function SEDU_Is_Supported
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function SEDU_Is_Infinite
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function SEDU_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_SEDU
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Connect_Data_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Connect_Data
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Disconnect_Data_Is_Valid
   (Info_Item : Communications_Provider_Info)
  return Boolean;
function Get_Max_Size_Disconnect_Data
   (Info_Item : Communications_Provider_Info)
  return Positive;
function Get_CP_Flags
   (Info_Item : Communications_Provider_Info)
  return CP_Flags;
function Get_Service_Type
   (Info_Item : Communications_Provider_Info)
  return Service_Type;
```

17.4.1.2.2 Description

`Communications_Provider_Info` objects are used to identify protocol-specific service limits of the communications provider. Objects of type `Communications_Provider_Info` have (at least) the following attributes:

Max Size Protocol Address

The maximum size in octets of a communications protocol address. The `Protocol_Addresses_Are_Valid` function shall return `False` if the communications provider does not provide application access to communications protocol addresses. The `Get_Max_Size_Protocol_Address` function shall return the value of this attribute.

Max Size Protocol Options

The maximum number of octets of protocol-specific options supported by the provider. The `Protocol_Options_Are_Valid` function shall return `False` if the communications provider does not support application-settable options. The `Get_Max_Size_Protocol_Options` function shall return the value of this attribute.

Max Size SDU

The maximum size in octets of a SDU. The `Get_Max_Size_SDU` function shall return the value of this attribute. The `SDU_Is_Supported` function shall return `False` if the communications provider does not support the concept of SDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection. The `SDU_Is_Infinite` function shall return `True` if there is no limit on the size of an SDU. The `SDU_Is_Valid` function shall return `False` if the transfer of normal data is not supported by the communications provider.

Max Size SEDU

The maximum size in octets of a SEDU. The `Get_Max_Size_SEDU` function shall return the value of this attribute. The `SEDU_Is_Supported` function shall return `False` if the communications provider does not support the concept of an SEDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection. The `SEDU_Is_Infinite` function shall return `True` if there is no limit on the size of an SEDU. The `SEDU_Is_Valid` function shall return `False` if the transfer of expedited data is not supported by the communications provider. (The semantics of expedited data may be quite different for different communications providers (see section D.2).)

Max Size Connect Data

The maximum number of octets of data that may be associated with connection establishment operations. The `Connect_Data_Is_Valid` function shall return `False` if the communications provider does not allow data to be sent with connection establishment operations. The `Get_Max_Size_Connect_Data` function shall return the value of this attribute.

Max Size Disconnect Data

The `Disconnect_Data_Is_Valid` function shall return `False` if the communications provider does not allow data to be sent with the abortive release operations. If this function returns `True`, the `Get_Max_Size_Disconnect_Data` function shall return the value of this attribute. If the `Orderly_Release_Data_Supported` flag is clear, this value specifies the maximum number of octets of data that may be associated with the `Send_Disconnect_Request` and `Retrieve_Disconnect_Info` operations. If the `Orderly_Release_Data_Supported` flag is set, this value specifies the maximum number of octets that may be associated with the `Acknowledge_Orderly_Release_With_Data`, `Initiate_Orderly_Release_With_Data`, `Send_Disconnect_Request`, and `Retrieve_Disconnect_Info` operations.

Service Type

The service type supported by the communications provider. The `Get_Service_Type` function shall return the value of this attribute as follows:

`Connection_Mode`

The communications provider supports a connection mode service, but does not support the optional orderly release facility.

`Connection_Mode_With_Orderly_Release`

The communications provider supports a connection mode service with the optional orderly release facility.

Connectionless_Mode

The communications provider supports a connectionless mode service. For this service type, Open shall set the Max Size SEDU, Max Size Connect Data, and Max Size Disconnect Data attributes of the Communications_Provider_Info so that Connect_Data_Is_Valid, Disconnect_Data_Is_Valid, and SDU_Is_Valid all return False.

CP Flags

This is an option set used to specify other information about the communications provider. If the Orderly_Release_Data_Supported flag is set, the communications provider supports application data to be sent with an orderly release. If the Zero_Length_SDU_Supported flag is set, the underlying communications provider supports the sending of zero-length SDUs. See D.2 for a discussion of the separate issue of zero-length fragments within an SDU.

If an application is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers need to be to hold each piece of information. An error shall result if an application exceeds the allowed data size on any operation. The value of each field may change as a result of protocol option negotiation during connection establishment. (The Manage_Options operation has no effect on the values returned by Get_Info.) These values shall only change from the values presented to Open after the endpoint enters the Data Transfer state.

The Get_CP_Flags function shall return the value of the CP Flags attribute as type CP_Flags. The operations "+", "-", ">", "<", ">=", "<=", and Empty_Set are available on the type CP_Flags via the derived type semantics of Ada, from the operations available for POSIX.Option_Set. The appropriate operations can be used to create and examine a set containing the required flags.

17.4.1.2.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Implemented

The operation is not supported by this implementation.

17.4.1.3 XTI Addresses

17.4.1.3.1 Synopsis

```
type XTI_Address_Pointer is private;
Null_XTI_Address : constant XTI_Address_Pointer;
```

17.4.1.3.2 Description

All network addresses are described using a general object called XTI_Address_Pointer, which references a protocol-specific network address. Each protocol imposes finer and more specific structure, generally defining a new object with attributes specific to the protocol. The protocol-specific attributes of these objects, the

operations that manipulate them, and operations to convert access values that designate these protocol-specific objects to and from the `XTI_Address_Pointer` type are described in D.2.

The constant `Null_XTI_Address` is a special value that does not refer to any address (*i.e.*, the null address). This value is used when an `XTI_Address_Pointer` object can optionally be omitted by the application.

17.4.1.4 Linger Information Objects

17.4.1.4.1 Synopsis

```
type Linger_Info is private;
subtype Linger_Time is POSIX.Seconds range 1 .. POSIX.Seconds'Last;
type Linger_Option is (Linger_Off, Linger_On);
function Get_Status (Item : Linger_Info)
  return Linger_Option;
procedure Set_Status
   (Item   : in out Linger_Info;
    Linger : in     Linger_Option);
function Period_Is_Infinite (Item : Linger_Info)
  return Boolean;
function Period_Is_Unspecified (Item : Linger_Info)
  return Boolean;
function Get_Period (Item : Linger_Info)
  return Linger_Time;
procedure Set_Period_Infinite
   (Item : in out Linger_Info);
procedure Set_Period_Unspecified
   (Item : in out Linger_Info);
procedure Set_Period
   (Item : in out Linger_Info;
    Time : in     Linger_Time);
```

17.4.1.4.2 Description

`Linger_Info` objects are used to identify the time to wait before completing the execution of a `Close` procedure when the send buffer of an endpoint still has data that have not been sent. Objects of type `Linger_Info` have (at least) the following attributes:

Linger Status

Whether the linger option is to be activated or disabled.

Linger Period

The length of time to linger in seconds.

The `Linger_Info` object is used with the `Protocol_Option` object and ultimately passed to the `Manage_Options` procedure through the `Protocol_Option_List` object.

The `Get_Status` function shall return the value of the Linger Status attribute, which indicates whether the linger option is set for this object. The `Set_Status` procedure shall set the Linger Status attribute to the `Linger` parameter.

The Period_Is_Infinite function shall return True if the time to wait is infinite. The Set_Period_Infinite procedure shall set the attribute to infinite. The Period_Is_Unspecified function shall return True if the time to wait is currently set to the communications provider default. The Set_Period_Unspecified procedure shall set the attribute to the communications provider default.

The Get_Period function shall return the value of the Linger Period attribute (in seconds). The Set__Period procedure shall set the Linger Period attribute to the value of the Time parameter (in seconds). The value returned by the Get_Period function following a Set_Period_Infinite operation is undefined.

17.4.1.4.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Implemented

The operation is not supported by this implementation.

17.4.1.5 Protocol Option Objects

17.4.1.5.1 Synopsis

```
type Protocol_Option is private;
type Option_Value is range implementation-defined;
type Option_Level is range implementation-defined;
type Option_Name is range implementation-defined;
type Option_Status is private;
Success         : constant Option_Status := implementation-defined;
Partial_Success : constant Option_Status := implementation-defined;
Failure         : constant Option_Status := implementation-defined;
Read_Only       : constant Option_Status := implementation-defined;
Not_Supported   : constant Option_Status := implementation-defined;
type Option_Value_Array is array (Positive range <>) of Option_Value;
function Get_Level (Option_Item : Protocol_Option)
  return Option_Level;
function Get_Name (Option_Item : Protocol_Option)
  return Option_Name;
function Get_Status (Option_Item : Protocol_Option)
  return Option_Status;
function Get_Value (Option_Item : Protocol_Option)
  return Option_Value;
function Get_Value (Option_Item : Protocol_Option)
  return Option_Value_Array;
function Get_Value (Option_Item : Protocol_Option)
  return Linger_Info;
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name);
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name;
    Value       : in      Option_Value);
```

```
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name;
    Value       : in      Option_Value_Array);
procedure Set_Option
   (Option_Item : in out  Protocol_Option;
    Level       : in      Option_Level;
    Name        : in      Option_Name;
    Value       : in      Linger_Info);
```

17.4.1.5.2 Description

The Protocol_Option object is used to identify options, and may also include values for those options, at either the XTI interface level or the protocol level of the communication provider. Objects of type Protocol_Option have (at least) the following attributes:

Level

The protocol level to which this option object is related.

Name

The name of the option to be affected.

Status

The status value for this particular option object. This attribute only has meaning when set by the implementation.

Value

The value of this option. This attribute may not be valid for some options (*i.e.*, some options do not have values associated with them).

The Protocol_Option object is used along with the Protocol_Option_List object to either get or set options through the Manage_Options procedure. The Protocol_Option_List object is used along with the Connection_Info object to either return or set options during the execution of the Accept_Connection, Connect, and Listen procedures. The Protocol_Option_List object is used to either return options during the execution of the Receive_Data_Unit procedure or set options during the execution of the Send_Data_Unit procedure. The Protocol_Option_List object is used to return options during the execution of the Retrieve_Data_Unit_Error procedure.

The Get_Level function shall return the value of the Level attribute, which indicates the protocol level of this option.

The Get_Name function shall return the value of the Name attribute, which indicates the name of the option.

The Get_Status function shall return the value of the Status attribute, which indicates the present status of this option. The Status attribute shall only be meaningful after a successful call to Accept_Connection, Connect, Listen, Manage_Options, Receive_Data_Unit, or Retrieve_Data_Unit_Error. The Status attribute shall be set by the implementation and may be any of the following: Success, Partial_Success, Failure, Read_Only, or Not_Supported.

The Get_Value function shall return the Value attribute, which indicates the value of this option. The Get_Value function is overloaded to accommodate different types of the Value parameter. that exist. Each option, identified by the Level and Name attributes, has only one Value type associated with it. Use of the wrong Value type for a particular option shall result in an error.

The Set_Option procedure shall set the Level attribute of the Option_Item object to the Level parameter, the Name attribute of Option_Item object to the Name parameter, and the Value attribute of the Option_Item object to the Value parameter (if one is provided). The Set_Option procedure is overloaded to accommodate the different Value types that may be associated with each option. Some options do not have any Value associated with them.

17.4.1.5.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Implemented

The operation is not supported by this implementation.

Operation_Not_Permitted

The Value type used for a particular option during a Get_Value function call or Set_Option procedure call is not permitted.

17.4.1.6 Protocol Option List Objects

17.4.1.6.1 Synopsis

```
type Protocol_Option_List is private;
type Protocol_Option_List_Pointer is
  access all Protocol_Option_List;
procedure Make_Empty
   (Info_Item : in out Protocol_Option_List);
procedure Append
   (Info_Item : in out Protocol_Option_List;
    Option    : in     Protocol_Option);
type Octet_Buffer_Pointer is access all POSIX.Octet_Array;
procedure Set_Buffer
   (Info_Item      : in out Protocol_Option_List;
    Options_Buffer : in     Octet_Buffer_Pointer);
generic
   with procedure Action
     (Info : in     Protocol_Option;
      Quit : in out Boolean);
procedure For_Every_Item (Info_Item : in Protocol_Option_List);
function Number_Of_Options (Info_Item : Protocol_Option_List)
  return Natural;
procedure Get_Option
   (Info_Item     : in  Protocol_Option_List;
    Option_Number : in  Positive;
    Option        : out Protocol_Option);
```

17.4.1.6.2 Description

The Protocol_Option_List object is used to identify options. The Protocol_Option_List object is used in the Manage_Options, Gather_And_Send_Data_Unit, Receive_Data_Unit, and Retrieve_Data_Unit_Error operations to convey or obtain options to/from the interface. Additionally, the Protocol_Option_List object is used with the Connection_Info object to convey or obtain options to/from the interface. Objects of type Protocol_Option_List have (at least) the following attribute:

Options Buffer

A pointer to a buffer where data are to be stored or are stored. The value is required to designate an object of type Octet_Buffer_Pointer. Its initial value is null.

The application is responsible for the allocation and deallocation of I/O buffers. The effect of deallocating or modifying the contents of an I/O buffer before the I/O operation has been completed is undefined.

Each Protocol_Option_List object shall identify an options buffer after a successful Set_Buffer operation has been performed. The Set_Buffer procedure shall set the Options Buffer attribute to point to the buffer identified by the Options_Buffer parameter. This buffer provides storage for the Protocol_Option_List. A Set_Buffer operation shall result in an empty list; that is, any data previously contained in the Octet_Buffer are ignored.

The Append procedure shall add an option to the end of the list of options represented by the Options Buffer attribute of the Info_Item parameter. It also adds one to the current count of options in the list. The number of options in the list of options represented by the Options Buffer attribute of the Info_Item shall be returned by the Number_Of_Options function. If an attempt is made to appended to a Protocol_Option_List an option that already exists in the list, the behavior of the procedure is implementation defined (see 17.3.2.3).

The Make_Empty procedure shall remove all options from the Options Buffer attribute of the Info_Item parameter, freeing any dynamically allocated storage associated with the object.

The application program instantiates the generic procedure For_Every_Item with an actual procedure for the generic formal procedure Action. When called, the instance shall call the actual procedure supplied for Action once for each element in the associated list.

Action shall be able to force termination of the generic instance either by setting Quit to True or by raising an exception. Prior to calling Action, the instance shall set Quit to False. Exceptions raised by Action shall terminate iteration and shall be propagated back to the caller of the instance. After an exception is raised by Action or Action returns with Quit set to True, no more calls to Action shall occur.

The Get_Option procedure shall fill the Options parameter with the option specified by the Option_Number parameter in the list of options represented by the Options Buffer attribute of the Info_Item parameter.

17.4.1.6.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Operation_Not_Implemented`

The operation is not supported by this implementation.

`Invalid_Argument`

The value of `Options_Buffer` is not a valid `Octet_Buffer_Pointer`. For this error, the implementation under some circumstances may instead raise `Constraint_Error`.

The value of the `Option_Number` parameter in the `Get_Option` procedure is out of range. For this error, the implementation under some circumstances may instead raise `Constraint_Error`.

`Buffer_Not_Large_Enough`

An attempt to perform an `Append` operation on a `Protocol_Option_List` object cannot be performed because the buffer represented by the Options Buffer attribute is not large enough or has a value of `null`.

17.4.1.7 Disconnect Reason Codes

17.4.1.7.1 Synopsis

```
type Reason_Code is range implementation-defined;
```

17.4.1.7.2 Description

The `Reason_Code` type is used by the `Acknowledge_Orderly_Release`, `Acknowledge_Orderly_Release_With_Data`, `Initiate_Orderly_Release`, `Initiate_Orderly_Release_With_Data`, and `Retrieve_Disconnect_Info` procedures to specify a a protocol-specific reason code associated with disconnections.

17.4.1.8 Connection Information Objects

17.4.1.8.1 Synopsis

```
type Connection_Info is limited private;
procedure Set_Address
   (Info_Item  : in out Connection_Info;
    Address    : in     XTI_Address_Pointer);
function Get_Options (Info_Item : Connection_Info)
  return Protocol_Option_List;
procedure Set_Options
   (Info_Item : in out Connection_Info;
    Options   : in     Protocol_Option_List_Pointer);
procedure Set_User_Data
   (Info_Item  : in out Connection_Info;
    User_Data  : in     System.Address;
    Max_Length : in     POSIX.IO_Count);
procedure Set_User_Data_Length
   (Info_Item  : in out Connection_Info;
    Length     : in     POSIX.IO_Count);
```

```
function Get_User_Data_Length
   (Info_Item : Connection_Info)
  return POSIX.IO_Count;
function Get_Sequence_Number (Info_Item : Connection_Info)
  return Integer;
procedure Set_Sequence_Number
   (Info_Item : in out Connection_Info;
    Number    : in     Integer);
```

17.4.1.8.2 Description

The Connection_Info object is used to provide information about the protocol address, options, application data, and the sequence number for a connection. The Connection_Info object is used in the Accept_Connection, Confirm_Connection, Connect, Listen, and Send_Disconnect_Request procedures. Objects of type Connection_Info have (at least) the following attributes:

Address

An XTI_Address_Pointer value, designating an object containing an address.

Options

A Protocol_Option_List object, representing options.

User Data

A pointer to the application data.

Sequence

A sequence number.

The Address attribute shall contain a protocol-specific address. The Address attribute shall be set by the Set_Address procedure and shall be obtained by the Get_Address procedure. The Get_Address procedure is protocol-specific (see D.2).

The Options attribute may contain options. The Options attribute shall be set by the Set_Options procedure and shall be obtained by the Get_Options function. Set_Options establishes a reference to a Protocol_Option_List object so that any subsequent changes made to the Protocol_Option_List are propagated back to the Connection_Info object.

The User Data attribute may contain application data that are to be sent or were sent along with the request. The User Data attribute shall be set by the Set_User_Data procedure and shall be obtained by the Get_User_Data function. User Data shall include both a Max_Length and a Length component (as parameters to Set_User_Data). Max_Length represents the size of the User Data buffer and Length represents the length of usable data sent or received.

The Sequence Number attribute may identify an outstanding connect indication with which the particular operation is associated. The Sequence Number attribute shall be set by the Set_Sequence_Number procedure and shall be obtained by the Get_Sequence_Number function.

Upon declaration of the Connection_Info object, the User Data attribute shall be set to System.Null_Address and its Length and Max Length attributes shall be set to

zero. The Options attribute shall be set to the null access value and shall represent an empty protocol option list. The Address attribute shall be set to the Null_XTI_-Address constant.

17.4.1.8.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

The Length parameter was greater than the Max_Length specified for the Connection_Info object.

Operation_Not_Implemented

The operation is not supported by this implementation.

17.4.1.9 Scatter/Gather Vector Objects

17.4.1.9.1 Synopsis

```
subtype IO_Vector_Range is Positive range
   1 .. POSIX_Limits.XTI_IO_Vector_Maxima'Last;
type IO_Vector_Array is array
   (IO_Vector_Range range <>) of POSIX_IO.IO_Vector;
```

17.4.1.9.2 Description

The IO_Vector_Array object is used by Receive_And_Scatter_Data, Receive_-And_Scatter_Data_Unit, Gather_And_Send_Data, and Gather_And_Send_-Data_Unit to allow multiple data buffers to be received or sent in a single operation. Each element in the IO_Vector_Array is a POSIX_IO.IO_Vector with at least the following attributes:

Buffer

The address of the buffer into which the data are to be read, or from which the data are to be written.

NOTE: Buffer is a reference to storage that may persist beyond an I/O operation. The effect of deallocating or modifying the contents of the buffers in an I/O vector before the I/O operation has completed is undefined.

NOTE: Applications may use pointers to the POSIX.Octet_Array type to ensure proper data width for network I/O operations. When using pointers to other data types, byte width and ordering issues (i.e., big endian, little endian) are the responsibility of the application.

Length

The length of the data to be transferred or received, as a count of octets.

POSIX_Limits.XTI_IO_Vector_Maxima'Last shall indicate the maximum number of POSIX_IO.IO_Vector objects that can be contained in the IO_Vector_Array.

Each POSIX_IO.IO_Vector shall identify a data buffer after a successful POSIX_-IO.Set_Buffer operation has been performed. POSIX_IO.Set_Buffer shall set the Buffer attribute of the POSIX_IO.IO_Vector to point to the storage indicated

by the Buffer parameter and the Length attribute of the POSIX_IO.IO_Vector to Length (in octets). Any particular POSIX_IO.IO_Vector element shall be reset to indicate no data by either a Length of zero or the Buffer parameter set to System.Null_Address. The Length attribute may be set to less than the size of the storage allocated for Buffer. The POSIX_IO.Get_Buffer procedure shall return the Buffer and Length attributes of the POSIX_IO.IO_Vector.

17.4.1.9.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Operation_Not_Implemented
: The operation is not supported by this implementation.

Invalid_Argument
: The Buffer parameter is not a valid System.Address. For this error, the implementation under some circumstances may instead raise Constraint_Error.

17.4.1.10 Communications Interface States

17.4.1.10.1 Synopsis

```
type Interface_State is private;
Uninitialized    : constant Interface_State;
Unbound          : constant Interface_State;
Idle             : constant Interface_State;
Outgoing_Connect : constant Interface_State;
Incoming_Connect : constant Interface_State;
Data_Transfer    : constant Interface_State;
Outgoing_Release : constant Interface_State;
Incoming_Release : constant Interface_State;
```

17.4.1.10.2 Description

These symbols shall represent the states described in Table 17.7interface states of 17.2.1. The function Get_Current_State returns the current state of an endpoint (see 17.4.11).

17.4.2 Accept a Connection Request

17.4.2.1 Synopsis

```
procedure Accept_Connection
   (Listening_Endpoint  : in POSIX_IO.File_Descriptor;
    Responding_Endpoint : in POSIX_IO.File_Descriptor;
    Call                : in Connection_Info);
procedure Accept_Connection
   (Listening_Endpoint  : in POSIX_IO.File_Descriptor;
    Responding_Endpoint : in POSIX_IO.File_Descriptor);
```

17.4.2.2 Description

The procedure Accept_Connection is used to accept a connection request indicated by the Listen procedure. Listening_Endpoint specifies the local communications endpoint where the connection indication arrived. Responding_Endpoint specifies the local communications endpoint where the connection is to be established. The Call parameter is a Connection_Info object specifying connection information for the communications provider.

The Address attribute of Call is an XTI_Address_Pointer object that designates an object containing the protocol address of the calling application, *i.e.*, the local address. Address may be set to the constant Null_XTI_Address. If not set to Null_XTI_-Address, it may optionally be checked by XTI.

The Options attribute of Call is a Protocol_Option_List object that indicates any protocol-specific options associated with the connection.

The User Data attribute of Call specifies application data to be returned to the remote peer. The amount of application data shall not exceed the limits supported by the communications provider as returned in the Max Size Connect Data attribute of the Communications_Provider_Info parameter of Open or Get_Info. If the length of the User Data attribute is zero, no data shall be returned to the remote peer.

The Sequence attribute of Call is the unique identifier for the connection that was previously returned by a call to Listen.

An application may establish a connection on either the same, or on a different, local communications endpoint than the one on which the connection indication arrived. Before the connection can be accepted on the same endpoint (Listening_Endpoint equals Responding_Endpoint), the application shall have responded to any previous connection indications received on that communications endpoint (via Accept_-Connection or Send_Disconnect_Request). Otherwise, Accept_Connection shall fail with the error Outstanding_Connection_Indications. If the application accepts the connection on the listening endpoint, then the endpoint is said to be bound to a protocol address and the application cannot listen for further connections on that communications endpoint until after an Unbind on the protocol address is issued. (In this case, it follows that Request_Queue_Length parameter submitted to the Bind procedure was greater than zero.)

If a different communications endpoint is specified (Listening_Endpoint does not equal Responding_Endpoint), then the application can bind the responding endpoint before the Accept_Connection is issued. If the application does not bind the endpoint, the endpoint shall be in the Unbound state before the Accept_Connection is issued, and the communications provider shall automatically bind it to an address that is appropriate for the protocol concerned (see D.2). If the application does bind the endpoint, it shall supply a Request_Queue_Length of zero to Bind or issue a Bind procedure call without a Request_Queue_Length. The responding endpoint shall then be in the Idle state when Accept_Connection is called. See also 17.4.5.

The call to Accept_Connection shall fail with the error Event_Requires_Attention if indications (*e.g.*, connection or disconnection) are outstanding on Listening_Endpoint.

When the application does not indicate any option in the Options attribute of `Call`, the connection shall be accepted with the option values currently set for the `Responding_Endpoint` (see 17.3.2.4 and 17.3.3).

There may be communications provider-specific restrictions on address binding. See D.2.

Some communications providers do not differentiate between a connection indication and the connection itself. If the connection has already been established after a successful return from `Listen`, `Accept_Connection` shall assign the existing connection to the communications endpoint specified by `Responding_Endpoint` (see D.2).

17.4.2.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Insufficient_Permission`

The application does not have the permission either to accept the connection on the responding communications endpoint or to use a specified option.

`Incorrect_Address_Format`

The address specified by the application contained an incorrect protocol address, or the protocol address was in an incorrect format.

`Illegal_Data_Range`

The application attempted to send an illegal amount of application data.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified in either `Listening_Endpoint` or `Responding_Endpoint`, or the application is illegally accepting a connection on the same communications endpoint on which the connection indication arrived.

`Incorrect_Or_Illegal_Option`

The option specified by the application contained incorrect information, or the information was in an incorrect format.

`Invalid_Sequence_Number`

The application specified an incorrect sequence number.

Outstanding_Connection_Indications

Connection indications are outstanding on the endpoint when the application called Accept_Connection with Listening_Endpoint equal to Responding_Endpoint. (To avoid this error an application should accept the other connections on a different endpoint by using Accept_Connection, or reject them with Send_Disconnect_Request.)

Event_Requires_Attention

An asynchronous event is outstanding on the communications endpoint specified by Listening_Endpoint. (Applications can call Look to retrieve the event.)

XTI_Operation_Not_Supported

The communications provider does not support this operation.

Operation_Not_Valid_For_State

The procedure was called with the communications provider (specified either by Listening_Endpoint or Responding_Endpoint) in the wrong state.

Protocol_Error

A communication problem has occurred, and there is no other appropriate error number.

Communications_Provider_Mismatch

Listening_Endpoint and Responding_Endpoint do not specify the same communications provider.

Incorrect_Surrogate_Queue_Length

An attempt was made to accept a connection on Responding_Endpoint (where Responding_Endpoint does not equal Listening_Endpoint) with an endpoint queue length greater than zero.

Surrogate_File_Descriptor_Mismatch

The communications provider only allows Listening_Endpoint and Responding_Endpoint to be bound to the same address.

17.4.3 Acknowledge Receipt of an Orderly Release Indication

17.4.3.1 Synopsis

```
procedure Acknowledge_Orderly_Release
   (Endpoint : in POSIX_IO.File_Descriptor);
procedure Acknowledge_Orderly_Release
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Reason   : out Reason_Code);
```

17.4.3.2 Description

The procedure Acknowledge_Orderly_Release is used to acknowledge receipt of an orderly release indication or confirmation from the communications provider. Endpoint is the file descriptor of the local communications endpoint where the connection exists. The Reason parameter, if used, specifies the reason for the orderly release through a protocol-specific reason code.

After calling this procedure, the application shall not attempt to receive more data via `Receive` or `Receive_And_Scatter_Data`. Such an attempt shall fail with the error `Operation_Not_Valid_For_State`. After the orderly release indication is received, the Application may continue to send data over the connection but only as long as `Initiate_Orderly_Release` has not been called by the application.

This procedure is one of the optional services of the communications provider and is supported only if the `Open` or `Get_Info` call returned service type of `Connection_-Mode_With_Orderly_Release`.

This procedure shallt be used from the Data Transfer state or the Outgoing Release state.

Any application data that may be associated with the orderly release indication shall be discarded when `Acknowledge_Orderly_Release` is called.

17.4.3.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Event_Requires_Attention`
: An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`No_Orderly_Release_Indication_On_Endpoint`
: No orderly release indication is outstanding on the communications endpoint specified by `Endpoint`.

`XTI_Operation_Not_Supported`
: This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`
: The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

17.4.4 Acknowledge Receipt of an Orderly Release Indication with Data

17.4.4.1 Synopsis

```
procedure Acknowledge_Orderly_Release_With_Data
   (Endpoint         : in  POSIX_IO.File_Descriptor;
    Reason           : out Reason_Code;
    User_Data        : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count);
```

17.4.4.2 Description

The Acknowledge_Orderly_Release_With_Data procedure is used to acknowledge receipt of an orderly release indication or confirmation and to retrieve any application data sent with the release. The Endpoint parameter identifies the local communications endpoint where the connection exists. The amount of data requested is specified in the Octets_Requested parameter. Octets_Received returns the number of octets of data returned on success.

After receipt of an orderly release indication, the application shall not attempt to receive more data via Receive or Receive_And_Scatter_Data. Such an attempt shall fail with error code Operation_Not_Valid_For_State. However, the application can continue to send data over the connection if it has not called Initiate_Orderly_Release or Initiate_Orderly_Release_With_Data.

The Reason parameter specifies the reason for the orderly release through a protocol-specific reason code, and the User_Data parameter identifies any application data that were received with the orderly release indication.

If User_Data is set to System.Null_Address, no value shall be returned. If User_Data points to a buffer that is less than the length of the application data, Acknowledge_Orderly_Release_With_Data shall fail with the error code Buffer_Not_Large_Enough.

This operation is an optional service of the communications provider and is only supported if the communications provider returned service type Connection_Mode_With_Orderly_Release on Open or Get_Info.

17.4.4.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_File_Descriptor
: A bad file descriptor was specified for the communications endpoint.

Buffer_Not_Large_Enough
: The amount of data to be returned in one of the buffers is greater than the buffer size specified for the User_Data parameter, and the disconnection information to be returned in User_Data shall be discarded. The state of the endpoint, as seen by the application, shall be changed as if the data were successfully retrieved.

Event_Requires_Attention
: An asynchronous event is outstanding on the communications endpoint specified by Endpoint. (The application can call Look to retrieve the event.)

`No_Orderly_Release_Indication_On_Endpoint`

No orderly release indication is outstanding on the communications endpoint specified by `Endpoint`.

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.5 Bind an Address to a Communications Endpoint

17.4.5.1 Synopsis

```
procedure Bind
   (Endpoint              : in  POSIX_IO.File_Descriptor;
    Request_Address       : in  XTI_Address_Pointer;
    Request_Queue_Length  : in  Natural;
    Response_Address      : in  XTI_Address_Pointer;
    Response_Queue_Length : out Natural);
procedure Bind
   (Endpoint              : in  POSIX_IO.File_Descriptor;
    Request_Queue_Length  : in  Natural;
    Response_Address      : in  XTI_Address_Pointer;
    Response_Queue_Length : out Natural);
procedure Bind
   (Endpoint             : in POSIX_IO.File_Descriptor;
    Request_Address      : in XTI_Address_Pointer;
    Request_Queue_Length : in Natural);
procedure Bind
   (Endpoint         : in POSIX_IO.File_Descriptor;
    Response_Address : in XTI_Address_Pointer);
procedure Bind
   (Endpoint : in POSIX_IO.File_Descriptor);
```

17.4.5.2 Description

The procedure `Bind` shall associate a protocol address with the communications endpoint specified by `Endpoint` and activates that communications endpoint. In connection mode, this association enables the communications provider to either service connection requests or to enqueue incoming connection indications. In connectionless mode, the application may send or receive data units through the communications endpoint. (The passive application would be able to determine whether the communications provider has accepted a connection indication by issuing a `Listen`. Similarly the active application would be able to issue a `Connect`.)

The parameters `Request_Address` and `Response_Address` are `XTI_Address_Pointer` values. The `Request_Address` parameter specifies the requested protocol address to be bound to the given communications endpoint. The `Response_Address` parameter returns the actual protocol address bound to the communications endpoint.

On return, Response_Address contains an encoding for the address that the communications provider actually bound to the communications endpoint. If an address was specified in Request_Address, then the returned value shall be an encoding of the same address.

Either or both Response_Address and Request_Address may be omitted by using the appropriate overloaded Bind procedure. However, if the Request_Address is omitted, then the communications provider shall support optional automatic generation of addresses.

NOTE: To find out whether the communications provider supports this option, the application may specify a call to Bind without Request_Address. If not supported, the communications provider will generate the error Could_Not_Allocate_Address.

If the requested address is not available, Bind shall raise the appropriate error. If no request address is specified, the communications provider shall assign an appropriate address to be bound and shall return that address in Response_Address if provided. If the communications provider could not allocate an address, Bind shall fail with the error Could_Not_Allocate_Address. Similarly, no response address is specified if the application does not care what address was bound by the provider. The Response_Queue_Length is omitted if the application is not interested in the negotiated response value.

The Request_Queue_Length and Response_Queue_Length parameters have meaning only when initializing a connection mode service. They specify the number of outstanding connection indications that the communications provider should support for the given communications endpoint. The provider may queue more connect indications than Response_Queue_Length specifies, but shall ensure that there are never more than Response_Queue_Length of them delivered to the application than are still outstanding at any given time.

An outstanding connection indication is one that has been passed to the communications application by the communications provider, but that has not been accepted or rejected. A value of Request_Queue_Length greater than zero is only meaningful when issued by a passive application that expects other applications to call it. The value of Request_Queue_Length shall be negotiated by the communications provider and may be changed if the communications provider cannot support the specified number of outstanding connection indications. However, this value of Request_Queue_Length shall never be negotiated from a requested value greater than zero to zero. On return, the Response_Queue_Length parameter shall contain the negotiated value. If Request_Queue_Length parameter is omitted, it is assumed to be zero.

If Endpoint refers to a connection mode service, Bind allows more than one communications endpoint to be bound to the same protocol address (however, for this to work it is necessary for the communications provider to also support this capability), but it is not possible to bind more than one protocol address to the same communications endpoint. If an application binds more than one communications endpoint to the same protocol address, only one endpoint can be used to listen for connection indications associated with that protocol address. In other words, only one Bind for a given protocol address may specify a value of Request_Queue_Length greater than zero. In this way, the communications provider can identify which communications

endpoint should be notified of an incoming connection indication. If an application attempts to bind a protocol address to a second communications endpoint with a value of `Request_Queue_Length` greater than zero, `Bind` shall generate the error `Address_In_Use`. When an application establishes a connection on the communications endpoint that is being used as the listening endpoint (*i.e.*, without using a new `Responding_Endpoint`, see 17.4.2), the bound protocol address shall be found to be busy for the duration of the connection, until an `Unbind` or `Close` call has been issued. No other communications endpoints may be bound for listening on that same protocol address while that initial listening endpoint is active (in the Data Transfer state or in the Idle state); that is, the implementation shall prevent more than one communications endpoint bound to the same protocol address from accepting connection indications.

An implementation need not allow an application explicitly to bind more than one communications endpoint to a single protocol address, while permitting more than one connection to be accepted to the same protocol address. In other words, although an attempt to bind a communications endpoint to an address with `Request_Queue_Length` $= 0$ might be rejected with `Address_In_Use`, the application can nevertheless use this (unbound) endpoint as a responding endpoint in a call to `Accept_Connection`. To become independent of such implementation differences, the application should supply unbound responding endpoints to `Accept_Connection`.

If `Endpoint` refers to a connectionless mode service, only one endpoint may be associated with a protocol address. If an application attempts to bind a second communications endpoint to an already bound protocol address, `Bind` shall generate the error `Address_In_Use`.

This procedure shall be used from the Unbound state.

17.4.5.3 Error Handling

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Incorrect_Address_Format`

The parameters `Request_Address` and `Response_Address` designate objects with incompatible protocol-specific address types.

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Insufficient_Permission`

The application does not have the permission to use the specified address.

`Address_In_Use`

The requested address is in use.

`Incorrect_Address_Format`

The address specified by the application contained an incorrect protocol address or was in an incorrect format.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Buffer_Not_Large_Enough`

The type of the actual `Response_Address` parameter is not appropriate for the kind of address bound. (The size of the address to be returned is greater than the space available.)

`Could_Not_Allocate_Address`

An address could not be allocated by the communications provider.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.6 Close a Communications Endpoint

17.4.6.1 Synopsis

```
procedure Close
   (Endpoint : in POSIX_IO.File_Descriptor);
```

17.4.6.2 Description

The procedure `Close` informs the communications provider that the application is finished with the communications endpoint specified by `Endpoint`, and frees any local resources associated with the endpoint. In addition, `Close` closes the file associated with the communications endpoint.

`Close` should be called from the Unbound state (see 17.4.11). However, this procedure does not check state information, so it may be called from any state to close a communications endpoint. The resources associated with the endpoint shall be freed automatically. In addition, `POSIX_IO.Close` shall be issued for that file descriptor. If no other descriptors in this process or in another process reference the communication endpoint, any connection that may be associated with that endpoint shall be broken. The connection may be terminated in an orderly or abortive manner.

A `Close` issued on a connection endpoint may cause data previously sent, or data not yet received, to be lost. It is the responsibility of the application to ensure that data are received by the remote peer.

This procedure may be used from all states except Uninitialized.

17.4.6.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

17.4.7 Receive the Confirmation from a Connection Request

17.4.7.1 Synopsis

```
procedure Confirm_Connection
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Call     : in out Connection_Info);
procedure Confirm_Connection
   (Endpoint : in POSIX_IO.File_Descriptor);
```

17.4.7.2 Description

The procedure `Confirm_Connection` is used by a calling application to determine whether a previously issued connection request has completed on a given communications endpoint specified by `Endpoint`. It is used in conjunction with `Connect`, either when `POSIX_IO.Non_Blocking` is set, or when `POSIX_IO.Non_Blocking` is clear and the call to `Connect` has been interrupted by a signal. Successful completion of this procedure indicates that a connection has been established.

`Call` is a `Connection_Info` object used to return information associated with the new connection. The Address attribute returns the protocol address associated with the responding communications endpoint. The Options attribute returns protocol specific options associated with the newly established connection. The User Data attribute returns optional application data that may be returned by the destination application during connection establishment. The Sequence attribute is not used.

The Address attribute, the Options Buffer attribute of Options and the Buffer attribute of User Data shall be supplied by the caller to specify the buffers to be used to return options and application data. If the size of the storage referenced by the Address attribute, the Options Buffer attribute of Options or the length of the User Data is zero, no information shall be returned. If the size of the storage referenced by the the Address attribute, the Options Buffer attribute of Options, or the User Data attribute is greater than zero and less than the length of the returned information, `Confirm_Connection` shall fail with the error `Buffer_Not_Large_Enough`.

The application may ignore the data provided in the `Connection_Info` object by omitting the `Call` parameter.

If `POSIX_IO.Non_Blocking` is clear, `Confirm_Connection` waits for the connection to be established before returning. On return, the Address, Options and User Data attributes reflect values associated with the connection.

If `POSIX_IO.Non_Blocking` is set, `Confirm_Connection` reduces to a poll for existing connection confirmations. If none are available, `Confirm_Connection` fails and returns immediately without waiting for the connection to be established. In this case, `Confirm_Connection` needs to be called again to complete the connection establishment phase and retrieve the information returned in `Call`.

This procedure shall be used from the Outgoing Connect state.

The type and number of supported options vary with the communications provider. In addition, the format of the address pointed to by the Address attribute can vary depending on whether the implementation is fully XTI compliant.

NOTE: The communications provider should provide documentation on supported options and the formats of the protocol option lists.

17.4.7.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Buffer_Not_Large_Enough`
: The number of octets allocated for the received information is not sufficient to store the data, and the connection information to be returned in `Call` shall be discarded. The state of the endpoint, as seen by the application, shall be changed to Data Transfer.

`Event_Requires_Attention`
: An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (Applications can call `Look` to retrieve the event.)

`No_Data_Available`
: In nonblocking mode, no connection confirmations have arrived yet.

`XTI_Operation_Not_Supported`
: This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`
: The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.8 Establish a Connection with Peer

17.4.8.1 Synopsis

```
procedure Connect
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Send     : in     Connection_Info;
    Receive  : in out Connection_Info);
procedure Connect
   (Endpoint : in POSIX_IO.File_Descriptor;
    Send     : in Connection_Info);
```

17.4.8.2 Description

The procedure `Connect` enables an application to request a connection to the specified destination. The parameter `Endpoint` identifies the local communications endpoint where the connection is established. The parameter `Send` specifies information needed by the communications provider to establish a connection. The parameter `Receive` specifies information that is associated with the newly established connection.

The `Send` and `Receive` parameters are `Connection_Info` objects containing an Options attribute (a `Protocol_Option_List_Pointer`), a User Data attribute (defined with an address and length pair), an Address attribute (an `XTI_Address_Pointer` value), and a Sequence attribute.

In `Send`, the Address attribute specifies the protocol address of the destination application. The Options attribute presents any protocol-specific information that might be needed by the communications provider. The User Data attribute specifies optional application data that may be passed to the destination application during connection establishment. The Sequence attribute has no meaning for this procedure.

The Options attribute permits applications to define the options that can be passed to the communications provider. These options are specific to the underlying protocol of the communications provider. If the application provides an empty list by using the `Make_Empty` procedure on the Options attribute, then the provider shall use the option values currently set for the communications endpoint (see 17.3.2.3 and 17.3.3)

If used, the Options Buffer attribute of `Options` shall specify a buffer with the corresponding options.

The User Data attribute enables the caller to pass application data to the destination application and receive application data from the destination application during the connection establishment. However, the amount of application data shall not exceed the limits supported by the communications provider as returned in the Max Size Connect Data attribute of the `Communications_Provider_Info` parameter of `Get_Info` or `Open`. If the length of the User Data attribute is zero in `Send`, no data shall be sent to the destination application.

On return, the Address, Options, and User Data attributes of the Receive parameter shall be updated to reflect values associated with the connection. The Address, the Options Buffer attributes of Options, and the Buffer attribute of User Data shall be supplied by the caller to specify the storage to be used to return the address, options, and application data. The Address, Options, and User Data attributes of the Send and Receive parameters may point to the same storage. If connection information was requested with the Receive parameter, and if the size of the storage referenced by the Address or Options Buffer attribute of Options or the User Data attribute is insufficient to store the returned connection information, Connect shall fail with the error Buffer_Not_Large_Enough.

The application may ignore the data provided in the Connection_Info object by omitting the Receive parameter. The application may selectively ignore portions of the data by setting individual attributes of the Connection_Info object. Address may be set to Null_XTI_Addres, User Data may be set to System.Null_Address, and the Options Buffer attribute of Options may be set to the null access value.

By default, Connect waits for a response from the destination application before returning control to the local application. A successful return indicates that the requested connection has been established. However, if POSIX_IO.Non_Blocking is set, a call to Connect shall not wait for the response of the remote application but shall return control immediately to the local application and generate No_Data_Available to indicate that the connection has not yet been established (assuming no other error indications were detected) . In this way, the operation simply initiates the connection establishment procedure by sending a connection request to the destination application. Confirm_Connection can be used in conjunction with Connect to determine the status of the requested connection.

When POSIX_IO.Non_Blocking is clear and a call to Connect is interrupted by a signal, the state of the endpoint shall become Outgoing Connect, allowing a subsequent call to Confirm_Connection, Retrieve_Disconnect_Info or Send_Disconnect_Request. When POSIX_IO.Non_Blocking is set and a call to Connect is interrupted by a signal, the endpoint shall remain in the Idle state.

The type and number of supported options vary with the communications provider. In addition, the format of the address pointed to by the Address attribute can vary depending on whether the implementation is fully XTI compliant. See communications provider documentation for information on supported options and the formats of the protocol option lists.

17.4.8.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

`Insufficient_Permission`
: Either the application does not have the permission to use the specified address or options.

`Address_In_Use`
: The requested address is in use.

`Incorrect_Address_Format`
: The address specified by the application contained an incorrect protocol address or was in an incorrect format.

`Illegal_Data_Range`
: The application attempted to send an illegal amount of data.
Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Incorrect_Or_Illegal_Option`
: The option specified by the application contained incorrect information, or information was in an incorrect format.

`Buffer_Not_Large_Enough`
: The amount of data to be returned in one of the buffers is greater than the size of the buffer.

`Event_Requires_Attention`
: An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`No_Data_Available`
: In nonblocking mode, the procedure has successfully issued a connect but did not wait for a response from the application.

`XTI_Operation_Not_Supported`
: The communications provider does not support this operation.

`Operation_Not_Valid_For_State`
: The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

17.4.9 Gather and send data or expedited data over a connection

17.4.9.1 Synopsis

```
procedure Gather_And_Send_Data
   (Endpoint        : in  POSIX_IO.File_Descriptor;
    Vector          : in  IO_Vector_Array;
    Flags           : in  XTI_Flags;
    Octets_Sent     : out POSIX.IO_Count);
```

17.4.9.2 Description

The Gather_And_Send_Data procedure is used to send either normal or expedited data. Endpoint shall identify the local communications endpoint over which data are to be sent. The parameter Vector identifies an IO_Vector_Array object, which is an array of POSIX_IO.IO_Vector objects with Buffer and Length attributes (*i.e.*, an array of buffer address/buffer size pairs). See 17.4.1.9 for a detailed description of these objects, including the appropriate attribute access operations. Buffer sizes can be zero.

The number of POSIX_IO.IO_Vector objects to send is implicit in the array or array slice specified by the Vector parameter. The IO_Vector objects shall be sent starting from Vector'First.

If POSIX_IO.Non_Blocking is clear (via Open or Set_File_Control), Gather_And_Send_Data shall return immediately if there is a permanent failure condition, but can wait if there is a transient error condition or if flow control restrictions prevent the data from being accepted by the local communications provider at the time the call is made.

If POSIX_IO.Non_Blocking is set, Gather_And_Send_Data shall fail immediately if there is a permanent or transient failure condition, or there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared via either Look or via the event management operations defined in 19.1 (or another suitable interface offered by the implementation).

On successful completion, Gather_And_Send_Data shall return the number of octets accepted by the communications provider in the Octets_Sent parameter. Normally this number will equal the number of octets to be sent. However, it is possible that only part of the data will actually be accepted by the communications provider. In this case, Octets_Sent shall contain a value that is less than the total number of octets to be sent. The application should adjust the input parameters and call Send or Gather_And_Send_Data again in order to send the remaining octets.

If Gather_And_Send_Data is interrupted by a signal before it could transfer data to the communications provider, it shall raise POSIX_Error with error code Interrupted_Operation.

If the number of octets in Vector is zero and sending of zero octets is not supported by the underlying communications service, Gather_And_Send_Data shall raise POSIX_Error with error code Illegal_Data_Range. The size of each SDU or SEDU shall not exceed the limits of the communications provider as specified by the current values in the Max Size SDU or Max Size SEDU attribute of the Communications_Provider_Info object returned by Open and Get_Info.

The error Event_Requires_Attention may be returned to inform the process that an event (*e.g.*, a disconnection) is outstanding.

17.4.9.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Illegal_Data_Range`

The application attempted to send zero octets of data and the sending of zero octets is not supported by the underlying communications service, or the application attempted to send a nonzero illegal amount of data, such as in the following cases:

— A single send was attempted specifying an SDU, an SEDU, a fragment SDU, or a fragment SEDU greater than that specified by the current values of Max Size SDU or Max Size SEDU,

— a send of a zero octet SDU, a zero octet SEDU, a zero octet fragment of an SDU, or a zero octet fragment of an SEDU is not supported by the provider (see D.2), or

— multiple sends were attempted resulting in an SDU or SEDU larger than that specified by the current values of Max Size SDU or Max Size SEDU - the ability of an XTI implementation to detect such an error case is implementation dependent.

— The length of `Vector` was greater than `POSIX_Limits.XTI_IO_Vector_Maxima'Last`.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Invalid_Flag`

The application specified an invalid flag.

`Flow_Control_Error`

Flow conditions prevented the sending of data at this time in asynchronous (nonblocking) mode.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.10 Gather and Send a Data Unit

17.4.10.1 Synopsis

```
procedure Gather_And_Send_Data_Unit
   (Endpoint        : in POSIX_IO.File_Descriptor;
    Address         : in XTI_Address_Pointer;
    Vector          : in IO_Vector_Array);
procedure Gather_And_Send_Data_Unit
   (Endpoint        : in POSIX_IO.File_Descriptor;
    Address         : in XTI_Address_Pointer;
    Vector          : in IO_Vector_Array;
    Options         : in Protocol_Option_List);
```

17.4.10.2 Description

The `Gather_And_Send_Data_Unit` procedure is used in connectionless mode to send a data unit to another communications application. `Endpoint` shall identify the local communications endpoint through which data are to be sent. The parameter `Vector` identifies an `IO_Vector_Array` object, which is an array of `POSIX_IO.IO_Vector` objects with Buffer and Length attributes (*i.e.*, an array of buffer address/buffer size pairs). See 17.4.1.9 for a detailed description of these objects, including the appropriate attribute access operations. Buffer sizes can be zero.

The number of `POSIX_IO.IO_Vector` objects to send is implicit in the array or array slice specified by the `Vector` parameter. The `POSIX_IO.IO_Vector` objects shall be sent starting from `Vector'First`.

The `Address` parameter shall specify the protocol address of the destination application, `Options` shall identify options that the application wants associated with this request. The application need not specify what protocol options are associated with the transfer. In this case the provider shall use the option values currently set for this communications endpoint (see 17.3.2.3 and 17.3.3). The data to be sent shall be contained in the `POSIX_IO.IO_Vector` objects in the `IO_Vector_Array` referenced by `Vector`.

If the number of octets to send is zero and sending of zero octets is not supported by the underlying communications service, then `Gather_And_Send_Data_Unit` shall raise `POSIX_Error` with error code `Illegal_Data_Range`.

If `POSIX_IO.Non_Blocking` is clear (via `Open` or `Set_File_Control`), `Gather_And_Send_Data_Unit` can wait if flow control restrictions prevent the data from being accepted by the local communications provider at the time the call is made. However, if `POSIX_IO.Non_Blocking` is set, `Gather_And_Send_Data_Unit` shall fail under such conditions. The process can arrange to be notified of the clearance of a flow control restriction via either `Look` or the event management operations defined in 19.1 (or another suitable interface offered by the implementation).

If the amount of data specified in `Vector` exceeds the SDU size as returned in the Max Size SDU attribute of the `Communications_Provider_Info` object returned by `Open` or `Get_Info` or if the amount is zero and sending of zero octets is not supported by the communications provider, a `Illegal_Data_Range` error shall be generated. If `Gather_And_Send_Data_Unit` is called before the destination application has activated its communications endpoint (see `Bind`), the data unit may be discarded.

If it is not possible for the communications provider to immediately detect the conditions that cause the errors `Illegal_Data_Range` and `Incorrect_Or_Illegal_Option`, these errors shall alternatively be detected by `Retrieve_Data_Unit_Error`. Therefore, an application shall be prepared to receive these errors in both of these ways.

If `Gather_And_Send_Data_Unit` is interrupted by a signal, it shall raise `POSIX_Error` with error code `Interrupted_Operation`, and the data unit (datagram) shall not have been sent.

17.4.10.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Incorrect_Address_Format`

The address specified by the application contained an incorrect protocol address or was in an incorrect format.

`Illegal_Data_Range`

The application attempted to send a illegal amount of data.

— A single send was attempted specifying an SDU greater than that specified by the current values of Max Size SDU,

— a send of a zero octet SDU is not supported by the provider (see D.2), or

— The length of `Vector` was greater than `POSIX_Limits.XTI_IO_Vector_Maxima'Last`.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Incorrect_Or_Illegal_Option`

The option specified by the application contained incorrect information, or the information was in an incorrect format.

`Flow_Control_Error`

Flow conditions prevented the sending of data at this time in asynchronous (nonblocking) mode.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.11 Get the Current State

17.4.11.1 Synopsis

```
function Get_Current_State (Endpoint : POSIX_IO.File_Descriptor)
  return Interface_State;
```

17.4.11.2 Description

The `Get_Current_State` function returns the state of the communications endpoint specified by `Endpoint`. The state returned is the current state of the endpoint as seen by the application. This state is consistent with the processing performed by the previous XTI calls of the application. It is not necessarily the same as the state maintained in the provider, since the provider receives events before the application. See Table 17.6 and Figure 17.1.

The valid states are returned with type `Interface_State`, and are as follows:

`Idle`

No connection established

`Unbound`

Unbound

`Outgoing_Connect`

Outgoing connection pending for active application

`Incoming_Connect`

Incoming connection pending for passive application

`Data_Transfer`

Data transfer

`Outgoing_Release`

Outgoing orderly release (waiting for orderly release indication)

`Incoming_Release`

Incoming orderly release (waiting to send orderly release request)

This function may be used from all states except Uninitialized.

17.4.11.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

`State_Change_In_Progress`
: The communications provider is undergoing a state change.

17.4.12 Get Protocol-Specific Service Information

17.4.12.1 Synopsis

```
procedure Get_Info
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Info     : out Communications_Provider_Info);
```

17.4.12.2 Description

`Get_Info` returns the current characteristics of the underlying communications protocol and/or communications connection associated with `Endpoint`. It enables a communications application to access this information during any phase of communication.

`Info` is a `Communications_Provider_Info` object returning the default characteristics of the underlying communications provider. This information has the same structure as that obtained by `Open`, but not necessarily the same values. These values will differ from those returned in `Open` after the endpoint enters the Data Transfer state. See 17.4.1.2 for detailed information about this object.

The `Manage_Options` operation has no effect on the values returned by `Get_Info`.

This procedure may be used from all states except Uninitialized.

17.4.12.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

Invalid_File_Descriptor
: A bad file descriptor was specified for the communications endpoint.

Protocol_Error
: A communication problem has occurred, and there is no other appropriate error number.

17.4.13 Get the Protocol Address

17.4.13.1 Synopsis

```
procedure Get_Protocol_Address
   (Endpoint      : in POSIX_IO.File_Descriptor;
    Bound_Address : in XTI_Address_Pointer;
    Peer_Address  : in XTI_Address_Pointer);
```

17.4.13.2 Description

The Get_Protocol_Address procedure returns local and remote protocol addresses currently associated with the communications endpoint specified by Endpoint. In Bound_Address and Peer_Address, the application specifies an XTI_Address_Pointer.

On return, Bound_Address contains the address, if any, currently bound to Endpoint. Peer_Address contains the address, if any, currently connected to Endpoint.

17.4.13.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_File_Descriptor
: The specified file descriptor does not refer to a communications endpoint.

Incorrect_Address_Format
: The address specified by the application contained an incorrect protocol address, or the protocol address was in an incorrect format.

Protocol_Error
: A communication problem has been detected between XTI and the communications provider for which there is no other suitable value.

17.4.14 Initiate an Orderly Release

17.4.14.1 Synopsis

```
procedure Initiate_Orderly_Release
   (Endpoint : in POSIX_IO.File_Descriptor);
procedure Initiate_Orderly_Release
   (Endpoint : in POSIX_IO.File_Descriptor;
    Reason   : in Reason_Code);
```

17.4.14.2 Description

The procedure `Initiate_Orderly_Release` is used to initiate an orderly release of the connection on `Endpoint` or to respond to an orderly release indication. It indicates to the communications provider that the communications application has finished sending data. The `Reason` parameter specifies the reason for the orderly release through a protocol-specific reason code.

After the orderly release indication is sent, the application may continue to receive data as long as an orderly release indication has not been received, but no additional data can be sent over the connection after the `Initiate_Orderly_Release` call.

This procedure is one of the optional services of the communications provider and is supported only if the `Open` or `Get_Info` call returned service type of `Connection_-Mode_With_Orderly_Release`.

This procedure shall be used from the Data Transfer state or the Incoming Release state.

17.4.14.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Flow_Control_Error`
: Flow conditions prevented the sending of data at this time in asynchronous (nonblocking) mode.

`Event_Requires_Attention`
: An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`XTI_Operation_Not_Supported`
: This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`
: The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

17.4.15 Initiate an Orderly Release with Application Data

17.4.15.1 Synopsis

```
procedure Initiate_Orderly_Release_With_Data
   (Endpoint       : in POSIX_IO.File_Descriptor;
    Reason         : in Reason_Code;
    User_Data      : in System.Address;
    Octets_To_Send : in POSIX.IO_Count);
```

17.4.15.2 Description

The `Initiate_Orderly_Release_With_Data` procedure is used to initiate an orderly release of a connection or to respond to an orderly release indication. It permits an application to send data with the release. `Endpoint` identifies the local communications endpoint where the connection exists.

After calling `Initiate_Orderly_Release_With_Data`, the application shall not send any more data over the connection. However, an application can continue to receive data if no orderly release confirmation has been received.

The `Reason` parameter specifies the reason for the orderly release through a protocol-specific reason code, and `User_Data` identifies any application data that is sent to the remote application with the orderly release.

The amount of application data to be sent, specified by the `Octets_To_Send` parameter, is limited by the Max Size Disconnect Data value returned in the `Communications_Provider_Info` object from `Open` or `Get_Info`. If `Octets_To_Send` is zero, no data shall be sent to the remote application.

This operation is an optional service of the communications provider and is only supported if the communications provider returned service type `Connection_Mode_With_Orderly_Release` on `Open` or `Get_Info`. The flag `Orderly_Release_Data_Supported` in the CP Flags attribute of the `Communications_Provider_Info` object returned by `Open` or `Get_Info` indicates that the provider will accept orderly release application data.

17.4.15.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Illegal_Data_Range`
: The application attempted to send an illegal amount of data.

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Flow_Control_Error`

`POSIX_IO.Non_Blocking` was set, but the flow control mechanism prevented the communications provider from accepting the operation at this time.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

17.4.16 Listen for a Connection Indication

17.4.16.1 Synopsis

```
procedure Listen
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Call     : in out Connection_Info);
```

17.4.16.2 Description

The procedure `Listen` is called by the server (or passive application) to listen for a connection indication from a client (or calling application). `Endpoint` identifies the local communications endpoint where connection indications arrive; and, on return, `Call` contains information describing the connection indication.

The `Call` parameter is a `Connection_Info` object that, on return, contains information about the connection to the client (the active application that issued the connection request). The Address attribute is used to return the protocol address in a format usable in future calls to `Connect`. However, `Connect` may fail for other reasons, for example, `Address_In_Use`. The Options attribute is used to return protocol-specific options associated with the connection indication. The User Data attribute returns application data sent by the client on the connection request. The Sequence attribute is a number that uniquely identifies the returned connection indication. The Sequence attribute gives the application the option of listening for multiple connection requests before responding to any of them.

The Address attribute, the Options Buffer attribute of Options, and the User Data attribute shall be supplied by the application to specify the buffers to be used to return

options and application data. If the size of the storage referenced by the Address attribute, the Options Buffer attribute of Options, or the User Data attribute is zero, no information shall be returned. If the size of the storage referenced by any of these attributes is greater than zero and less than the length of the connection information, `Listen` shall fail with the error `Buffer_Not_Large_Enough`.

The maximum number of queued connection requests is limited by the value of the `Response_Queue_Length` parameter of the `Bind` procedure (see 17.4.5).

By default, `Listen` waits for a connection indication to arrive before returning to the application. However, if `POSIX_IO.Non_Blocking` is set via `Open` or `Set_File_Control`, `Listen` reduces to a poll for existing connection indications. If none is available, it shall raise `POSIX_Error` with error code `No_Data_Available`.

Some communications providers do not differentiate between a connection indication and the connection itself. In this case, a successful return of `Listen` indicates an existing connection (see D.2).

This procedure shall be used from the Idle state or the Incoming Connect state.

17.4.16.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Endpoint_Queue_Length_Is_Zero`
: The endpoint queue length was zero when it was expected to be greater than zero.

`Buffer_Not_Large_Enough`
: The amount of data to be returned in one of the buffers is greater than the buffer size. The state of the endpoint, as seen by the application, changes to Incoming Connect, and the connection indication information to be returned in `Call` is discarded. The value of the Sequence attribute returned can be used as a parameter to a subsequent call to `Send_Disconnect_Request`.

`Event_Requires_Attention`
: An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (Applications can call `Look` to retrieve the event.)

`No_Data_Available`
: In nonblocking mode, no connection indications are present.

`XTI_Operation_Not_Supported`
: This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

`Endpoint_Queue_Full`

The maximum number of queued connection requests has been reached.

17.4.17 Look at the Current Event on a Communication Endpoint

17.4.17.1 Synopsis

```
type XTI_Events is new POSIX.Option_Set;
Connect_Request_Received    : constant XTI_Events := implementation-defined;
Connect_Response_Received   : constant XTI_Events := implementation-defined;
Disconnect_Request_Received : constant XTI_Events := implementation-defined;
Error_In_Previously_Sent_Datagram
                            : constant XTI_Events := implementation-defined;
Expedited_Data_Received     : constant XTI_Events := implementation-defined;
Normal_Data_Received        : constant XTI_Events := implementation-defined;
Okay_To_Send_Expedited_Data : constant XTI_Events := implementation-defined;
Okay_To_Send_Normal_Data    : constant XTI_Events := implementation-defined;
Orderly_Release_Request_Received
                            : constant XTI_Events := implementation-defined;
function Look (Endpoint : POSIX_IO.File_Descriptor)
  return XTI_Events;
```

17.4.17.2 Description

The function `Look` is used to return the current event on the communications endpoint specified by `Endpoint`. This function enables a communications provider to notify an application of an event when the application is calling operations with `POSIX_IO.Non_Blocking` not set. Certain events require immediate notification of the application and are indicated by a specific error, `Event_Requires_Attention`, on the current or next operation to be executed. Details on events that cause operations to fail with `Event_Requires_Attention` may be found in 17.2.6.

This operation also enables an application to poll a communications endpoint periodically for events.

The type `XTI_Events` shall denote a set of XTI event flags. The operations `"+"`, `"-"`, `">"`, `"<"`, `">="`, `"<="`, and `Empty_Set` are available on the type `XTI_Events` via the derived type semantics of Ada, from the operations available for `POSIX.Option_Set`. The appropriate operations can be used to create and examine a set containing the required event flags.

Valid events are as follows:

`Connect_Request_Received`

A connection indication has been received.

`Connect_Response_Received`

A connect confirmation has been received.

`Normal_Data_Received`
: Normal data have been received.

`Expedited_Data_Received`
: Expedited data have been received.

`Disconnect_Request_Received`
: A disconnect request has been received.

`Error_In_Previously_Sent_Datagram`
: An error has been detected in a previously sent datagram.

`Orderly_Release_Request_Received`
: An orderly release request has been received.

`Okay_To_Send_Normal_Data`
: Flow control conditions have been lifted, and normal data can again be sent.

`Okay_To_Send_Expedited_Data`
: Flow control conditions have been lifted, and expedited data can again be sent.

This procedure may be used from all states except Uninitialized.

17.4.17.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

17.4.18 Manage options for a communication endpoint

17.4.18.1 Synopsis

```
XTI_Protocol_Level     : constant Option_Level := implementation-defined;
Unspecified            : constant Option_Value := implementation-defined;
All_Options            : constant Option_Name := implementation-defined;
Enable_Debugging       : constant Option_Name := implementation-defined;
Linger_On_Close_If_Data_Present
                       : constant Option_Name := implementation-defined;
Receive_Buffer_Size    : constant Option_Name := implementation-defined;
Receive_Low_Water_Mark : constant Option_Name := implementation-defined;
Send_Buffer_Size       : constant Option_Name := implementation-defined;
Send_Low_Water_Mark    : constant Option_Name := implementation-defined;
procedure Manage_Options
   (Endpoint       : in     POSIX_IO.File_Descriptor;
    Request        : in     Protocol_Option_List;
    Request_Flag   : in     Options_Flags;
    Response       : in out Protocol_Option_List;
    Response_Flags : out    Option_Status);
```

17.4.18.2 Description

The Manage_Options procedure enables an application to retrieve, verify, or negotiate protocol options with the communications provider. The parameter Endpoint identifies a communications endpoint.

The Request and Response parameters are Protocol_Option_List objects used to specify the option(s) that are of interest. See 17.4.1.6 for a detailed description of the Protocol_Option_List object. The Request_Flag parameter has the type Options_Flags, and the Response_Flags parameter has the type Option_Status. Request_Flag requests actions of the communications provider relative to the option(s). See 17.4.1.1 for a description of the Options_Flags type.

The parameter Request is used to request a specific action of the provider and to send options to the provider. The communications provider may return options and flag values to the application through Response and Response_Flags. For Response, the Options Buffer attribute specifies the buffer where the options are to be placed. If the size of the Options Buffer attribute of Response is zero, then no option values shall be returned.

NOTE: The Options Buffer attribute of the Response and Request parameters may point to the same POSIX.Octet_Array object.

If the application specifies several options in Request, all options shall address the same level. If any option in the options buffer does not indicate the same level as the first option or the level specified is unsupported, then the Manage_Options request shall fail with Incorrect_Or_Illegal_Option. If the error is detected, some options have possibly been successfully negotiated. The application can check the current status by calling Manage_Options with the Request_Flag parameter set to Get_Current_Options.

See 17.3 for a detailed description of the use of options, which should be read before using this operation.

The Request_Flag parameter shall specify one of the following actions:

Negotiate_Options

This action enables the application to negotiate option values.

The application specifies the options of interest and their values in the Options Buffer attribute of the Request parameter. The negotiated option values are returned in the Options Buffer attribute of the Response parameter. The Status attribute of each returned option is set to indicate the result of the negotiation. The value is Success if the proposed value was negotiated, Partial_Success if a degraded value was negotiated, Failure if the negotiation failed (according to the negotiation rules), Not_Supported if the communications provider does not support this option or illegally requests negotiation of a privileged option, and Read_Only if modification of a read-only option was requested. If the status is Success, Failure, Not_Supported, or Read_Only, the returned option value is the same as the one requested on input. If the status is Partial_Success, the returned option value is the negotiated option value.

The overall result of the negotiation is returned in Response_Flags.

This attribute contains the worst single result. The rating is done according to the order Not_Supported, Read_Only, Failure, Partial_Success, Success. In other words, the value Not_Supported is the worst result and Success is the best.

For each level, the option name All_Options can be requested on input. No value is given with this option. This input requests to negotiate all supported options of this level to their default values. The result is returned option by option in the Options Buffer attribute of Response.

NOTE: Depending on the state of the communications endpoint, not all requests to negotiate the default value may be successful.

Check_Options

This action enables the application to verify whether the options specified in Request are supported by the communications provider.

If an option is specified with no option value, the option is returned with its status attribute set to Success if it is supported, Not_Supported if it is not supported or needs additional application privileges, and Read_Only if it is read-only (in the current XTI state). No option value is returned.

If an option is specified with an option value, the Status attribute of the returned option has the same value as if the application had tried to negotiate this value with Negotiate_Options. If the status is Success, Failure, Not_Supported or Read_Only, the returned option value is the same as the one requested on input. If the status is Partial_Success, the returned option value is the negotiated option value.

The overall result of the option checks is returned in Response_Flags. This value contains the worst single result of the option checks, where worst has the same meaning as in the description of Negotiate_Options (above).

Get_Default_Options

This action enables the application to retrieve the default option values. The application specifies the options of interest in the Options Buffer attribute of the Request parameter. The option values are irrelevant and shall be ignored; it is sufficient to specify only the Option_Level and Option_Name part of an option. The default values are then returned in the Options Buffer attribute of the Response parameter.

The Status attribute returned is Not_Supported if the protocol level does not support this option or the application illegally requested a privileged option, Read_Only if the option is read-only, and Success in all other cases. The overall result of the request is returned in Response_Flags. This value contains the worst single result, where worst has the same meaning as in the description of Negotiate_Options (above).

For each level, the option name All_Options can be requested in Request. All supported options of this level with their default values are then returned. In this case, the size of the Options Buffer attribute shall be at least the value of the Max Size Protocol Options attribute of Communications_Provider_Info object (see Get_Info, 17.4.12 and Open, 17.4.19) before the call.

`Get_Current_Options`

This action enables the application to retrieve the currently effective option values. The application specifies the options of interest in the Options Buffer attribute of `Request`. The option values are irrelevant and shall be ignored; it is sufficient to specify the `Option_Level` and `Option_Name` part of an option only. The currently effective values are then returned in the Options Buffer attribute of `Response`.

The status attribute returned is `Not_Supported` if the protocol level does not support this option or the application illegally requested a privileged option, `Read_Only` if the option is read-only, and set to `Success` in all other cases. The overall result of the request is returned in `Response_Flags`. This attribute contains the worst single result, where worst has the same meaning as in the description of `Negotiate_Options` (above).

For each level, the option name `All_Options` can be requested on input. All supported options of this level with their currently effective values are then returned.

The option name `All_Options` be used only with `Manage_Options` and the actions `Negotiate_Options`, `Get_Default_Options`, and `Get_Current_Options`. It can be used with any supported level and addresses all supported options of this level. The option has no value. Since in a `Manage_Options` call only options of one level may be addressed, this option should not be requested together with other options. The operation returns as soon as this option has been processed.

Options are independently processed in the order they appear in the input protocol option list. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

Communications providers may not be able to provide an interface capable of supporting `Negotiate_Options` and/or `Check_Options` functionalities. In this case, the error `XTI_Operation_Not_Supported` is generated.

`Manage_Options` may block under various circumstances and depending on the implementation. The procedure shall block, for instance, if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints, *i.e.*, if data sent previously across this communications endpoint have not yet been fully processed. If the operation is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behavior of the operation is not changed if `POSIX_IO.Non_Blocking` is set.

XTI-level options are not specific for a particular communications provider. An XTI implementation supports none, all, or any subset of the options listed in this subclause. An implementation may restrict the use of any of these options by offering them in only the privileged or read-only mode or if `Endpoint` relates to specific communications providers.

The subsequent options do not have end-to-end significance (see 17.3). They may be negotiated in all XTI states except Uninitialized.

The protocol level is `XTI Protocol Level`. For this level, the Enable Debugging XTI option, the Linger On Close If Data Present XTI option, the Receive Buffer Size XTI option,

the Receive Low Water Mark XTI option, the Send Buffer Size XTI option, and the Send Low Water Mark XTI option are defined.

A request for the Enable Debugging XTI option is an absolute requirement. A request to activate the Linger On Close If Data Present XTI option is an absolute requirement; the timeout value to this option is not. The Receive Buffer Size XTI option, the Receive Low Water Mark XTI option, the Send Buffer Size XTI option, and Send Low Water Mark XTI option are not absolute requirements.

The meanings of these options and the correspondance to Ada constants are as follows:

`Enable_Debugging`

The Enable Debugging XTI option enables debugging. The values of this option are implementation defined. Debugging is disabled if the option is specified with no value, *i.e.*, with only an `Option_Level` and `Option_Name` part of an option specified.

`Linger_On_Close_If_Data_Present`

The Linger On Close If Data Present XTI option is used to delay the execution of a `Close` if send data are still queued in the send buffer. The option value specifies the linger period. If a `Close` is issued and the send buffer is not empty, the system attempts to send the pending data within the linger period before closing the endpoint. Data still pending after the linger period has elapsed are discarded.

Depending on the implementation, `Close` may either block for at maximum the linger period, or immediately return, whereupon the system holds the connection in existence for at most the linger period.

The option value consists of a `Linger_Info` object that shall contain the Linger Status and Linger Period attributes. See 17.4.1.4 for a complete description of the attributes of this object. Legal values for the Linger Status attribute are

`Off`

Switch option off

`On`

Activate option

The value Linger Status is an absolute requirement.

The attribute Linger Period determines the linger period in seconds. The application can request the default value or set the attribute to infinite via the procedures `Set_Period_Unspecified` and `Set_Period_Infinite` respectively.

NOTE: The default timeout value depends on the underlying communications provider (it is often infinite).

The Linger Period value is not an absolute requirement. The implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

This option does not linger the execution of `Send_Disconnect_Request`.

`Receive_Buffer_Size`

The Receive Buffer Size XTI option is the internal buffer size, in octets, allocated for the receive buffer. The buffer size may be increased for high-volume connections, or decreased to limit the possible backlog of incoming data.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

`Receive_Low_Water_Mark`

The Receive Low Water Mark XTI option is used to set a low-water mark in the receive buffer. The option value gives the minimal number of octets that shall have accumulated in the receive buffer before they become visible to the application. If and when the amount of accumulated receive data exceeds the low-water mark, a `Normal_Data_Received` event is created, an event mechanism (*e.g.*, `Poll` or `Select_File`) indicates the data, and the data can be read by `Receive` or `Receive_Data_Unit`.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

`Send_Buffer_Size` The Send Buffer Size XTI option is used to adjust the internal buffer size, in octets, allocated for the send buffer.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers

`Send_Low_Water_Mark`

The Send Low Water Mark XTI option a low-water mark in the send buffer. The option value gives the minimal number of octets that shall have accumulated in the send buffer before they are sent.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

17.4.18.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Incorrect_Or_Illegal_Option`

The option specified by the application contained incorrect information, or information was in an incorrect format.

`Invalid_Flag`

The application specified an invalid flag.

`Buffer_Not_Large_Enough`

The amount of data to be returned in one of the buffers is greater than the size of the buffer.

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.19 Establish a Communication Endpoint

17.4.19.1 Synopsis

```
procedure Open
   (Endpoint : out    POSIX_IO.File_Descriptor;
    Name     : in     POSIX.POSIX_String;
    Mode     : in     POSIX_IO.File_Mode;
    Options  : in     POSIX_IO.Open_Option_Set;
    Info     : in out Communications_Provider_Info);
procedure Open
   (Endpoint : out POSIX_IO.File_Descriptor;
    Name     : in  POSIX.POSIX_String;
    Mode     : in  POSIX_IO.File_Mode;
    Options  : in  POSIX_IO.Open_Option_Set);
```

17.4.19.2 Description

The procedure `Open` shall be called as the first step in the initialization of a communications endpoint. This operation establishes a communications endpoint by supplying a communications provider identifier that indicates a particular communications provider (*i.e.*, communications protocol) and returning a file descriptor that identifies that endpoint. The file descriptor returned, `Endpoint`, is used in subsequent procedure calls.

`Name` is a string containing the name of the communications provider (for example, "`/dev/tcp`" for the TCP/IP protocol or "`/dev/cot4`" for the OSI protocol) and specifies the particular communications provider to associate with the endpoint.

`Mode` and `Options` are used to specify the file mode and whether the communications endpoint shall operate in blocking or nonblocking mode.

The default characteristics of the communications provider are returned as `Communications_Provider_Info` object by the `Info` parameter. See 17.4.1.2 for detailed information about this object.

If an application is concerned with protocol independence, it can use the sizes given in the `Communications_Provider_Info` object to determine how large the buffers

need to be to hold each piece of information. An error shall result if an application exceeds the allowed data size on any operation.

The Service Type attribute of `Info` shall specify one of the values shown in 17.4.1.2 on return. A single communications endpoint can support only one of these services at one time.

If `Info` is omitted, no protocol information shall be returned.

This procedure shall be used from Uninitialized state.

17.4.19.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Flag`

The application specified an invalid flag.

`Invalid_Communications_Provider`

The application specified a bad name for the communications provider.

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.20 Receive Data or Expedited Data Sent Over a Connection

17.4.20.1 Synopsis

```
procedure Receive
   (Endpoint         : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Flags            : out XTI_Flags);
```

17.4.20.2 Description

The procedure `Receive` is called to receive either normal or expedited data through the local communications endpoint specified by `Endpoint` into the buffer specified by `Buffer`. The amount of data requested is specified in the `Octets_Requested` parameter. `Octets_Received` returns the number of octets of data returned on success. The parameter `Flags` may be set on return from `Receive` and specifies optional flags as described in this subclause. `Flags` has the type `XTI_Flags`, derived from the type `POSIX.Option_Set`. See 17.4.1.1 for a description of how to create and examine sets of these flags.

If POSIX_IO.Non_Blocking is clear (via Open or Set_File_Control), Receive shall wait for data to arrive if none is currently available. However, if POSIX_IO.-Non_Blocking is set, Receive shall fail if no data are available. If, on return from the operation, More_Data is set in Flags, this indicates that there is more data and therefore the SDU or SEDU needs to be received using multiple Receive calls.

The More_Data flag may be set on return from Receive even when the number of octets received is less than the size of the receive buffer specified, *e.g.*because POSIX_IO.Non_Blocking was set and the operation was interrupted by a signal when an Expedited_Data_Received event occurred.

Each Receive with the More_Data flag set indicates that another Receive is needed to get more data for the current SDU. The end of the SDU is identified by the return of a Receive operation with the More_Data flag not set. If the communications provider does not support the concept of a SDU as indicated in the Communications_Provider_Info parameter on return from Open or Get_Info, the More_Data flag is not meaningful and should be ignored. Receive shall return an Octets_Received value of zero only if the end of an SDU is being returned to the application.

If Receive returns with Expedited_Data set in Flags, then the data returned are expedited data. If the number of octets of expedited data exceeds the size of the buffer, or a signal interrupts the procedure, or (for communications protocols that support fragmentation of SEDUs) an entire SEDU is not available, then Receive shall set Expedited_Data and More_Data on return from the initial call. Subsequent calls that retrieve the remainder of the SEDU shall have Expedited_Data set on return. The end of the SEDU is identified by the return of Receive with the Expedited_Data flag set and the More_Data flag not set. If an entire SEDU is not available, it is possible for normal data fragments to be returned between fragments of the SEDU.

If a signal arrives Receive shall return, giving the application any data currently available. If no data are available, Receive shall generate the error Interrupted_-Operation. If some data are available, Receive shall return the number of octets received and set the More_Data flag if the communications provider supports the concept of an SDU.

When POSIX_IO.Non_Blocking is clear, the only way for the application to be notified of the arrival of normal or expedited data is to issue this operation or check for the Normal_Data_Received or the Expedited_Data_Received events using the Look function. Additionally, the process can arrange to be notified via the event management operations defined in 19.1 (or another suitable interface offered by the implementation).

This procedure shall be used from the Data Transfer state or the Outgoing Release state.

17.4.20.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_File_Descriptor

A bad file descriptor was specified for the communications endpoint.

Event_Requires_Attention

An asynchronous event is outstanding on the communications endpoint specified by Endpoint. (The application can call Look to retrieve the event.)

No_Data_Available

In nonblocking mode, data are not available.

XTI_Operation_Not_Supported

This operation is not supported by the current implementation of XTI.

Operation_Not_Valid_For_State

The operation was called with the communications provider in the wrong state (bad sequence).

Protocol_Error

A communication problem has occurred, and there is no other appropriate error number.

17.4.21 Receive and Scatter Data or Expedited Data Sent Over a Connection

17.4.21.1 Synopsis

```
procedure Receive_And_Scatter_Data
   (Endpoint          : in  POSIX_IO.File_Descriptor;
    Vector            : in  IO_Vector_Array;
    Octets_Received   : out POSIX.IO_Count;
    Flags             : out XTI_Flags);
```

17.4.21.2 Description

The Receive_And_Scatter_Data procedure receives either normal or expedited data. Endpoint identifies the communications endpoint through which data are to arrive, and Vector identifies an IO_Vector_Array object, which is an array of POSIX_IO.IO_Vector objects with Buffer and Length attributes (*i.e.*, an array of buffer address/buffer size pairs). See 17.4.1.9 for a detailed description of IO_Vector_Array objects, including the appropriate attribute access operations. Procedure Receive_And_Scatter_Data shall receive data into these buffers, always filling each buffer with the number of octets specified by its Length attribute before proceeding to the next.

NOTE: The total size of the buffers passed may be constrained by implementation limits. In practice, the availability of memory to an application is likely to impose a limit on the amount of data that can be sent or received using scatter/gather operations.

The number of POSIX_IO.IO_Vector objects to receive is implicit in the array or array slice specified by the Vector parameter. The POSIX_IO.IO_Vector objects shall be received starting from Vector'First. Argument Flags is type XTI_Flags and may be set on return from Receive_And_Scatter_Data.

If POSIX_IO.Non_Blocking is clear, Receive_And_Scatter_Data shall wait for data to arrive if none is currently available. However, if POSIX_IO.Non_Blocking is set (via Open or Set_File_Control), Receive_And_Scatter_Data shall fail with error code No_Data_Available if no data are available.

If, on return from the operation, More_Data is set in Flags, there are more data and therefore the SDU or SEDU needs to be received using multiple Receive calls.

The More_Data flag may be set on return from the Receive_And_Scatter_Data procedure even when the number of octets received is less than the total size of all the receive buffers specified.

NOTE: The following are typical situations that cause less than the requested number of octets to be received:

— *POSIX_IO.Non_Blocking is set.*

— *The operation is interrupted by a signal.*

— *An Expedited_Data_Received event occurs.*

Each Receive_And_Scatter_Data call with the More_Data flag set indicates that another Receive_And_Scatter_Data call needs to be made to obtain more data for the current SDU. The end of the SDU is identified by the return of a Receive_And_-Scatter_Data call with the More_Data flag cleared. If the communications provider does not support the concept of an SDU as indicated in the Communications_-Provider_Info parameter on return from Open or Get_Info, the More_Data flag is not meaningful and shall be ignored. If the amount of buffer space passed in Vector is greater than zero on the call to Receive_And_Scatter_Data, then Receive_-And_Scatter_Data shall return zero in the Octets_Received parameter only if the end of an SDU is being returned to the application.

On return, the data returned are expedited data if Expedited_Data is set in Flags. If the number of octets of expedited data exceeds the total number of octets available in all the buffers passed, or a signal interrupts the operation, or (for communications protocols that support fragmentation of SEDUs) an entire SEDU is not available, then Receive_And_Scatter_Data shall set Expedited_Data and More_Data on return from the initial call. Subsequent calls that retrieve the remainder of the SEDU shall have Expedited_Data set on return. The end of the SEDU is identified by the return of Receive_And_Scatter_Data with the Expedited_Data flag set and the More_Data flag clear. If an entire SEDU is not available, it is possible for normal data fragments to be returned between fragments of the SEDU.

If a signal arrives, Receive_And_Scatter_Data shall return, giving the application any data currently available. If no data are available, Receive_And_Scatter_-Data shall raise POSIX_Error with error code Interrupted_Operation. If some data are available, Receive_And_Scatter_Data shall return the number of octets received in the Octets_Received parameter and set the More_Data flag if the communications provider supports the concept of an SDU.

When POSIX_IO.Non_Blocking is clear, the only way for the application to be notified of the arrival of normal or expedited data is to issue this operation or check for the Normal_Data_Received or Expedited_Data_Received events using the Look function. Additionally, the process can arrange to be notified via the event management operations defined in 19.1.

17.4.21.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Illegal_Data_Range`

The length of the array specified by `Vector` was greater than the limit specified by `POSIX_Limits.XTI_IO_Vector_Maxima'Last`.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`No_Data_Available`

In nonblocking mode, data are not available.

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.22 Receive and Scatter a Data Unit

17.4.22.1 Synopsis

```
procedure Receive_And_Scatter_Data_Unit
   (Endpoint          : in     POSIX_IO.File_Descriptor;
    Address           : in     XTI_Address_Pointer;
    Options           : in out Protocol_Option_List;
    Vector            : in     IO_Vector_Array;
    Octets_Received   : out POSIX.IO_Count;
    Flags             : out XTI_Flags);
```

17.4.22.2 Description

The Receive_And_Scatter_Data_Unit procedure is used in connectionless mode to receive a data unit from another endpoint. Endpoint identifies the local communications endpoint through which data shall be received. The Address parameter (an XTI_Address_Pointer value) designates an object containing the address associated with the received data unit. The Options parameter (a Protocol_Option_List object) holds options associated with the received data unit. Vector identifies an IO_Vector_Array object, which is an array of POSIX_IO.IO_Vector objects with Buffer and Length attributes (*i.e.*, an array of buffer address/buffer size pairs). See 17.4.1.9 for a detailed description of these objects, including the appropriate attribute access operations. Procedure Receive_And_Scatter_Data_Unit shall receive data into these buffers, always filling each buffer with the number of octets specified by its Length attribute before proceeding to the next.

The number of POSIX_IO.IO_Vector objects to receive is implicit in the array or array slice specified by the Vector parameter. The POSIX_IO.IO_Vector objects shall be received starting from Vector'First. Octets_Received contains the number of octets of data received. If the complete data unit was received, Flags shall be returned as the empty option set. Otherwise, Flags shall be returned as a nonempty option set.

The Options Buffer attribute of Options is set before this procedure is called to indicate the buffer to be used to store options associated with the received data. The Buffer and Length attributes of the IO_Vector_Array referenced by Vector are set before Receive_And_Scatter_Data_Unit is called to define the buffers where the application data are to be placed (see 17.4.1.9). If the size of the Options Buffer attribute of Options is set to zero, then no information shall be returned in the Options Buffer attribute for this parameter.

NOTE: The total size of the buffers passed may be constrained by implementation limits. In practice, the availability of memory to an application is likely to impose a limit on the amount of data that can be sent or received using scatter/gather operations.

If the value of the Address parameter is nonnull, it shall be a conversion to type XTI_Address_Pointer of a value that designates an object of a protocol-specific address type allocated by the application before the call (see 17.4.1.3). Alternately, the Null_XTI_Address constant may be passed in Address causing this parameter to be ignored.

On return from this operation, the address object designated by Address shall contain the protocol address of the sending application and Options shall identify options that were associated with this data unit. The application data received shall be located in the buffers described by Vector. Octets_Received shall contain the number of octets of data received by the application.

If POSIX_IO.Non_Blocking is clear (*via* Open or Set_File_Control), Receive_And_Scatter_Data_Unit shall wait for a data unit to arrive (or for an error to be detected by the communications provider or for a signal to arrive) if none is currently available. If POSIX_IO.Non_Blocking is set, Receive_And_Scatter_Data_Unit shall fail if no data units are available.

If the buffers referenced by `Vector` are not large enough to hold the current data unit, the buffers shall be filled, and `More_Data` shall be set in `Flags` on return to indicate that another `Receive_And_Scatter_Data_Unit` should be called to retrieve the rest of the data unit. On subsequent calls to `Receive_And_Scatter_Data_Unit` (until this full data unit has been received), an empty `Protocol_Option_List` (as from `Make_Empty`) shall be returned, and the `Address` parameter shall be undefined.

If `Receive_And_Scatter_Data_Unit` is interrupted by a signal, it shall raise the exception `POSIX_Error` with error code `Interrupted_Operation`. Any incoming data unit (datagram) shall not be discarded and shall remain available to be returned by a subsequent call to `Receive_And_Scatter_Data_Unit`.

17.4.22.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Illegal_Data_Range`

The length of the array specified by `Vector` was greater than the limit specified by `POSIX_Limits.XTI_IO_Vector_Maxima'Last`.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Buffer_Not_Large_Enough`

The amount of data to be returned in one of the buffers is greater than the size of the buffer.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`No_Data_Available`

In nonblocking mode, data are not available.

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

Protocol_Error

A communication problem has occurred, and there is no other appropriate error number.

17.4.23 Receive a Data Unit

17.4.23.1 Synopsis

```
procedure Receive_Data_Unit
   (Endpoint         : in     POSIX_IO.File_Descriptor;
    User_Data        : in     System.Address;
    Octets_Requested : in     POSIX.IO_Count;
    Octets_Received  : out    POSIX.IO_Count;
    Address          : in     XTI_Address_Pointer;
    Options          : in out Protocol_Option_List;
    Flags            : out XTI_Flags);
procedure Receive_Data_Unit
   (Endpoint         : in  POSIX_IO.File_Descriptor;
    User_Data        : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Address          : in  XTI_Address_Pointer;
    Flags            : out XTI_Flags);
```

17.4.23.2 Description

The Receive_Data_Unit procedure is used in connectionless mode to receive a data unit from another endpoint. The Endpoint parameter identifies the local communications endpoint through which data shall be received. The User_Data parameter identifies the buffer where the received data unit shall be stored. The Address parameter (an XTI_Address_Pointer value) designates where the address information associated with the received data unit shall be stored. The Options parameter (a Protocol_Option_List object) indicates where option information associated with the received data unit shall be stored. Flags may be set on return to indicate that the complete data unit was not received. Octets_Received contains the number of octets of data received.

The Options Buffer attribute of Options shall be set before calling this procedure to indicate the buffer to be used. If the Options Buffer attribute designates a null Octet_Array or is the null access value, then no information shall be returned for Options. The Options parameter may be omitted using the overloaded procedure.

If the value of the Address parameter is nonnull, it shall be a conversion to type XTI_Address_Pointer of a value that designates an object of a protocol-specific address type allocated by the application before the call (see 17.4.1.3). Alternately, the Null_XTI_Address constant may be passed in Address causing this parameter to be ignored.

On return from this operation, the the address object designated by Address contains the protocol address of the sending application, Options identifies options that were associated with this data unit, and User_Data specifies the application data that were received.

If POSIX_IO.Non_Blocking is clear (via Open or Set_File_Control), Receive_Data_Unit shall wait for a data unit to arrive (or for an error to be detected by the

communications provider or for a signal to arrive) if none is currently available. If POSIX_IO.Non_Blocking is set, Receive_Data_Unit shall fail if no data units are available.

If the buffer defined in the User_Data parameter is not large enough to hold the current data unit, the buffer shall be filled, and More_Data shall be set in Flags on return to indicate that another Receive_Data_Unit should be called to retrieve the rest of the data unit. On subsequent calls to Receive_Data_Unit (until this full data unit has been received), an empty Protocol_Option_List (as from Make_Empty) shall be returned, and the Address parameter shall be undefined.

If Receive_Data_Unit is interrupted by a signal, it shall raise POSIX_Error with error code Interrupted_Operation. Any incoming data unit (datagram) shall not be discarded and shall remain available to be returned by a subsequent call to Receive_Data_Unit.

17.4.23.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_File_Descriptor
: A bad file descriptor was specified for the communications endpoint.

Buffer_Not_Large_Enough
: The amount of data to be returned in one of the buffers is greater than the buffer size. The unit data information to be returned shall be discarded.

Event_Requires_Attention
: An asynchronous event is outstanding on the communications endpoint specified by Endpoint. (The application can call Look to retrieve the event.)

No_Data_Available
: In nonblocking mode, data are not available.

XTI_Operation_Not_Supported
: This operation is not supported by the current implementation of XTI.

Operation_Not_Valid_For_State
: The operation was called with the communications provider in the wrong state (bad sequence).

Protocol_Error
: A communication problem has occurred, and there is no other appropriate error number.

17.4.24 Retrieve a Unit Data Error Indication

17.4.24.1 Synopsis

```
type Unit_Data_Error_Code is implementation-defined-integer;
procedure Retrieve_Data_Unit_Error
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Address  : in     XTI_Address_Pointer;
    Options  : in out Protocol_Option_List;
    Error    : out Unit_Data_Error_Code);
procedure Retrieve_Data_Unit_Error
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Address  : in  XTI_Address_Pointer;
    Error    : out Unit_Data_Error_Code);
procedure Retrieve_Data_Unit_Error
   (Endpoint : in     POSIX_IO.File_Descriptor;
    Options  : in out Protocol_Option_List;
    Error    : out    Unit_Data_Error_Code);
procedure Retrieve_Data_Unit_Error
   (Endpoint : in  POSIX_IO.File_Descriptor;
    Error    : out Unit_Data_Error_Code);
```

17.4.24.2 Description

The `Retrieve_Data_Unit_Error` procedure is used in connectionless mode to receive information concerning an error on a previously sent data unit and should only be issued following a unit data error (as indicated by the event `Error_In_Previously_Sent_Datagram` retrieved by `Look`). It informs the application that a data unit with a specific destination address and protocol options produced an error.

The `Endpoint` parameter identifies the local communications endpoint through which the error report shall be received. The `Address` parameter (an `XTI_Address_Pointer` value) designates where the address information associated with the received data unit error shall be stored. The `Options` parameter (a `Protocol_Option_List` object) indicates where option information associated with the received data unit error shall be stored. The `Error` parameter (of type `Unit_Data_Error_Code`) returns the error code associated with the received data unit error.

The Options Buffer attribute of the `Options` parameter shall be set before calling this function to indicate the buffer to be used to store the options. This parameter may be omitted. If the size of the Options Buffer is set to zero, then no information shall be returned in the Options Buffer attribute for this parameter.

On return from this operation, the object designated by the `XTI_Address_Pointer` value given by the `Address` parameter identifies the destination protocol address of the erroneous data unit, the `Protocol_Option_List` object identified by the `Options` parameter identifies options that were associated with the data unit, and the `Error_Code` parameter specifies a protocol-specific error code. `Retrieve_Data_Unit_Error` shall clear the unit data error indication.

17.4.24.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Buffer_Not_Large_Enough`
: The amount of data to be returned in one of the buffers is greater than the buffer size. The unit data error information to be returned shall be discarded.

`XTI_Operation_Not_Supported`
: This operation is not supported by the current implementation of XTI.

`No_Unit_Data_Error_On_Endpoint`
: No unit data error indication currently exists on the specified communications endpoint.

`Operation_Not_Valid_For_State`
: The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

17.4.25 Retrieve Information from Disconnect

17.4.25.1 Synopsis

```
procedure Retrieve_Disconnect_Info
   (Endpoint         : in  POSIX_IO.File_Descriptor;
    User_Data        : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Retrieved : out POSIX.IO_Count;
    Reason           : out Reason_Code;
    Sequence_Number  : out Natural);
procedure Clear_Disconnect_Info
   (Endpoint : in POSIX_IO.File_Descriptor);
```

17.4.25.2 Description

The `Retrieve_Disconnect_Info` procedure is used to identify the cause of a disconnection and to retrieve any application data sent with the disconnection request. The `Clear_Disconnect_Info` procedure is used to clear a disconnect request from the interface. `Endpoint` identifies the local communications endpoint where the connection existed. `Reason` specifies the reason for the disconnection through a protocol-specific reason code. `User_Data` identifies any application data that were sent with the disconnection request. `Sequence_Number` may identify an outstanding connection indication with which the disconnection is associated. The amount of application data requested is specified in the `Octets_Requested` parameter. `Octets_Retrieved` indicates the number of octets of application data returned on success.

`Sequence_Number` is only meaningful when `Retrieve_Disconnect_Info` is issued by a passive application that has executed one or more `Listen` functions and is processing the resulting connection indications. If a disconnection indication occurs, `Sequence_Number` can be used to identify which of the outstanding connection indications is associated with the disconnection.

If the `User_Data` parameter is `System.Null_Address`, no information shall be returned. If `Octets_Requested` is greater than zero and less than the length of the application data, `Retrieve_Disconnect_Info` shall fail with the error `Buffer_Not_Large_Enough`.

If `Clear_Disconnect_Info` is used, then no information shall be returned, and any application data associated with the disconnection indication shall be discarded. If an application has retrieved more than one outstanding connection indication (via `Listen`) and `Clear_Disconnect_Info` is used, then it follows that the application will be unable to identify with which connection indication the disconnection is associated.

This procedure shall be used from the Data Transfer, Outgoing Connect, Outgoing Release, Incoming Release, or Incoming Connect (when Outstanding Connection Count is greater than zero) state.

17.4.25.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`
: A bad file descriptor was specified for the communications endpoint.

`Buffer_Not_Large_Enough`
: The buffer allocated for incoming application data is not large enough to handle the data. If `Endpoint` is a passive endpoint, it remains in Incoming Connect state, otherwise, the state of the endpoint is set to Idle.

`No_Disconnect_Indication_On_Endpoint`
: There are no disconnect indications on the communications endpoint specified by `Endpoint`.

`XTI_Operation_Not_Supported`
: This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`
: The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`
: A communication problem has occurred, and there is no other appropriate error number.

17.4.26 Send Data or Expedited Data Over a Connection

17.4.26.1 Synopsis

```
procedure Send
   (Endpoint       : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Flags          : in  XTI_Flags;
    Octets_Sent    : out POSIX.IO_Count);
```

17.4.26.2 Description

The Send procedure is used to send either normal or expedited data. The parameter Endpoint identifies the local communications endpoint over which data should be sent, Buffer points to the application data, Octets_To_Send specifies the number of octets of application data to be sent. Flags specifies any of the following optional flags.

Expedited_Data

If set, the data shall be sent as expedited data and shall be subject to the interpretations of the communications provider.

More_Data

If set SDU or SEDU is being sent through multiple subprogram calls. Each subprogram call with the More_Data flag set indicates that another subprogram call should follow with more data for the current SDU or SEDU.

The end of the SDU (or SEDU) is identified by a subprogram call with the More_Data flag not set. Use of More_Data enables an application to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data are packaged for transfer. If the communications provider does not support the concept of SDU as indicated in the Info parameter on return from Open or Get_Info, the More_Data flag is not meaningful and shall be ignored if set.

The sending of a zero length fragment of an SDU or SEDU is only permitted where this is used to indicate the end of an SDU or SEDU, *i.e.*, when the More_Data flag is not set. Some communications providers also forbid zero length SDUs and SEDUs (See D.2)

Push_Data

If set, the communications provider shall send all data that are currently in its send buffers. If this flag is not set, the behavior shall be protocol-specific.

NOTE: The communications provider is free to collect data in a send buffer until it accumulates a sufficient amount for transmission (see Send Low Water Mark XTI option in 17.4.18).

If POSIX_IO.Non_Blocking is clear (via Open or Set_File_Control), Send shall return immediately if there is a permanent failure condition, but can wait if there is a transient error condition or if flow control restrictions prevent the data from being accepted by the local communications provider at the time the call is made. If POSIX_IO.Non_Blocking is set, Send shall fail immediately if there is a permanent or transient failure condition, or there are flow control restrictions. The process can

arrange to be informed when the flow control restrictions are cleared via either Look or the event management operations defined in 19.1 (or another suitable interface offered by the implementation).

On successful completion, Send returns the number of octets accepted by the communications provider in Octets_Sent. Normally this value shall equal the number of octets specified in Octets_To_Send. However, if POSIX_IO.Non_Blocking is set or the operation is interrupted by a signal, it is possible that only part of the data have been accepted by the communications provider. In this case, Send shall return a value in Octets_Sent that is less than the value of Octets_To_Send. If Send is interrupted by a signal before it could transfer data to the communications provider, it shall raise POSIX_Error with error code Interrupted_Operation.

If Octets_To_Send is zero and sending of zero octets is not supported by the underlying communications service, Send shall raise POSIX_Error with error code Illegal_Data_Range.

If Octets_To_Send is greater than the size of Buffer, Send shall raise POSIX_Error with error code Invalid_Argument.

The size of each SDU or SEDU shall not exceed the limits of the communications provider as specified by the current values in the Max Size SDU or Max Size SEDU attributes of the Communications_Provider_Info object returned by Get_Info.

The error Event_Requires_Attention may be returned to inform the process that an event (*e.g.*, a disconnection) is outstanding.

This procedure shall be used from the Data Transfer state or the Incoming Release state.

17.4.26.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_Argument

: The application specified a value for Octets_To_Send greater than the size of Buffer. For this error, the implementation under some circumstances may instead raise Constraint_Error.

Illegal_Data_Range

: The application attempted to send an illegal amount of data. The probable cause is that an attempt was made to send a SDU or SEDU larger than the size of SDU or SEDU returned by the Communications_Provider_Info in Open (or Get_Info), or an attempt was made to send a zero length data unit (or fragment). The ability of an XTI implementation to detect such an error case is implementation dependent.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Invalid_Flag`

The application specified an invalid flag.

`Flow_Control_Error`

Flow conditions prevented the sending of data at this time in asynchronous (nonblocking) mode.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.27 Send a Data Unit

17.4.27.1 Synopsis

```
procedure Send_Data_Unit
   (Endpoint       : in POSIX_IO.File_Descriptor;
    User_Data      : in System.Address;
    Octets_To_Send : in POSIX.IO_Count;
    Address        : in XTI_Address_Pointer;
    Options        : in Protocol_Option_List);
procedure Send_Data_Unit
   (Endpoint       : in POSIX_IO.File_Descriptor;
    User_Data      : in System.Address;
    Octets_To_Send : in POSIX.IO_Count;
    Address        : in XTI_Address_Pointer);
```

17.4.27.2 Description

The `Send_Data_Unit` procedure is used in connectionless mode to send a data unit to another communications application. The `Endpoint` parameter identifies the local communications endpoint through which data shall be sent. The `Address` parameter (an `XTI_Address_Pointer` value) designates an object that specifies the protocol

address of the destination application. The Options parameter (a Protocol_Option_List object) identifies options that the application wants associated with this request. The User_Data parameter specifies the application data to be sent.

The application may choose not to specify what protocol options are associated with the transfer by providing an empty option list in the Options parameter (using the Make_Empty procedure on the Protocol_Option_List object), or the Options parameter may be omitted using the overloaded procedure. In this case, the provider shall use the option values currently set for this communications endpoint.

If the Octets_To_Send parameter is zero and sending of zero octets is not supported by the underlying communications service, Send_Data_Unit shall raise POSIX_Error with error code Illegal_Data_Range.

If POSIX_IO.Non_Blocking is clear (via Open or Set_File_Control), Send_Data_Unit can wait if flow control restrictions prevent the data from being accepted by the local communications provider at the time the call is made. However, if POSIX_IO.Non_Blocking is set, Send_Data_Unit shall fail under such conditions. The process can arrange to be notified of the clearance of a flow control restriction via either Look or the event management operations defined in 19.1 (or another suitable interface offered by the implementation).

If the amount of data specified in the Octets_To_Send parameter exceeds the maximum SDU size as returned in the Max Size SDU attribute of the Communications_Provider_Info parameter of Open or Get_Info, an Illegal_Data_Range error shall be generated. If Send_Data_Unit is called before the destination application has activated its communications endpoint (see 17.4.5), the data unit may be discarded.

If it is not possible for the communications provider to immediately detect the conditions that cause the errors Illegal_Data_Range and Incorrect_Or_Illegal_Option, these errors shall alternatively be returned by Retrieve_Data_Unit_Error. Therefore, an application shall be prepared to receive these errors in either of these ways.

If Send_Data_Unit is interrupted by a signal, it shall raise the exception POSIX_Error with error code Interrupted_Operation, and the data unit (datagram) shall not have been sent.

17.4.27.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Incorrect_Address_Format

The address specified by the application contained an incorrect protocol address or was in an incorrect format.

`Illegal_Data_Range`

The application attempted to send an illegal amount of data.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Incorrect_Or_Illegal_Option`

The option specified by the application contained incorrect information, or the information was in an incorrect format.

`Flow_Control_Error`

Flow conditions prevented the sending of data at this time in asynchronous (nonblocking) mode.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.28 Send Application-Initiated Disconnection Request

17.4.28.1 Synopsis

```
procedure Send_Disconnect_Request
   (Endpoint : in POSIX_IO.File_Descriptor;
    Call     : in Connection_Info);
procedure Send_Disconnect_Request
   (Endpoint : in POSIX_IO.File_Descriptor);
procedure Send_Disconnect_Request
   (Endpoint        : in POSIX_IO.File_Descriptor;
    Sequence_Number : in Natural);
procedure Send_Disconnect_Request
   (Endpoint       : in POSIX_IO.File_Descriptor;
    User_Data      : in System.Address;
    Octets_To_Send : in POSIX.IO_Count);
```

17.4.28.2 Description

The procedure `Send_Disconnect_Request` is used to initiate an abortive release on an already established connection or to reject a connection request. The parameter `Endpoint` identifies the local communications endpoint of the connection, and `Call` specifies information associated with the abortive release. The application can initiate this request either during data transfer or during connection establishment. `Call` is a `Connection_Info` object containing the attributes: Options (a `Protocol_Option_List` object), User Data, Address (an `XTI_Address_Pointer` value), and Sequence Number.

The values in `Call` have different semantics, depending on the context of the call to `Send_Disconnect_Request`.

— When rejecting a connection request, `Call` shall contain a valid value of Sequence Number to uniquely identify the rejected connection indication to the communications provider. The Sequence Number attribute is only meaningful if the connection is in the Incoming Connect state. The Address and Options attributes of `Call` are ignored. Ignoring these attributes can also be accomplished by using the overloaded version of the procedure that requires only a `Sequence_Number` parameter.

— In all other cases, `Call` need only be used when data are being sent with the disconnection request. The Address, Options, and Sequence attributes of the `Call` object are ignored. Ignoring these attributes can also be accomplished by using the overloaded version of the procedure that requires only a `Data` parameter. If the application does not wish to send data to the remote application, `Call` may be omitted.

The User Data attribute of the `Call` parameter (or the `User_Data` and `Octets_To_Send` parameter) specifies the application data to be sent to the remote application. The amount of application data shall not exceed the limits supported by the communications provider, as indicated in the Max Size Disconnect Data value returned in the `Communications_Provider_Info` object from `Open` or `Get_Info`. If the length of the User Data attribute (or the `Octets_To_Send` parameter) is zero, no data shall be sent to the remote application.

This procedure shall be used from the Data Transfer, Outgoing Connect, Outgoing Release, Incoming Release, or Incoming Connect (when Outstanding Connection Count is greater than zero) states.

17.4.28.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Illegal_Data_Range`

The application attempted to send an illegal amount of data.

Unless range checks are suppressed, some conditions that correspond to `Illegal_Data_Range` may be caught first by the normal Ada-language range check, which raises `Constraint_Error` for out-of-range values. However, if the application circumvents the Ada language range checks, the system may still catch the error, in which case it shall raise `POSIX_Error` with `Illegal_Data_Range`.

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint.

`Invalid_Sequence_Number`

The application specified an incorrect sequence number or in the case of a connect request being rejected, `Call` was wrongly omitted. Some outbound data queued for this endpoint may be lost.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`XTI_Operation_Not_Supported`

This operation is not supported by the current implementation of XTI.

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

17.4.29 Synchronize Communications Endpoint

17.4.29.1 Synopsis

```
function Synchronize_Endpoint
   (Endpoint : in POSIX_IO.File_Descriptor)
  return Interface_State;
```

17.4.29.2 Description

For the communications endpoint specified by `Endpoint`, `Synchronize_Endpoint` synchronizes the data structures associated with the XTI implementation with information from the underlying communications provider. In doing so, it can convert an uninitialized file descriptor (obtained via `POSIX_IO.Open` or `POSIX_IO.Duplicate`, or as a result of a `POSIX_Process_Primitives.Start_Process` and `POSIX_Process_Primitives.Exec`) to an initialized communications endpoint, assuming that the file descriptor referenced a communications endpoint, by updating and allocating the necessary data structures. This operation also allows two cooperating processes to synchronize their interaction with a communications provider.

For example, if a process creates a new process and issues an Exec, the new process needs to issue a Synchronize_Endpoint to build the private data structure associated with a communications endpoint and to synchronize the data structure with the relevant provider information.

The function shall fail if executed while the communications provider is undergoing a state change.

The communications provider treats all applications using a communications endpoint as a single application. If multiple processes are using the same endpoint, they need to coordinate their activities so they do not violate the state of the communications endpoint. Synchronize_Endpoint returns the current state of the communications endpoint to the application, thereby enabling the application to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the state of the endpoint after a Synchronize_Endpoint is issued.

Synchronize_Endpoint returns the communications interface state of the communications provider (type Interface_State) as defined in 17.2.1.

This procedure may be used from all states except Uninitialized.

17.4.29.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception POSIX_Error shall be raised with error code Operation_Not_Valid_For_State.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Invalid_File_Descriptor
: A bad file descriptor was specified for the communications endpoint. This error may be returned when the Endpoint has been previously closed or when an erroneous number may have been passed to the call.

Protocol_Error
: A communication problem has occurred, and there is no other appropriate error number.

State_Change_In_Progress
: The communications provider is undergoing a state change.

17.4.30 Disable a Communications Endpoint

17.4.30.1 Synopsis

```
procedure Unbind
   (Endpoint : in POSIX_IO.File_Descriptor);
```

17.4.30.2 Description

The procedure `Unbind` disables the communications endpoint specified by `Endpoint` that was previously bound by `Bind`. On completion of this operation, no further data or events destined for this communications endpoint shall be accepted by the communications provider. An endpoint that is disabled by using `Unbind` can be enabled by a subsequent call to `Bind`.

This procedure shall be used from the Idle state.

17.4.30.3 Error Handling

If an operation is called from the wrong XTI state (see Table 17.6), the exception `POSIX_Error` shall be raised with error code `Operation_Not_Valid_For_State`.

In addition to the XTI error conditions listed below, operating system service or other implementation defined general error conditions can also occur (see 17.1.7).

If any of the following XTI error conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_File_Descriptor`

A bad file descriptor was specified for the communications endpoint. This error may be returned when the `Endpoint` has been previously closed or when an erroneous number may have been passed to the call.

`Event_Requires_Attention`

An asynchronous event is outstanding on the communications endpoint specified by `Endpoint`. (The application can call `Look` to retrieve the event.)

`Operation_Not_Valid_For_State`

The operation was called with the communications provider in the wrong state (bad sequence).

`Protocol_Error`

A communication problem has occurred, and there is no other appropriate error number.

c

Section 18: Detailed Network Interface - Socket

This section specifies services that provide the Sockets protocol-independent process-to-process communication interface, as defined in package `POSIX_Sockets`.

18.1 Introduction

A *socket* is an endpoint for communication using the facilities described in this section. A pair of sockets constitutes a bi-directional communication channel that enables a process to exchange data with another process, called a *peer*, locally and over networks. Sockets may be created in pairs or given names (addresses) in order to exchange data with another socket in a communications domain. A socket is accessed via a file descriptor obtained when the socket is created. A socket exists only as long as some process holds a file descriptor referencing it.

A socket is created with a specific socket type (see 18.1.4) and protocol (see 18.1.2). The socket type and protocol determine whether the socket is connection-mode or connectionless-mode.

In a connection-oriented protocol, sockets are characterized as being either active or passive. An active socket initiates a connection with a peer socket at some known address. A passive socket is bound to an address, which is usually associated with some network service, and placed in a listening mode to accept connections from active peers.

NOTE: In client/server architectures, the server socket is passive, and clients sockets are active. In a connectionless protocol, either side can play the server or client role.

The socket and the supporting protocol implementation maintain state information about the socket, addressing information, and protocol-independent and protocol-specific information as described in this section and D.1.

18.1.1 Protocol Families

Each network protocol is associated with a specific protocol family. A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family normally comprises a number of protocols. Each protocol is characterized by an abstract socket type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

18.1.2 Protocols

A protocol supports one of the socket abstractions detailed in 18.1.4. A specific protocol may be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one specific address type (see 18.1.4), usually determined by the addressing object inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol-specific.

All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide nonstandard facilities or extensions to a mechanism. C

Protocol mappings to the Internet and ISO protocols are described in D.1.

18.1.3 Addressing

A socket address specifies the local or remote portion of an endpoint association. Socket addresses are represented by the general type `Socket_Address_Pointer`, described in 18.4.1.2. Protocol-specific socket address objects are described in D.1.

18.1.4 Socket Types

A socket has an abstract *socket type* which is specified when the socket is created. The socket type, together with the protocol family, allows the selection of an appropriate communication protocol for use with the socket.

Four types are defined: `Stream_Socket`, `Sequenced_Packet_Socket`, `Datagram_Socket`, and `Raw_Socket`.

The `Stream_Socket` type provides reliable, sequenced, full-duplex octet streams between the socket and a peer to which the socket is connected. A socket of type `Stream_Socket` does not exchange any data with its peer until it enters the Connected state. Record boundaries are not maintained; data sent on a stream socket using output operations of one size may be received using input operations of smaller or larger sizes without loss of data. Data may be buffered; successful return from an output function does not imply that the data have been delivered to the peer or even transmitted from the local system. If data cannot be successfully transmitted within a given time (see Socket Send Timeout socket option in 18.3), then the connection is considered broken, and subsequent operations shall fail. `POSIX_Signals.Signal_Pipe_Write` is raised if a process sends on a broken stream (one that is no longer connected). Support for an out-of-band data transmission facility is protocol-specific.

The `Sequenced_Packet_Socket` socket type is similar to the `Stream_Socket` type and is also connection-oriented. The only difference between these types is that record boundaries are maintained using the `Sequenced_Packet_Socket` type. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers parts of more than one record. Record boundaries are visible to the receiver via an `End_Of_Message` flag in the received message flags returned by the `Receive_Message` procedure. It is protocol-specificwhether a maximum record size is imposed.

The `Datagram_Socket` socket type supports connectionless data transfer that is not necessarily acknowledged or reliable. Each call to an output operation may specify a different destination peer , and incoming datagrams may be received from multiple sources. The source address of each datagram is available when receiving the datagram. An application may also prespecify a peer address, in which case calls to output functions shall send to the prespecified peer. If a peer has been specified, only datagrams from that peer are received. A datagram is sent in a single output operation and received in a single input operation. The maximum size of a datagram is protocol-specific; with some protocols, the limit is implementation defined. Output datagrams may be buffered within the system; thus, a successful return from an output operation does not guarantee that a datagram will actually be sent or received. However, implementations should attempt to detect any errors possible before the

return of an output function, reporting any error by raising `POSIX_Error` with the appropriate error code.

The `Raw_Socket` socket type is similar to the `Datagram_Socket` type. It differs in that it is normally used with communication providers that underlie those used for the other socket types. For this reason the creation of a socket with type `Raw_Socket` shall require appropriate privilege. The format of datagrams sent and received with this socket type generally include specific protocol headers, and the formats are protocol-specific.

18.2 Events and States

18.2.1 Events

The state of a socket changes due to the occurrence of events. Events are caused by operations invoked by the application, by input from the network, and internally by the protocol machines. The set of events that can occur on a socket are protocol-specific; *i.e.*, not all events can occur with all protocols. The complete list of events and their definitions is in Table 18.1. See D.1 for specific events for each protocol.

18.2.2 State

A socket and the underlying protocol engine supporting the socket both have a substantial amount of state. The state of the socket depends on the type of protocol (connection-mode, connectionless-mode) and on the specific protocol in use. In general, this protocol engine state includes addressing information and connection state (for connection-mode protocols). In some states, data may be sent and/or received, and in other states data transfer may not be possible in one or both directions. In addition, data buffering and flow control state may be managed by the socket and the protocol. A generalized socket state diagram is presented in this clause; the specific states and transitions utilized by individual protocols may be subsets of this state diagram. Separate subclauses in D.1 for each protocol define the state diagram and transitions for that protocol.

The socket states and their definitions are listed in Table 18.2.

18.2.3 Connection-Mode Sockets

Figure 18.1 shows the generalized state diagram for connection-mode sockets (those with type `Stream_Socket` or `Sequenced_Packet_Socket`). Each state is represented by a box. The events that cause state transitions are indicated by labels along each arrow. The events and their definitions are listed in Table 18.1.

The operations listed in the figure shall be called only in the states shown. In the Null state, no socket exists. It can be reached from any other state by closing the socket.

NOTE: If the socket is open in another process, it may not transition to the Null state.

A socket is first created in the Ground state.

The Connecting state is reached when a call to the `Connect` procedure initiates an attempt by the protocol engine to connect to another endpoint. The endpoint remains

Table 18.1 – Socket Events

Event	Definition
Create	Successful return from `Create`.
Bind	Successful return from `Bind`.
Close	Successful return from `Close`.
Connect	A successful blocking `Connect`, or successful return from a nonblocking `Connect` in connection-mode.
Prespecify	Successful return from `Specify_Peer` in connectionless-mode.
Connect Failure	Failure of an attempt to connect using a connection-mode socket.
Unspecify	Successful return from `Unspecify_Peer`, in connectionless-mode
Receive Connection	Protocol-specific event indicating that a pending connection attempt has completed successfully.
Accept Connection	A blocking `Accept_Connection`, or a successful return from a nonblocking `Accept_Connection`.
Data Transfer	Successful return from `Read`, `Receive`, or `Receive_Message` where the number of octets received is greater than zero. Successful return from `Send`, `Send_Message`, or `Write` where the number of output octets is greater than zero.
Send Confirmation	Successful return from `Send_Message` specifying ancillary data only. Successful return from `Set_Connection_Confirmation_Data`. (This option is specific to the ISO Transport Protocol. See D.1.) Receipt of an inbound connection open request where the protocol automatically accepts the connection. This case causes a blocking `Accept_Connection` to succeed. Call to `Select_File` or `Poll` on the socket (19.1.1 and 19.1.2).
Read	Successful return from `Read`.
Receive	Successful return from `Receive`.
Receive Message	Successful return from `Receive_Message`.
Send	Successful return from `Send`.
Send Message	Successful return from `Send_Message`.
Write	Successful return from `Write`.
Shutdown0	Successful return from `Shutdown` with `Further_Receives_Disallowed`, or a protocol-specific event indicating that no additional data will be received.
Shutdown1	Successful return from `Shutdown` with `Further_Sends_Disallowed`, or a protocol-specific event indicating that no additional data can be sent.
Shutdown2	Successful return from `Shutdown` with `Further_Sends_And_Receives_Disallowed`, or a protocol-specific event such as connection close or abort.
Shutdown3	Successful return from `Shutdown` with `Further_Sends_Disallowed` or `Further_Sends_And_Receives_Disallowed`, or a protocol-specific event indicating that no additional data will be sent.
Shutdown4	Successful return from `Shutdown` with `Further_Receives_Disallowed` or `Further_Sends_And_Receives_Disallowed`, or a protocol-specific event indicating that no additional data will be received.

in this state until the protocol engine reports either success or failure in establishing the connection. The Connected state is entered if the connection attempt is successful, and the Failed state is entered if the attempt is unsuccessful.

Table 18.2 – Socket States

State	Definition
Null	In this state the socket is not initialized.
Ground	In this state the socket has just been created.
Bound	In this state the socket has a local socket address associated with it.
Connecting	In this state the socket is in the process of trying to open a connection to a specified destination.
Listening	In this state the socket is waiting for an inbound connection open request to be received.
Confirming	In this state the socket is deciding whether to accept an inbound connection open request.
Failed	In this state an outbound connection open has failed.
Connected	In this state an inbound open has been accepted or an outbound open has succeeded. Data can be transferred in both directions.
Sending Only	In this state the peer has indicated that it will not send any more data. Output data can still be sent on the socket.
Receiving Only	In this state the local application has indicated that it is done sending data. Input data can still be received on the socket.
Dead	In this state input and output cannot be sent on the socket. The linger option is relevant in this state.
Open	In this state, the remote socket address is fixed for both input and output operations.

When a connection indication is received for a socket in the Listening state, a new socket is created. The new socket is used for the new connection, and the original socket remains in the Listening state. The descriptor for the new socket is made available to the application via the `Accept_Connection` procedure. In the Confirming state the connection request has been neither confirmed nor rejected. It is protocol-specificwhether the new socket is returned in the Confirming state or whether the connection is confirmed and completed before the new descriptor is made available.

Not all states are necessarily supported by each protocol. The Confirming state and the Connecting state are not used in some protocols. Applications that do not use protocol-specificfeatures cannot depend on the existence of these states, as the next logical state is entered immediately. For example, a socket returned by the `Accept_Connection` function is either in the Confirming state or the Connected state. An application not using protocol-specificcalls to confirm or reject the connection would initiate I/O on the descriptor, advancing the socket to the Connected state.

Another example is that many implementations of the `Local_Protocol` IPC protocol either complete or reject connection attempts, even when `POSIX_IO.Non_Blocking` is set. The Failed state may be the same as the Ground state for some protocols or implementations; applications should not use any call other than the `Close` procedure in this state. The Sending Only state and the Receiving Only state do not necessarily cause any change in the protocol state, and thus the state transition may not be transmitted to the remote endpoint of the connection. The interface specification for each connection-mode protocol in D.1 notes any differences between this state diagram and that for the protocol and specifies the protocol actions taken when the `Shutdown` procedure is called.

Although not shown in the diagram, a socket in the Confirming, Connected, Sending Only

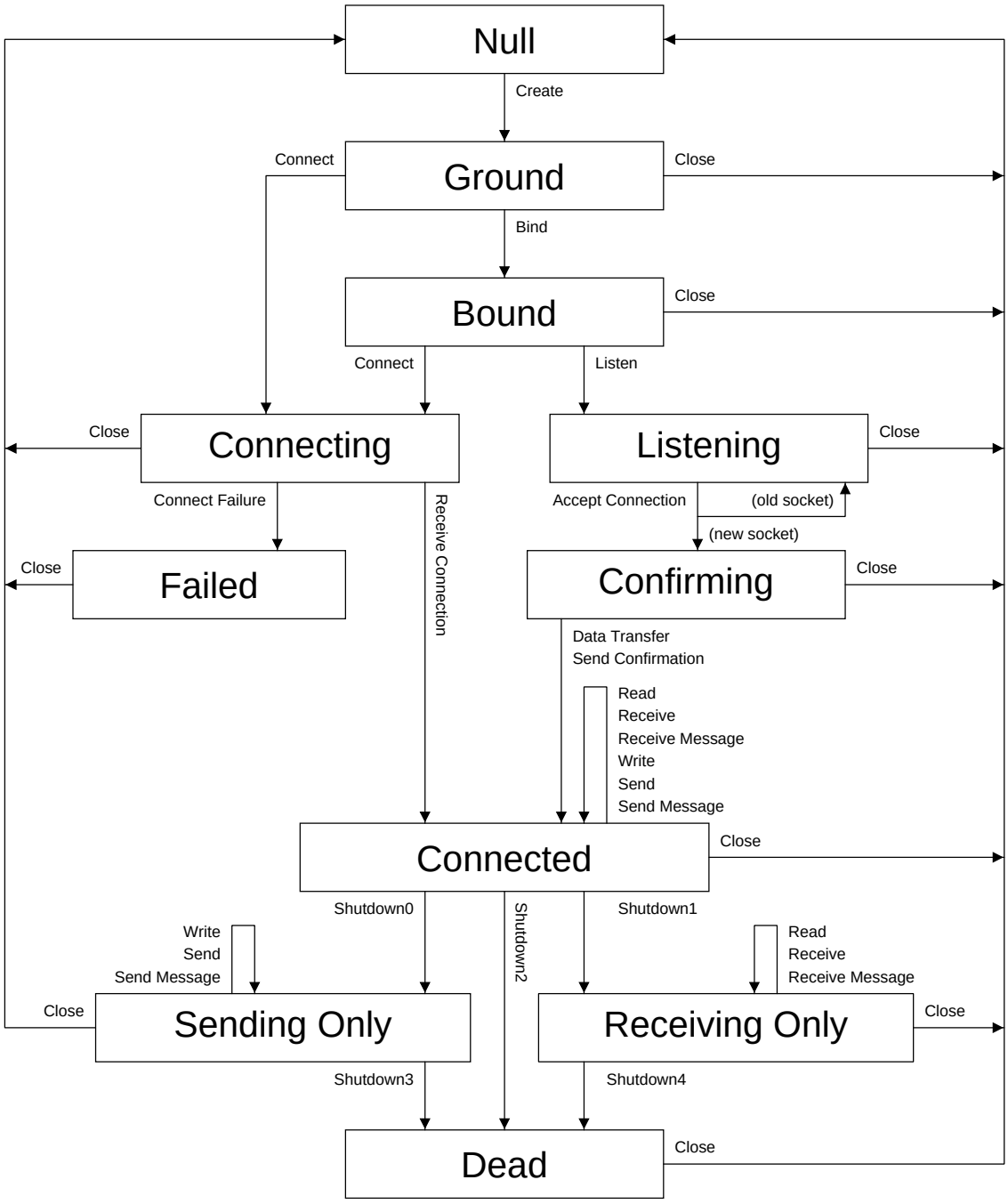

Figure 18.1 – Connection-Mode Sockets State Diagram

or Receiving Only states can transition to the Dead state by actions of the protocol. If a connection is aborted by the protocol or the remote endpoint, the socket enters the Dead state.

18.2.4 Connectionless-Mode Sockets

Figure 18.2 shows a generalized state diagram for connectionless-mode sockets (sockets with type `Datagram_Socket`), using the same notation as in Figure 18.1. The specification for each datagram protocol in D.1 notes any differences between this state diagram and that for the protocol. Figure 18.2 also illustrates the state diagram for sockets with type `Raw_Socket`. However, all uses of `Raw_Socket` sockets are protocol-specific, and specific protocols may use completely different states. It is protocol-specificwhether a call to `Send` or `Send_Message` in the Ground state causes a transition to the Bound state.

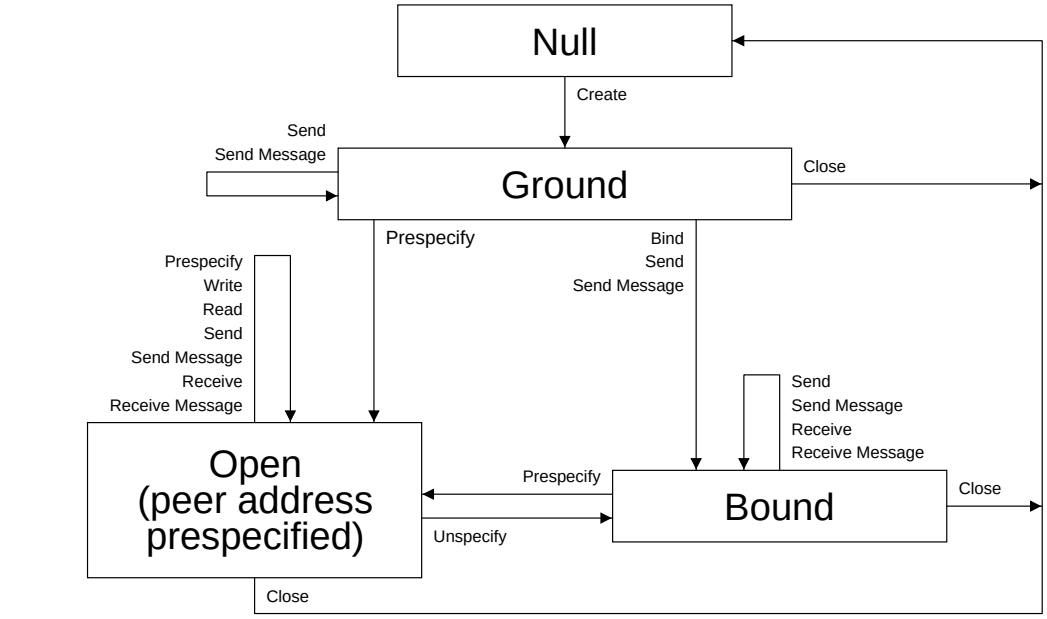

Figure 18.2 – Connectionless-Mode Sockets State Diagram

18.2.5 Socket State Elements

A socket has several state elements that are independent of the protocol, although the use of this state may depend on the protocol. The state of a socket includes the following items:

— I/O mode

— Owner (process or process group)

— Receive and transmit queue limits

— Error status

— Queued data that have been received

— Out-of-band state

— Connection status

— For listening connection-mode sockets, a queue of connection indications

— Socket options, defined in 18.3

18.2.5.1 Socket I/O Mode

The I/O mode of a socket is described by two file status flags. The two flags pertain to the open file description for the socket. Both flags are initially off when a socket is created, but they may be set and cleared by the use of the `POSIX_IO.Set_File_Control` procedure. The two flags are `POSIX_IO.Non_Blocking` and `POSIX_IO.Signal_When_Socket_Ready`.

When the `POSIX_IO.Non_Blocking` flag is set, functions that would normally block until they are complete either return immediately with an error, or they complete asynchronously to the execution of the calling process. Data transfer operations (the `Read`, `Write`, `Send`, and `Receive` functions) shall complete immediately, transfer less than requested, and then return without blocking; or they shall return an error indicating that no transfer could be made without blocking. The `Connect` procedure initiates a connection and returns without blocking when `POSIX_IO.Non_Blocking` is set; it returns the error `Already_Awaiting_Connection` to indicate that the connection was initiated successfully, but that it has not yet completed.

The socket is in signal-driven mode when the `POSIX_IO.Signal_When_Socket_Ready` flag is set. This mode is generally used with `POSIX_IO.Non_Blocking` set. In signal-driven mode, `POSIX_Signals.Signal_IO` is sent to the owner of the socket whenever an I/O operation becomes possible (when additional data could be sent, when new data arrive to be received, or a state transition such as connection establishment or error detection would allow a `Send` or `Receive` call to return status without blocking).

18.2.5.2 Socket Owner

The owner of a socket is unset when a socket is created. The owner may be set to a process ID or process group ID using the `POSIX_IO.Set_Socket_Process_Owner` and `POSIX_IO.Set_Socket_Group_Owner` procedures. The procedure `POSIX_IO.Get_Owner` returns the process owner or the process group owner for a socket.

18.2.5.3 Socket Queue Limits

The transmit and receive queue sizes for a socket are set when the socket is created. The default sizes used are both protocol-specificand implementation defined. The sizes may be changed using the socket options functions.

18.2.5.4 Pending Error

Errors may occur asynchronously and be reported to the socket in response to input from the network protocol. The socket stores the pending error to be reported to an application using the socket at the next opportunity. The error is returned in response to a subsequent `Send`, `Receive`, or socket option operation on the socket, and the pending error is then cleared.

18.2.5.5 Socket Receive Queue

A socket has a receive queue that buffers data when they are received by the system until they are removed by a receive call. Depending on the type of the socket and the

communication provider, the receive queue may also contain ancillary data such as the addressing and other protocol data associated with the normal data in the queue and may contain out-of-band or expedited data. (*Expedited data* are the mechanism used to bypass the flow control of normal data in the ISO protocols, and *out-of-band data* are data associated with an out-of-band signal in the Internet protocols.) The limit on the queue size includes any normal data, out-of-band data, datagram source addresses and ancillary data in the queue. The description in this subclause applies to all sockets, even though some elements cannot be present in some instances.

The contents of a receive buffer are logically structured as a series of data segments with associated ancillary data and other information. A data segment may contain normal data or out-of-band data, but never both. A data segment may complete a record if the protocol supports records (always true for types `Sequenced_Packet_Socket`, and `Datagram_Socket`). A record may be stored as more than one segment. The complete record might never be present in the receive buffer at one time, as a portion might already have been returned to the application and another portion might not yet have been received from the communications provider. A data segment may contain ancillary protocol data that are logically associated with the segment.

Ancillary data are received as if they were queued along with the first normal data octet in the segment (if any). It is possible for a segment to have ancillary data only. For the purposes of this subclause, a datagram is considered to be a data segment that terminates a record and that includes a source address as a special type of ancillary data. Data segments are placed into the queue as data are delivered to the socket by the protocol. Normal data segments are placed at the end of the queue as they are delivered. If a new segment contains the same type of data as the preceding segment and includes no ancillary data and if the preceding segment does not terminate a record, the segments are logically merged into a single segment.

The receive queue is logically terminated if an end-of-file indication has been received or a connection has been terminated.

A segment shall be considered to be terminated if another segment follows it in the queue, if the segment completes a record, or if an end-of-file or other connection termination has been reported. The last segment in the receive queue shall also be considered to be terminated while the socket has a pending error to be reported.

A receive operation shall never return data or ancillary data from more than one segment.

18.2.5.6 Socket Out-of-Band Data State

The handling of received out-of-band data is protocol-specific. Out-of-band data may be placed in the socket receive queue, either at the end of the queue or before all normal data in the queue. In this case, out-of-band data are returned to an application program by a normal `Receive` call. Out-of-band data may also be queued separately rather than being placed in the socket receive queue, in which case they shall be returned only in response to a `Receive` call that requests out-of-band data. It is protocol-specific whether an out-of-band data mark is placed in the receive queue to demarcate data preceding the out-of-band data and following the out-of-band data. An out-of-band data mark is logically an empty data segment that cannot be merged

with other segments in the queue. An out-of-band data mark is never returned in response to an input operation. The `Socket_Is_At_OOB_Mark` function can be used to test whether an out-of-band data mark is the first element in the queue. If an out-of-band data mark is the first element in the queue when `Read`, `Receive`, or `Receive_Message` is called without the `Peek_Only` option, the mark is removed from the queue and the following data (if any) are processed as if the mark had not been present.

18.2.5.7 Socket Connection Status

A connection-mode socket and/or its supporting protocol engine maintain generic connection status. This status includes indications that

— A connection has been initiated.

— A connection has been established.

— An end-of-file indication has been received.

— A disconnect has been initiated.

— A disconnect has been completed.

— No additional data may be sent due to protocol connection shutdown.

D.1 specifies for each protocol the events that cause each indication to be set.

18.2.5.8 Connection Indication Queue

Sockets that are used to accept incoming connections maintain a queue of outstanding connection indications. This queue is a list of connections that are awaiting acceptance by the application. The maximum size of the queue is constrained to be the lower of the backlog parameter set by the `Listen` function and the `Connection_Queue_Length_Maximum` constant. If the protocol returns sockets in the Connected state rather than the Confirming state via the `Accept_Connection` procedure, the implementation may include incomplete connections in the queue subject to the queue limit. The implementation may also increase the specified queue limit internally if it includes such incomplete connections in the queue subject to this limit.

18.2.6 Signals

One category of event at the socket interface is the generation of signals. These signals report protocol events or process errors relating to the state of the socket. The generation or delivery of a signal does not change the state of the socket, although the generation of the signal may have been caused by a state change.

`POSIX_Signals.Signal_Pipe_Write` shall be sent to a process that attempts to send data on a socket that is no longer able to send. In addition, the send operation fails with the error `Broken_Pipe`.

If a socket has an owner, `POSIX_Signals.Signal_Out_Of_Band_Data` is sent to the owner of the socket when it is notified of expedited or out-of-band data. The socket state at this time is protocol-specific, and the status of the socket is specified in D.1.1 for each protocol. Depending on the protocol, the expedited data may or may not have arrived at the time of signal generation.

If a socket is in signal-driven mode and has an owner, `POSIX_Signals.Signal_IO` is sent to the owner of the socket when new data arrive at the socket, when additional data may be sent on the socket due to a change in flow control, when a pending error is set for the socket, or when a change of state makes further send or receive operations impossible. A signal shall be also sent when a call to the `Accept_Connection` procedure would succeed without blocking and when it becomes possible to write data to a connection that was being established asynchronously. The procedures `Poll` and `Select_File` can be used for event management (see 19.1.1 and 19.1.2).

18.2.7 Asynchronous Errors

If any of the following conditions occur asynchronously for a socket, the corresponding value listed below shall become the pending error for the socket:

`Connection_Aborted`
: The connection was aborted locally.

`Connection_Refused`
: For a connection-mode socket attempting a nonblocking connection, the attempt to connect was forcefully rejected.

  For a connectionless-mode socket, an attempt to deliver a datagram was forcefully rejected.

`Connection_Reset`
: The peer has aborted the connection.

`Host_Down`
: The destination host has been determined to be down or disconnected.

`Host_Unreachable`
: The destination host is not reachable.

`Message_Too_Long`
: For a connectionless-mode socket, the size of a previously sent datagram prevented delivery.

`Network_Down`
: The local network connection is not operational.

`Network_Reset`
: The connection was aborted by the network.

`Network_Unreachable`
: The destination network is not reachable.

`Timed_Out`
: The connection timed out during or after connection establishment.

18.3 Use of Options

A number of socket options either specialize the behavior of a socket or provide useful information. Some options are protocol independent; these are described in 18.4.9 along with the functions and procedures that manipulate them. Others are protocol-specific; these options and their associated functions and procedures are described in D.1.

All of the options have default values. The type and meaning of these values is defined by the protocol to which they apply. Instead of using the default values, an application program may choose to customize one or more of the options. However, in the bulk of cases, the default values are sufficient for the application.

Some of the options are used to enable or disable certain behavior within the protocol modules (*e.g.*, turn on debugging) while others may be used to set protocol-specificinformation (*e.g.*, IP time-to-live on all the outgoing packets of the application).

18.4 Package POSIX_Sockets

The package POSIX_Sockets provides the Sockets protocol-independent process-to-process communications interface.

The functionality described in this clause is optional. If the Sockets Detailed Network Interface option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this clause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this clause.

```
with POSIX,
     POSIX_IO,
     POSIX_Limits,
     System;
package POSIX_Sockets is
   -- 18.4.1 Common Data Types and Constants
   -- 18.4.1.1 Socket Types and Protocols
   type Socket_Type is range implementation-defined;
   Stream_Socket           : constant Socket_Type := impl-def-static-expression;
   Datagram_Socket         : constant Socket_Type := impl-def-static-expression;
   Raw_Socket              : constant Socket_Type := impl-def-static-expression;
   Sequenced_Packet_Socket : constant Socket_Type := impl-def-static-expression;
   Unspecified_Socket_Type : constant Socket_Type := impl-def-static-expression;
   type Protocol_Family is range implementation-defined;
   Unspecified_Protocol_Family :
     constant Protocol_Family := impl-def-static-expression;
   type Protocol_Number is range implementation-defined;
   Default_Protocol : constant Protocol_Number := impl-def-static-expression;
   -- 18.4.1.2 Socket Addresses
   type Socket_Address_Pointer is private;
   Null_Socket_Address : constant Socket_Address_Pointer;
   -- 18.4.1.3 Socket Messages
   type Socket_Message is private;
   subtype IO_Vector_Range is Positive range
      1 .. POSIX_Limits.Socket_IO_Vector_Maxima'Last;
```

```
type IO_Vector_Array is array
   (IO_Vector_Range range <>) of POSIX_IO.IO_Vector;
procedure Set_Socket_Name
   (Message : in out Socket_Message;
    Name    : in     Socket_Address_Pointer);
type IO_Vector_Array_Pointer is access all IO_Vector_Array;
procedure Set_IO_Vector_Array
   (Message : in out Socket_Message;
    Pointer : in     IO_Vector_Array_Pointer);
function Get_IO_Vector_Array (Message : Socket_Message)
  return IO_Vector_Array_Pointer;
type Message_Option_Set is new POSIX.Option_Set;
Peek_Only           : constant Message_Option_Set := implementation-defined;
Process_OOB_Data    : constant Message_Option_Set := implementation-defined;
Wait_For_All_Data   : constant Message_Option_Set := implementation-defined;
Do_Not_Route        : constant Message_Option_Set := implementation-defined;
type Message_Status_Set is new POSIX.Option_Set;
Received_OOB_Data   : constant Message_Status_Set := implementation-defined;
End_Of_Message      : constant Message_Status_Set := implementation-defined;
Message_Truncated   : constant Message_Status_Set := implementation-defined;
Ancillary_Data_Lost : constant Message_Status_Set := implementation-defined;
procedure Set_Message_Options
   (Message : in out Socket_Message;
    Options : in     Message_Option_Set);
function Get_Message_Status
   (Message : Socket_Message)
  return Message_Status_Set;
procedure Set_Ancillary_Data
   (Message : in out Socket_Message;
    Data    : in     System.Address;
    Length  : in     POSIX.IO_Count);
procedure Get_Ancillary_Data
   (Message : in  Socket_Message;
    Data    : out System.Address;
    Length  : out POSIX.IO_Count);
-- 18.4.2 Dequeue a Connection Indication on a Socket
procedure Accept_Connection
   (Socket            : in  POSIX_IO.File_Descriptor;
    Connection_Socket : out POSIX_IO.File_Descriptor;
    Name              : in  Socket_Address_Pointer);
function Accept_Connection
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_IO.File_Descriptor;
-- 18.4.3 Bind a Socket Address to a Socket
procedure Bind
   (Socket : in POSIX_IO.File_Descriptor;
    Name   : in Socket_Address_Pointer);
-- 18.4.4 Initiate a Connection on a Socket
procedure Connect
   (Socket : in POSIX_IO.File_Descriptor;
    Peer   : in Socket_Address_Pointer);
procedure Specify_Peer
   (Socket : in POSIX_IO.File_Descriptor;
    Peer   : in Socket_Address_Pointer);
procedure Unspecify_Peer
   (Socket : in POSIX_IO.File_Descriptor);
```

```
-- 18.4.5 Create an Endpoint for Communication
function Create
   (Domain   : Protocol_Family;
    Of_Type  : Socket_Type;
    Protocol : Protocol_Number := Default_Protocol)
  return POSIX_IO.File_Descriptor;
-- 18.4.6 Create a Pair of Connected Sockets
procedure Create_Pair
   (Peer1    : out POSIX_IO.File_Descriptor;
    Peer2    : out POSIX_IO.File_Descriptor;
    Domain   : in  Protocol_Family;
    Of_Type  : in  Socket_Type;
    Protocol : in  Protocol_Number := Default_Protocol);
-- 18.4.7 Get Socket Address Information
type Socket_Address_Info is limited private;
type Socket_Address_Info_List is limited private;
procedure Make_Empty
   (Info_Item : in out Socket_Address_Info_List);
type Address_Flags is new POSIX.Option_Set;
Use_For_Binding : constant Address_Flags := implementation-defined;
Canonical_Name  : constant Address_Flags := implementation-defined;
procedure Set_Flags
   (Info_Item : in out Socket_Address_Info;
    Flags     : in     Address_Flags);
function Get_Flags (Info_Item : Socket_Address_Info)
  return Address_Flags;
procedure Set_Family
   (Info_Item : in out Socket_Address_Info;
    Family    : in     Protocol_Family);
function Get_Family (Info_Item : Socket_Address_Info)
  return Protocol_Family;
procedure Set_Socket_Type
   (Info_Item : in out Socket_Address_Info;
    To        : in     Socket_Type);
function Get_Socket_Type (Info_Item : Socket_Address_Info)
  return Socket_Type;
procedure Set_Protocol_Number
   (Info_Item : in out Socket_Address_Info;
    Protocol  : in     Protocol_Number);
function Get_Protocol_Number (Info_Item : Socket_Address_Info)
  return Protocol_Number;
function Get_Canonical_Name (Info_Item : Socket_Address_Info)
  return POSIX.POSIX_String;
procedure Get_Socket_Address_Info
   (Name    : in     POSIX.POSIX_String;
    Service : in     POSIX.POSIX_String;
    Info    : in out Socket_Address_Info_List);
procedure Get_Socket_Address_Info
   (Name    : in     POSIX.POSIX_String;
    Service : in     POSIX.POSIX_String;
    Request : in     Socket_Address_Info;
    Info    : in out Socket_Address_Info_List);
generic
   with procedure Action
      (Info : in     Socket_Address_Info;
       Quit : in out Boolean);
procedure For_Every_Item (List : in Socket_Address_Info_List);
```

```
   -- 18.4.8 Get Socket Information
   function Get_Socket_Error_Status
      (Socket : POSIX_IO.File_Descriptor)
     return POSIX.Error_Code;
   function Get_Socket_Type
      (Socket : POSIX_IO.File_Descriptor)
     return Socket_Type;
   -- 18.4.9 Get and Set Options on Sockets
   type Socket_Option_Value is (Enabled, Disabled);
   function Get_Socket_Broadcast
      (Socket : POSIX_IO.File_Descriptor)
     return Socket_Option_Value;
   procedure Set_Socket_Broadcast
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in Socket_Option_Value);
   function Get_Socket_Debugging
      (Socket : POSIX_IO.File_Descriptor)
     return Socket_Option_Value;
   procedure Set_Socket_Debugging
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in Socket_Option_Value);
   function Get_Socket_Routing
      (Socket : POSIX_IO.File_Descriptor)
     return Socket_Option_Value;
   procedure Set_Socket_Routing
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in Socket_Option_Value);
   function Get_Socket_Keep_Alive
      (Socket : POSIX_IO.File_Descriptor)
     return Socket_Option_Value;
   procedure Set_Socket_Keep_Alive
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in Socket_Option_Value);
   subtype Linger_Time is POSIX.Seconds range 0 .. POSIX.Seconds'Last;
   function Get_Socket_Linger_Time
      (Socket : POSIX_IO.File_Descriptor)
     return Linger_Time;
   procedure Set_Socket_Linger_Time
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in Linger_Time);
   function Get_Socket_OOB_Data_Inline
      (Socket : POSIX_IO.File_Descriptor)
     return Socket_Option_Value;
   procedure Set_Socket_OOB_Data_Inline
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in Socket_Option_Value);
   function Get_Socket_Receive_Buffer_Size
      (Socket : POSIX_IO.File_Descriptor)
     return POSIX.IO_Count;
   procedure Set_Socket_Receive_Buffer_Size
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in POSIX.IO_Count);
   function Get_Socket_Receive_Low_Water_Mark
      (Socket : POSIX_IO.File_Descriptor)
     return POSIX.IO_Count;
   procedure Set_Socket_Receive_Low_Water_Mark
      (Socket : in POSIX_IO.File_Descriptor;
       To     : in POSIX.IO_Count);
```

```
function Get_Socket_Receive_Timeout
   (Socket : POSIX_IO.File_Descriptor)
  return Duration;
procedure Set_Socket_Receive_Timeout
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Duration);
function Get_Socket_Reuse_Addresses
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Option_Value;
procedure Set_Socket_Reuse_Addresses
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Option_Value);
function Get_Socket_Send_Buffer_Size
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.IO_Count;
procedure Set_Socket_Send_Buffer_Size
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.IO_Count);
function Get_Socket_Send_Low_Water_Mark
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.IO_Count;
procedure Set_Socket_Send_Low_Water_Mark
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.IO_Count);
function Get_Socket_Send_Timeout
   (Socket : POSIX_IO.File_Descriptor)
  return Duration;
procedure Set_Socket_Send_Timeout
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Duration);
-- 18.4.10 Determine Whether a File Descriptor Refers to a Socket
function Is_A_Socket (File : POSIX_IO.File_Descriptor)
  return Boolean;
-- 18.4.11 Listen for Connections on a Sockets
Connection_Queue_Length_Maximum : constant := impl-def-static-expression;
subtype Connection_Queue_Length is natural
   range 0 .. Connection_Queue_Length_Maximum;
procedure Listen
   (Socket  : in POSIX_IO.File_Descriptor;
    Backlog : in Connection_Queue_Length :=
                 Connection_Queue_Length'Last);
-- 18.4.12 Receive Data From a Socket
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Masked_Signals   : in  POSIX.Signal_Masking;
    Options          : in  Message_Option_Set := Empty_Set);
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Options          : in  Message_Option_Set := Empty_Set);
```

```
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    From             : in  Socket_Address_Pointer;
    Masked_Signals   : in  POSIX.Signal_Masking;
    Options          : in  Message_Option_Set := Empty_Set);
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    From             : in  Socket_Address_Pointer;
    Options          : in  Message_Option_Set := Empty_Set);
procedure Receive_Message
   (Socket          : in     POSIX_IO.File_Descriptor;
    Message         : in out Socket_Message;
    Octets_Received : out    POSIX.IO_Count;
    Masked_Signals  : in     POSIX.Signal_Masking;
    Options         : in     Message_Option_Set := Empty_Set);
procedure Receive_Message
   (Socket          : in     POSIX_IO.File_Descriptor;
    Message         : in out Socket_Message;
    Octets_Received : out    POSIX.IO_Count;
    Options         : in     Message_Option_Set := Empty_Set);
-- 18.4.13 Send Data Over a Socket
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    Masked_Signals : in  POSIX.Signal_Masking;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    To             : in  Socket_Address_Pointer;
    Masked_Signals : in  POSIX.Signal_Masking;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    To             : in  Socket_Address_Pointer;
    Options        : in  Message_Option_Set := Empty_Set);
```

```
   procedure Send_Message
      (Socket         : in  POSIX_IO.File_Descriptor;
       Message        : in  Socket_Message;
       Octets_Sent    : out POSIX.IO_Count;
       Masked_Signals : in  POSIX.Signal_Masking;
       Options        : in  Message_Option_Set := Empty_Set);
   procedure Send_Message
      (Socket      : in  POSIX_IO.File_Descriptor;
       Message     : in  Socket_Message;
       Octets_Sent : out POSIX.IO_Count;
       Options     : in  Message_Option_Set := Empty_Set);
   -- 18.4.14 Shut Down Part of a Full-Duplex Connection
   type Shutdown_Mode is
      (Further_Receives_Disallowed,
       Further_Sends_Disallowed,
       Further_Sends_And_Receives_Disallowed);
   procedure Shutdown
      (Socket : in POSIX_IO.File_Descriptor;
       Mode   : in Shutdown_Mode);
   -- 18.4.15 Determine Whether a Socket is at the Out-of-Band Mark
   function Socket_Is_At_OOB_Mark (Socket : POSIX_IO.File_Descriptor)
     return Boolean;
private
   implementation-defined
end POSIX_Sockets;
```

18.4.1 Common Data Types and Constants

18.4.1.1 Socket Types and Protocols

18.4.1.1.1 Synopsis

```
type Socket_Type is range implementation-defined;
Stream_Socket           : constant Socket_Type := impl-def-static-expression;
Datagram_Socket         : constant Socket_Type := impl-def-static-expression;
Raw_Socket              : constant Socket_Type := impl-def-static-expression;
Sequenced_Packet_Socket : constant Socket_Type := impl-def-static-expression;
Unspecified_Socket_Type : constant Socket_Type := impl-def-static-expression;
type Protocol_Family is range implementation-defined;
Unspecified_Protocol_Family :
  constant Protocol_Family := impl-def-static-expression;
type Protocol_Number is range implementation-defined;
Default_Protocol : constant Protocol_Number := impl-def-static-expression;
```

18.4.1.1.2 Description

A `Socket_Type` indicates what type of communication service semantics will be associated with operations on the socket. These values shall be distinct.

A `Protocol_Family` (see 18.1.1), together with a socket type, determine the protocol that is used for subsequent operations on the socket. Values of type `Protocol_Family` shall be distinct. The value `Unspecified_Protocol_Family` indicates an unspecified protocol family or any protocol family that can be used with a particular name and/or service (see 18.4.7 and D.1.3.2).

`Protocol_Number` specifies a particular protocol within a given protocol family when creating a socket with `Create` or `Create_Pair`. `Default_Protocol` is the default value of parameter `Protocol` of function `Create` (18.4.5) and procedure `Create_Pair` (18.4.6). It specifies the default protocol to be used for the given `Socket_Type` and `Protocol_Family`.

18.4.1.2 Socket Addresses

18.4.1.2.1 Synopsis

```
type Socket_Address_Pointer is private;
Null_Socket_Address : constant Socket_Address_Pointer;
```

18.4.1.2.2 Description

Socket addresses are described using the general object `Socket_Address_Pointer`, which references a protocol-specific network address. The attributes of protocol-specificsocket address types, the operations that manipulate them, and operations to convert protocol-specific socket address objects to and from the `Socket_Address_Pointer` type are described in D.1.

The constant `Null_Socket_Address` is a special value that does not refer to any address (*i.e.*, the null address). This constant is used as a paremeter to certain operations when the `Socket_Address_Pointer` object can optionally be omitted by the application.

18.4.1.3 Socket Messages

18.4.1.3.1 Synopsis

```
type Socket_Message is private;
subtype IO_Vector_Range is Positive range
   1 .. POSIX_Limits.Socket_IO_Vector_Maxima'Last;
type IO_Vector_Array is array
   (IO_Vector_Range range <>) of POSIX_IO.IO_Vector;
procedure Set_Socket_Name
   (Message : in out Socket_Message;
    Name    : in     Socket_Address_Pointer);
type IO_Vector_Array_Pointer is access all IO_Vector_Array;
procedure Set_IO_Vector_Array
   (Message : in out Socket_Message;
    Pointer : in     IO_Vector_Array_Pointer);
function Get_IO_Vector_Array (Message : Socket_Message)
  return IO_Vector_Array_Pointer;
type Message_Option_Set is new POSIX.Option_Set;
Peek_Only           : constant Message_Option_Set := implementation-defined;
Process_OOB_Data    : constant Message_Option_Set := implementation-defined;
Wait_For_All_Data   : constant Message_Option_Set := implementation-defined;
Do_Not_Route        : constant Message_Option_Set := implementation-defined;
type Message_Status_Set is new POSIX.Option_Set;
Received_OOB_Data   : constant Message_Status_Set := implementation-defined;
End_Of_Message      : constant Message_Status_Set := implementation-defined;
Message_Truncated   : constant Message_Status_Set := implementation-defined;
Ancillary_Data_Lost : constant Message_Status_Set := implementation-defined;
procedure Set_Message_Options
   (Message : in out Socket_Message;
    Options : in     Message_Option_Set);
```

```
function Get_Message_Status
   (Message : Socket_Message)
  return Message_Status_Set;
procedure Set_Ancillary_Data
   (Message : in out Socket_Message;
    Data    : in     System.Address;
    Length  : in     POSIX.IO_Count);
procedure Get_Ancillary_Data
   (Message : in  Socket_Message;
    Data    : out System.Address;
    Length  : out POSIX.IO_Count);
```

18.4.1.3.2 Description

The Send_Message and Receive_Message operations use an object called a Socket_Message to specify the location of multiple data segments that can be sent or received in a single operation. (These data segments are not related to the data segments of 18.2.5.5.) In addition, these operations may also be used to send and receive protocol-specificancillary data. The Socket_Message consolidates all of the necessary information into a single object, thus simplifying the parameters in the Send_Message and Receive_Message calls.

A Socket_Message object is made up of the following attributes:

Name

In a receive operation, the address of the sending entity on a connectionless socket. In a send operation, it is the address of the peer to which the message is to be sent. The application shall set this attribute before the message can be sent over a connectionless socket. If the socket destination address has been prespecified with the Connect or Specify_Peer procedure, this attribute shall be left unspecified when the message is sent. Get_Socket_Name (see D.1) shall return the protocol-specific address of the sender of a received message if the socket is in a connectionless state. In all other cases the returned socket address is indeterminate. Set_Socket_Name is used to set the desired recipient of a message on a send operation for a connectionless socket. Set_Socket_Name may also be used by the application to initialize the name for a receive operation, so that the identity of the sender will not be recorded. Normally, the name of the sender is saved and can be retrieved with Get_Socket_Name. Any error checking on the address is not done until the message is sent via Send_Message.

IO Vector Array

A pointer to an object with the type IO_Vector_Array containing elements that have the type POSIX_IO.IO_Vector, each of which points to an individual data segment. A socket message consists of a sequence of these data segments. For a Send operation, the IO_Vector_Array specifies what data are to be sent; in a receive operation, it specifies where the received data are to be placed. The Set_IO_Vector_Array procedure and the Get_IO_Vector_Array function set and retrieve the IO Vector Array attribute of a Socket_Message. The POSIX_IO.Set_Buffer procedure and the POSIX_IO.Get_Buffer function manipulate pointers of type System.Address for each individual data segment of the IO_Vector_Array and also set the Length of each segment.

Ancillary Data

A pointer to a contiguous sequence of octets that make up protocol-specificancillary data being sent or received with the socket message. The format and type of the ancillary data (also referred to as *control data*) is protocol-specific. A mechanism to manipulate raw ancillary data buffers is provided by the `Set_Ancillary_Data` and `Get_Ancillary_Data` procedures. Overloaded versions of these subprograms for protocol-specific ancillary data formats are provided in D.1.

An application can issue a `Receive_Message` operation requesting ancillary data only or a `Send_Message` operation that sends ancillary data only by specifying a `Socket_Message` object that has not had any data segments specified by `Set_IO_Vector_Array`. The same effect can be accomplished by performing a `POSIX_IO.Set_Buffer` operation for each `POSIX_IO.IO_Vector` specifying `System.Null_Address` for `Buffer` and a `Length` of zero.

Message Status

Information on attributes of the message, as well as any buffer truncation that may have occurred (see 18.4.12). This attribute is used on received messages only. `Send_Message` does not return a message status. This attribute is distinct from the `Options` parameter used on receive and send operations.

The following steps are needed to create and manage a `Socket_Message`:

— The application defines each segment of the socket message with an `POSIX_IO.IO_Vector` object using the `POSIX_IO.Set_Buffer` procedure.

— The segments are combined in an `IO_Vector_Array`. The application shall set the address of this array in the socket message with `Set_IO_Vector_Array`.

— If needed, the application also sets up the Ancillary Data, Message Flags, and Name attributes.

The constant `POSIX_Limits.Socket_IO_Vector_Maxima'Last` defines how many message segments are allowed in a socket message.

NOTE: Applications may use pointers to the `POSIX.Octet_Array` type to ensure proper data width for network I/O operations. When using pointers to other data types, byte width and ordering issues (i.e., big endian, little endian) are the responsibility of the application.

18.4.2 Dequeue a Connection Indication on a Socket

18.4.2.1 Synopsis

```
procedure Accept_Connection
   (Socket            : in  POSIX_IO.File_Descriptor;
    Connection_Socket : out POSIX_IO.File_Descriptor;
    Name              : in  Socket_Address_Pointer);
function Accept_Connection
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_IO.File_Descriptor;
```

18.4.2.2 Description

`Accept_Connection` is used with connection-oriented socket types.

`Accept_Connection` extracts the first connection request on the queue of pending connections for a listening socket. The parameter `Socket` is a file descriptor referring to a socket that has been created with `Create`, has been bound to a local socket address with `Bind`, and is listening for connections as a result of the application having invoked `Listen`. `Accept_Connection` creates a new socket with similar properties to the `Socket` parameter that was passed as input, except it is not listening for connection requests, but instead is associated with a specific connection or connection request. The new socket created for the connection is returned by function `Accept_Connection` or as the `Connection_Socket` output parameter by procedure `Accept_Connection`. If no pending connections are present on the queue and the socket is not marked as nonblocking, `Accept_Connection` blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, `Accept_Connection` returns an error as described below. The file descriptor for the `Connection_Socket` (or the socket returned) can not be used to accept more connections. The original socket descriptor `Socket` remains open.

Procedure `Accept_Connection` returns the socket address of the connecting peer, as known to the communications layer, in the object referenced by parameter `Name`. The parameter `Name` shall be associated with a `Socket_Address_Pointer` value that designates an address object of the specific type that corresponds to the type and protocol family of the socket. Otherwise, `Accept_Connection` may return an error. Whether this error is detected is implementation dependent.

It is possible to determine when a new request has been queued by selecting the descriptor `Socket` for reading using the `Select_File` or `Poll` procedures from `POSIX_Event_Management` (see 19.1.1 and 19.1.2).

For protocols that require an explicit confirmation of communication requirements, such as ISO Transport, one should think of `Accept_Connection` as merely dequeuing the next connection request and associating an endpoint with it; these actions do not imply confirmation. Confirmation can be implied by an attempt to send or receive normal data on the new file descriptor, and rejection can be implied by closing the new socket.

An application can obtain connection request data without confirming the connection by issuing a `Receive_Message` operation that requests ancillary data only. Similarly, an application can provide connection rejection information by issuing a `Send_Message` providing only the control information. (See the protocol-specificspecifications in D.1 for details.)

18.4.2.3 Error Handling

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Incorrect_Address_Type`

The type of the address object designated by the `Name` parameter is not of

the appropriate type for the address format of this socket.

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor
: Socket is not a valid descriptor.

Connection_Aborted
: The peer closed the connection before the application processed the request.

Invalid_Argument
: Socket is not in the Listening state.

Interrupted_Operation
: The operation was interrupted by a signal before a connection was available to be returned.

Too_Many_Open_Files
: The per-process descriptor table is full.

Too_Many_Open_Files_In_System
: The system file table is full.

No_Buffer_Space
: Insufficient resources were available in the system to perform the operation.

Not_A_Socket
: The file descriptor Socket does not refer to a socket.

Option_Not_Supported
: Socket does not support the Accept_Connection operation.

Would_Block
: Socket is marked as nonblocking and no connections are present to be accepted.

18.4.3 Bind a Socket Address to a Socket

18.4.3.1 Synopsis

```
procedure Bind
   (Socket : in POSIX_IO.File_Descriptor;
    Name   : in Socket_Address_Pointer);
```

18.4.3.2 Description

Bind shall associate a local socket address with a socket (previously created with Create or Create_Pair). When a socket is created, it is associated with a specific protocol from the protocol family, but has no local socket address assigned to it.

The Bind procedure requests that the local socket address specified by Name be assigned to the socket. The parameter Name shall be associated with a Socket_Address_Pointer value that designates an address object of a specific type that is supported by the protocol of the socket. Otherwise, Bind may return an error.

`Bind` assigns the default permissions of read, write, and execute by owner, group, and others to the socket. These default permissions may be modified in protocol-specificways.

The rules used in name binding vary between communication domains (see D.1).

18.4.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: The requested socket address is reserved and the calling process does not have appropriate privileges to access it.

`Address_In_Use`
: The specified socket address is already in use.

`Address_Not_Available`
: The specified socket address is not available from the local machine.

`Incorrect_Address_Type`
: The type of the designated address object is incorrect for this socket.

`Bad_File_Descriptor`
: `Socket` is not a valid descriptor.

`Invalid_Argument`
: The socket is already associated with a local socket address, or the address is not valid for the specified address type.

`No_Buffer_Space`
: Insufficient resources were available in the system to perform the operation.

`Not_A_Socket`
: The file descriptor `Socket` does not refer to a socket.

18.4.4 Initiate a Connection on a Socket

18.4.4.1 Synopsis

```
procedure Connect
   (Socket : in POSIX_IO.File_Descriptor;
    Peer   : in Socket_Address_Pointer);
procedure Specify_Peer
   (Socket : in POSIX_IO.File_Descriptor;
    Peer   : in Socket_Address_Pointer);
procedure Unspecify_Peer
   (Socket : in POSIX_IO.File_Descriptor);
```

18.4.4.2 Description

Invoking `Connect` shall cause a socket, specified by the file descriptor `Socket`, to be connected to a peer socket whose socket address is specified by the `Peer` parameter.

The `Peer` parameter shall designate an object having a specific type that is supported by the protocol of the socket (see D.1).

If the socket has not been bound to a local address and its protocol family is not `Local_Protocol` (see D.1.1), the socket shall be bound to a local address.

If the socket is a connection-mode socket (`Stream_Socket` or `Sequenced_Packet_Socket`), then this operation attempts to make a connection to the peer. For such sockets, this call is valid only in the Ground, and Bound states.

If the socket is a connectionless-mode socket (`Datagram_Socket` or `Raw_Socket`), the `Specify_Peer` operation specifies the peer with which the socket is to be associated, the socket address to which datagrams are to be sent, and the only socket address from which datagrams are to be received. Upon successful completion, the socket state shall be Open. Calling `Specify_Peer` on a connectionless-mode socket that is already in the Open state shall change the peer with which the socket is associated. This call is valid in all states for connectionless-mode sockets. The `Specify_Peer` operation is equivalent to a `Connect` operation with a protocol-specific socket address entered in the `Peer` parameter.

Procedure `Unspecify_Peer` dissolves the peer association of the socket. If the connectionless-mode socket was in the Open state before the call, `Unspecify_Peer` shall leave the socket in the Bound state if it was in the Bound state at any previous time. Otherwise, it is protocol-specificwhether the socket shall be in the Bound or Ground state. The `Unspecify_Peer` operation is equivalent to a `Connect` operation with the constant `Null_Socket_Address` entered for the `Peer` parameter.

If the socket is of type `Stream_Socket` and it has not been marked as nonblocking, `Connect` shall block until the connection has been established or an error is detected. The time period for which attempts to make the connection will continue is unspecified and depends upon the implementation and on the protocol. If the socket has been marked as nonblocking, `Connect` shall return immediately. A call to the `Select_File` or `Poll` function (19.1.1 and 19.1.2) shall return `True` for writing on the descriptor if the connection attempt has succeeded. If the connection attempt fails then a call to `Select_File` or `Poll` shall indicate that the file descriptor is ready for reading and is ready for writing. The error indicating the reason for failure becomes the pending error for the socket at the time of the failure (see 18.2.5.4).

If the socket is of type `Stream_Socket`, it has been marked as nonblocking, and the connection does not complete immediately, the operation shall return the `Operation_In_Progress` error code and the socket state shall change to Connecting.

If the operation fails for a connection-mode socket, the state of the socket shall be Failed.

Applications should explicitly `Close` the socket descriptor prior to attempting to reinitiate the connection.

18.4.4.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: The calling process does not have the appropriate privileges.

`Address_In_Use`
: The specified socket address is already in use.

`Address_Not_Available`
: The specified socket address is of a valid address type, but is otherwise invalid or not available.

`Incorrect_Address_Type`
: The type of the designated address object is incorrect for this socket.

`Already_Awaiting_Connection`
: The socket is nonblocking and a previous connection attempt has not yet been completed.

`Bad_File_Descriptor`
: `Socket` is not a valid descriptor.

`Connection_Refused`
: The attempt to connect was forcefully refused.

`Host_Unreachable`
: The host specified in the socket address is not reachable.

`Operation_In_Progress`
: The socket is nonblocking and the connection cannot be completed immediately.

`Interrupted_Operation`
: The operation was interrupted by a signal before a connection was completed.

`Invalid_Argument`
: The socket address is not valid for the specified address type.

`Is_Already_Connected`
: The socket is already connected.

`Network_Down`
: The local network connection is not operational.

`Network_Unreachable`
: The network is not reachable from this host.

`No_Buffer_Space`
: Insufficient resources were available in the system to perform the operation.

`Not_A_Socket`
: The file descriptor does not refer to a socket.

`Timed_Out`
: Connection establishment timed out without establishing a connection.

18.4.5 Create an Endpoint for Communication

18.4.5.1 Synopsis

```
function Create
   (Domain   : Protocol_Family;
    Of_Type  : Socket_Type;
    Protocol : Protocol_Number := Default_Protocol)
  return POSIX_IO.File_Descriptor;
```

18.4.5.2 Description

`Create` shall create an endpoint for communication and return a POSIX file descriptor.

`Domain` specifies the protocol family (see 18.1.1) of the socket and constrains specific types of socket address parameters in subsequent calls to operations on the socket. `Of_Type` specifies the type of socket that will be used in the communication.

Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol shall be specified in this manner. The protocol number to use is specific to the communication domain in which communication is to take place. When the value of the `Protocol` parameter is `Default_Protocol`, the default protocol as given in D.1 for the socket type and protocol family is used.

An implementation need not make an internal distinction between pipes and some types of sockets.

18.4.5.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: Permission to create a socket of the specified type and/or protocol is denied.

`Too_Many_Open_Files`
: The per-process descriptor table is full.

`Too_Many_Open_Files_In_System`
: The system file table is full.

`No_Buffer_Space`
: Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

`Protocol_Not_Supported`
: The protocol family is not supported, or the type or specified protocol is not supported within the protocol family.

`Socket_Type_Not_Supported`
: The type of scket specified is not supported.

18.4.6 Create a Pair of Connected Sockets

18.4.6.1 Synopsis

```
procedure Create_Pair
   (Peer1    : out POSIX_IO.File_Descriptor;
    Peer2    : out POSIX_IO.File_Descriptor;
    Domain   : in  Protocol_Family;
    Of_Type  : in  Socket_Type;
    Protocol : in  Protocol_Number := Default_Protocol);
```

18.4.6.2 Description

`Create_Pair` shall create an unnamed pair of sockets and return a pair of socket descriptors. If the sockets are of connection-mode, on return they shall be connected to each other and in the Connected state. If the sockets are of connectionless-mode, on return they shall each be in the Open state with the other socket as the specified peer.

`Domain` specifies the set of permissible address formats that can be supplied in later operations on the socket where a socket address is required. `Of_Type` specifies the type of socket that will be used in the communication.

When the value of the `Protocol` parameter is `Default_Protocol`, the default protocol as given in D.1 for the socket type and protocol family is used.

NOTE: `Create_Pair` is typically used to create a pair of connected sockets before a process creation operation. See the rationale (B.19.4) for further information.

18.4.6.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: Permission to create a socket of the specified type and/or protocol is denied.

`Too_Many_Open_Files`
: The per-process descriptor table is full.

`Too_Many_Open_Files_In_System`
: The system file table is full.

`No_Buffer_Space`
: Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

`Option_Not_Supported`
: The specified protocol does not support the creation of socket pairs.

`Protocol_Not_Supported`
: The protocol domain is not supported, or the type or specified protocol is not supported within the domain.

18.4.7 Get Socket Address Information

18.4.7.1 Synopsis

```
type Socket_Address_Info is limited private;
type Socket_Address_Info_List is limited private;
procedure Make_Empty
   (Info_Item : in out Socket_Address_Info_List);
type Address_Flags is new POSIX.Option_Set;
Use_For_Binding : constant Address_Flags := implementation-defined;
Canonical_Name  : constant Address_Flags := implementation-defined;
procedure Set_Flags
   (Info_Item : in out Socket_Address_Info;
    Flags     : in     Address_Flags);
function Get_Flags (Info_Item : Socket_Address_Info)
  return Address_Flags;
procedure Set_Family
   (Info_Item : in out Socket_Address_Info;
    Family    : in     Protocol_Family);
function Get_Family (Info_Item : Socket_Address_Info)
  return Protocol_Family;
procedure Set_Socket_Type
   (Info_Item : in out Socket_Address_Info;
    To        : in     Socket_Type);
function Get_Socket_Type (Info_Item : Socket_Address_Info)
  return Socket_Type;
procedure Set_Protocol_Number
   (Info_Item : in out Socket_Address_Info;
    Protocol  : in     Protocol_Number);
function Get_Protocol_Number (Info_Item : Socket_Address_Info)
  return Protocol_Number;
function Get_Canonical_Name (Info_Item : Socket_Address_Info)
  return POSIX.POSIX_String;
procedure Get_Socket_Address_Info
   (Name    : in     POSIX.POSIX_String;
    Service : in     POSIX.POSIX_String;
    Info    : in out Socket_Address_Info_List);
procedure Get_Socket_Address_Info
   (Name    : in     POSIX.POSIX_String;
    Service : in     POSIX.POSIX_String;
    Request : in     Socket_Address_Info;
    Info    : in out Socket_Address_Info_List);
generic
   with procedure Action
      (Info : in     Socket_Address_Info;
       Quit : in out Boolean);
procedure For_Every_Item (List : in Socket_Address_Info_List);
```

18.4.7.2 Description

The functionality described in this subclause is optional. If the Network Management *option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise* `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

The `Get_Socket_Address_Info` procedure shall translate the name of a service location (for example, a host name) and/or a service name and return a list of `Socket_Address_Info` objects (contained in a `Socket_Address_Info_List` object). This list contains a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The `Socket_Address_Info` object shall include at least the following attributes.

Flags

Input flags. The function `Get_Flags` shall return this attribute. The procedure `Set_Flags` shall set this attribute.

Family

Protocol family for socket. The function `Get_Family` shall return this attribute. The procedure `Set_Family` shall set this attribute.

Socket Type

The socket type. The function `Get_Socket_Type` shall return this attribute. The procedure `Set_Socket_Type` shall set this attribute.

Protocol Number

Protocol for socket. The function `Get_Protocol_Number` shall return this attribute. The procedure `Set_Protocol_Number` shall set this attribute.

Address

Socket address for socket. The protocol-specificfunction `Get_Address` shall return this attribute (see D.1).

Canonical Name

Canonical name for service location. The function `Get_Canonical_Name` shall return this attribute.

The parameters `Name` and `Service` shall either be a `POSIX_String` or an empty `POSIX_String`. These strings be descriptive names or can be addresses in Internet address dot notation as specified in D.1.3.2.2.

If `Name` is not an empty `POSIX_String`, the requested service location is named by `Name`; otherwise, the requested service location is local to the caller. The format of a valid name depends on the protocol family or families. If a family is not specified and the name could be interpreted as valid within multiple supported families, address information may be returned for multiple families.

If `POSIX_Options.Internet_Protocol_Support` is `True` and the specified protocol family is `POSIX_Sockets_Internet.Internet_Protocol` or `Unspecified_Protocol_Family`, valid names include host names as specified in {B21} and strings specifying an address using Internet standard dot notation as specified in D.1.3.2.2.

If `Service` is not an empty `POSIX_String`, it is the name of the requested service. If `Service` is an empty `POSIX_String`, the call shall return network-level addresses for the specified `Name`. It is an error for both `Name` and `Service` to be an empty `POSIX_String`.

`Service` shall be a `POSIX_String` identifying the service. This string can be either a descriptive name or a numeric representation suitable for use with the protocol family or families. For the latter to be valid the caller shall also specify value other than `Unspecified_Socket_Type` for the Socket Type attribute of `Request`.

If `POSIX_Options.Internet_Protocol_Support` is `True` and the specified protocol family is `POSIX_Sockets_Internet.Internet_Protocol` (see D.1.3) or `Unspecified_Protocol_Family`, the service can be specified as a string specifying a decimal port number.

If the parameter Request is used, it shall refer to an object containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, protocol family, and/or protocol.

The Flags attribute of the Request parameter shall denote a set of address flags. The operations "+", "-", ">", "<", ">=", "<=", and Empty_Set are available on the type Address_Flags via the derived type semantics of Ada, from the operations available for POSIX.Option_Set. The appropriate operations can be used to create and examine a set containing one or more of the values Use_For_Binding and Canonical_Name. If the flag Use_For_Binding is specified, the returned address information shall be suitable for use in binding a socket for accepting incoming connections for the specified service. Otherwise, the returned address information shall be suitable for creating a connection to the specified service. If the flag Canonical_Name is specified and the Name parameter is not an empty POSIX_String, the function shall attempt to determine the canonical name corresponding to Name (for example, if Name is an alias or short-hand notation for a complete name).

The Socket Type attribute of Request shall specify the socket type for the service, as defined in 18.1.4. A value of Unspecified_Socket_Type indicates that any socket type may be returned; any other value limits the returned information to values with the specified socket type.

If the Family attribute of the Request parameter has the value Unspecified_Protocol_Family, addresses shall be returned for use with any protocol family that can be used with the specified name and/or service. Otherwise, addresses shall be returned for use only with the specified protocol family (as defined in 18.4.1.2). If Family is not Unspecified_Protocol_Family and Protocol Number is not Default_Protocol, then addresses shall be returned for use only with the specified protocol family and protocol. The value of Protocol Number shall be interpreted as in a call to the Create procedure with the corresponding values of Protocol_Family and Protocol_Number.

The Make_Empty procedure shall remove all Socket_Address_Info objects from a Socket_Address_Info_List, freeing any dynamically allocated storage associated with the object.

Upon successful return, Get_Socket_Address_Info returns a list of Socket_Address_Info objects (contained in a Socket_Address_Info_List object), each of which specifies a socket address and information for use in creating a socket with which to use that socket address. The application program instantiates the generic procedure For_Every_Item with an actual procedure for the generic formal procedure Action. When called, the instance shall call the actual procedure supplied for Action once for each element in the associated list.

Action shall be able to force termination of the generic instance either by setting Quit to True or by raising an exception. Prior to calling Action, the instance shall set Quit to False. Exceptions raised by Action shall terminate iteration and shall be propagated back to the caller of the instance. After an exception is raised by Action or Action returns with Quit set to True, no more calls to Action shall occur.

Each object on the list includes values for use with a call to the Create procedure,

and a socket address for use with the Connect procedure or, if the Use_For_Binding flag was specified, for use with the Bind procedure. The attributes Family, Socket Type, and Protocol shall be usable as the parameters to the Create function to create a socket suitable for use with the returned address. The Address attribute shall be usable as a parameter to the Connect or Bind procedures with such a socket, according to the Use_For_Binding flag. If Name is not an empty POSIX_String, and if requested by the Canonical_Name flag, the Canonical Name attribute of the first returned Socket_Address_Info object shall be a POSIX_String containing the canonical name corresponding to the given Name; if the canonical name is not available, Canonical_Name shall equal the given Name. The content of the Flags attribute of the returned objects is undefined.

18.4.7.3 Error Handling

The following errors are defined for the Get_Socket_Address_Info function only.

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Unknown_Address_Type
: The address found for the name was of an unsupported type.

Try_Again
: The name could not be resolved at this time. Future attempts may succeed.

Invalid_Flags
: The Flags parameter had an invalid value.

Name_Failed
: A nonrecoverable error occurred when attempting to resolve the name.

Unknown_Protocol_Family
: The protocol family was not recognized.

Memory_Allocation_Failed
: There was a memory allocation failure when trying to allocate storage for the return value.

No_Address_For_Name
: A valid name was passed, but no address was associated with the name.

Name_Not_Known
: The name is not known.
: Neither name nor service were passed. At least one of these shall be passed.

Service_Not_Supported
: The service passed was not recognized for the specified socket type.

Unknown_Socket_Type
: The intended socket type was not recognized.

18.4.8 Get Socket Information

18.4.8.1 Synopsis

```
function Get_Socket_Error_Status
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.Error_Code;
function Get_Socket_Type
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Type;
```

18.4.8.2 Description

`Get_Socket_Error_Status` shall return any pending error on the socket as defined in 18.1.4, and shall clear the error status. It returns a value of zero if there is no pending error. This operation may be used to check for asynchronous errors on connectionless-mode sockets in the Open state or for other types of asynchronous errors.

`Get_Socket_Type` shall return the type of the socket (*e.g.*, `Stream_Socket`). This function is useful to servers that inherit sockets on startup.

The `Socket` parameter is an open file descriptor referring to a socket.

Protocol specific socket information operations (*e.g.*, `Get_Socket_Name`, which returns the name (address) associated with a socket, and `Get_Peer_Name`, which returns the socket address of the peer connected to a socket) are described in D.1.

18.4.8.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: The calling process does not have the appropriate privileges.

`Bad_File_Descriptor`
: `Socket` not a valid descriptor.

`No_Buffer_Space`
: Insufficient resources were available in the system to perform the operation.

`Not_A_Socket`
: The file descriptor `Socket` does not refer to a socket.

18.4.9 Get and Set Options on Sockets

18.4.9.1 Synopsis

```
type Socket_Option_Value is (Enabled, Disabled);
function Get_Socket_Broadcast
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Option_Value;
procedure Set_Socket_Broadcast
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Option_Value);
```

```
function Get_Socket_Debugging
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Option_Value;
procedure Set_Socket_Debugging
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Option_Value);
function Get_Socket_Routing
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Option_Value;
procedure Set_Socket_Routing
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Option_Value);
function Get_Socket_Keep_Alive
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Option_Value;
procedure Set_Socket_Keep_Alive
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Option_Value);
subtype Linger_Time is POSIX.Seconds range 0 .. POSIX.Seconds'Last;
function Get_Socket_Linger_Time
   (Socket : POSIX_IO.File_Descriptor)
  return Linger_Time;
procedure Set_Socket_Linger_Time
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Linger_Time);
function Get_Socket_OOB_Data_Inline
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Option_Value;
procedure Set_Socket_OOB_Data_Inline
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Option_Value);
function Get_Socket_Receive_Buffer_Size
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.IO_Count;
procedure Set_Socket_Receive_Buffer_Size
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.IO_Count);
function Get_Socket_Receive_Low_Water_Mark
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.IO_Count;
procedure Set_Socket_Receive_Low_Water_Mark
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.IO_Count);
function Get_Socket_Receive_Timeout
   (Socket : POSIX_IO.File_Descriptor)
  return Duration;
procedure Set_Socket_Receive_Timeout
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Duration);
function Get_Socket_Reuse_Addresses
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Option_Value;
procedure Set_Socket_Reuse_Addresses
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Option_Value);
function Get_Socket_Send_Buffer_Size
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.IO_Count;
```

```
procedure Set_Socket_Send_Buffer_Size
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.IO_Count);
function Get_Socket_Send_Low_Water_Mark
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.IO_Count;
procedure Set_Socket_Send_Low_Water_Mark
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.IO_Count);
function Get_Socket_Send_Timeout
   (Socket : POSIX_IO.File_Descriptor)
  return Duration;
procedure Set_Socket_Send_Timeout
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Duration);
```

18.4.9.2 Description

The following list identifies the protocol-independent options, the types of the option value parameters associated with each option, the functions and procedures used to manipulate the option, the default values for protocol-independent options, and a synopsis of the meaning of the option value parameter. D.1 provides information regarding socket options that are specific to particular protocols supported under the sockets interface.

Socket Broadcast

This option defines whether the socket has permission to send broadcast datagrams. Support for this option is protocol-specific(see D.1). `Get_Socket_Broadcast` shall return the state of the Socket Broadcast socket option for the socket. `Set_Socket_Broadcast` shall set the Socket Broadcast socket option for the socket to the value of the `To` parameter. The default value of this parameter is `Disabled`, and the option is only valid for sockets of type `Datagram_Socket`.

Socket Debugging

This option enables debugging in the underlying protocol modules, which can be useful for tracing the behavior of the underlying protocol modules during normal system operation. The semantics of the debug reports are implementation defined. `Get_Socket_Debugging` shall return the state of the Socket Debugging socket option for the socket. `Set_Socket_Debugging` sets the Socket Debugging socket option to the value specified by the `To` parameter. The default value for this option is `Disabled`.

Socket Routing

Setting this option to `Disabled` requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly connected network, and messages are directed to the appropriate network interface according to the destination address. It is protocol-specificwhether this option has any effect and how the outgoing network interface is chosen. Support for this option with each protocol is implementation defined. `Get_Socket_Routing` shall return the value of the Socket Routing socket option for the socket. `Set_Socket_Routing` sets the Socket Routing socket option for the socket to the value specified by the `To` parameter. The default value for this option is `Enabled`.

Socket Keep Alive

This option enables the periodic transmission of messages on a connected socket. The behavior of this option is protocol-specificand defined in D.1. `Get_Socket_Keep_Alive` shall return the state of the Socket Keep Alive socket option for the socket. `Set_Socket_Keep_Alive` shall set the value of the Socket Keep Alive socket option for a socket to the value specified by the `To` parameter. The default value for this option is `Disabled`.

Socket Linger Time

This option controls the action of the interface when unsent messages are queued on a socket and a `Close` is performed. The details of this option are protocol-specific. A positive value for the Socket Linger Time socket option indicates that the option is on and the socket will linger for the specified number of seconds to close out unsent data. A zero value indicates the Socket Linger Time socket option is off. `Get_Socket_Linger_Time` shall return the value of the Socket Linger Time socket option for the socket. The default value for this option is zero.

Socket OOB Data Inline

This option requests that out-of-band data be placed in the normal data input queue as received; they will then be accessible using the `Read` or `Receive` procedures without the `Process_OOB_Data` flag set. The Socket OOB Data Inline socket option is valid only on protocols that support out-of-band data. (See D.1 for protocol-specificdetails.) `Get_Socket_OOB_Data_Inline` shall return the state of the Socket OOB Data Inline socket option for the socket. `Set_Socket_OOB_Data_Inline` shall set the state of the option to the value specified by the `To` parameter. The default for this option is `Disabled`; that is, out-of-band data shall not be placed in the normal data input queue.

Socket Receive Buffer Size

This option specifies the maximum buffer size, in octets, for data received on this socket. This maximum buffer size is the receive queue limit noted in 18.2.5.3. Applications may wish to increase buffer size for high-volume connections or may decrease buffer size to limit the possible backlog of incoming data. `Get_Socket_Receive_Buffer_Size` shall return the size of the receive buffer for the socket in octets. `Set_Socket_Receive_Buffer_Size` shall set the size of the receive buffer to the value specified by the `To` parameter. The default value for this option is implementation dependent, and may vary by protocol. The maximum value for this option is the value returned by `POSIX_Configurable_File_Limits.Socket_Buffer_Maximum`.

Socket Receive Low Water Mark

This option specifies the minimum number of octets to process for socket input operations. In general, receive operations will block until any (nonzero) amount of data are received, then return the smaller of the amount available or the amount requested. The default value of 1 does not affect the general case. If this option is set to a larger value, blocking receive calls normally wait until they have received the smaller of the Socket Receive Low Water Mark socket option value or the requested amount. Receive calls may still return

less than the Socket Receive Low Water Mark socket option value if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned (*e.g.*, out of band data). It is implementation defined whether this option can be set. `Get_Socket_Receive_Low_Water_Mark` shall return the Socket Receive Low Water Mark socket option value for the socket. `Set_Socket_Receive_Low_Water_Mark` shall set the Socket Receive Low Water Mark socket option value for the socket to the value specified by the `To` parameter. The default value for this option is 1.

Socket Receive Timeout

This option specifies a timeout value for input operations. It accepts a parameter of type `Duration` specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or generates the error code `Would_Block` if no data were received. The default value of zero indicates that a receive operation shall not time out. It is implementation defined whether this option can be set. `Get_Socket_Receive_Timeout` shall return the Socket Receive Timeout socket option for the Socket. `Set_Socket_Receive_Timeout` shall set the Socket Receive Timeout socket option for the socket to the value specified by the `To` parameter. The default value for this option is zero.

Socket Reuse Addresses

This option specifies whether the rules used in validating addresses supplied in a `Bind` should allow reuse of local addresses. Operation of this option is protocol-specific. `Get_Socket_Reuse_Addresses` shall return the state of the Socket Reuse Addresses socket option for the socket. `Set_Socket_Reuse_Addresses` shall set the Socket Reuse Addresses socket option for the socket to the value specified by the `To` parameter. The default value for this option is `Disabled`, that is, reuse of local addresses is not permitted.

Socket Send Buffer Size

This option specifies the maximum buffer size, in octets, for data sent on this socket. This is the send queue limit noted in 18.2.5.3. `Get_Socket_Send_Buffer_Size` shall return the size of the Socket Send Buffer Size socket option for the socket in octets. `Set_Socket_Send_Buffer_Size` shall set the Socket Send Buffer Size socket option to the number of octets specified by the `To` parameter. The default value is implementation dependent, and may vary by protocol. The maximum value for this option is the value returned by `POSIX_Configurable_File_Limits.Socket_Buffer_Maximum`.

Socket Send Low Water Mark

This option specifies the minimum number of octets to process for socket output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations shall process as much data as permitted subject to flow control without blocking, but shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. A `Select_File` operation (19.1.2) testing the ability to write to a socket shall return `True` only if the number of octets set by the water mark could be processed. It

is implementation defined whether this option can be set. `Get_Socket_Send_Low_Water_Mark` shall return the Socket Send Low Water Mark socket option for the socket. `Set_Socket_Send_Low_Water_Mark` shall set the Socket Send Low Water Mark socket option for the socket to the value specified by the `To` parameter. The default value is implementation defined and protocol-specific.

Socket Send Timeout

This option specifies a timeout value for the amount of time that an output operation shall block because flow control prevents data from being sent. The option value is a `Duration` type specifying the limit on how long to wait for an output operation to complete. If a send operation has blocked for this much time, it returns with a partial count or sets the error code `Would_Block` if no data were sent. It is implementation defined whether this option can be set. `Get_Socket_Send_Timeout` shall return the Socket Send Timeout socket option value for the socket. `Set_Socket_Send_Timeout` shall set Socket Send Timeout socket option for the socket to the value specified by the `To` parameter. The default for this option is zero, indicating that a send operation shall not time out.

18.4.9.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`

The calling process does not have the appropriate privileges.

`Bad_File_Descriptor`

`Socket` is not a valid descriptor.

`Not_A_Socket`

The file descriptor `Socket` does not refer to a socket.

`Is_Already_Connected`

The socket is connected, and the implementation does not allow the option to be set in the Connected state. This error code may be caused by calls to any of the operations that set socket options.

`No_Buffer_Space`

A send or receive buffer of the requested size cannot be allocated. Only returned on calls to `Set_Socket_Receive_Buffer_Size` and `Set_Socket_Send_Buffer_Size`.

`Domain_Error`

A time specification is too large for the socket. Only returned on calls to `Set_Socket_Receive_Timeout` and `Set_Socket_Send_Timeout`.

18.4.10 Determine Whether a File Descriptor Refers to a Socket

18.4.10.1 Synopsis

```
function Is_A_Socket (File : POSIX_IO.File_Descriptor)
  return Boolean;
```

18.4.10.2 Description

The Is_A_Socket function shall return True if the file descriptor File refers to a socket and False otherwise.

18.4.10.3 Error Handling

If the following condition occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor
: The File parameter is invalid.

18.4.11 Listen for Connections on a Sockets

18.4.11.1 Synopsis

```
Connection_Queue_Length_Maximum : constant := impl-def-static-expression;
subtype Connection_Queue_Length is natural
   range 0 .. Connection_Queue_Length_Maximum;
procedure Listen
   (Socket  : in POSIX_IO.File_Descriptor;
    Backlog : in Connection_Queue_Length :=
                 Connection_Queue_Length'Last);
```

18.4.11.2 Description

Invoking Listen indicates a willingness by a server application to accept requests for incoming socket connections and to establish an upper limit to the number of entries on its input queue.

To accept connections, a socket is first created with Create and bound to a local socket address with Bind. A willingness to accept incoming connections is specified with Listen, and connection requests are dequeued with Accept_Connection.

Listen applies only to sockets of type Stream_Socket or Sequenced_Packet_Socket.

Socket is an open file descriptor that refers to a socket. Connection_Queue_Length specifies the maximum allowable number of pending requests for connections. It shall be implementation defined whether, when Connection_Queue_Length is zero, the queue limit is set to zero or an error is returned.

If a connection request arrives with the queue full, the protocol engine may report an error causing the client to receive an error, or the protocol engine may discard the request so that retries may succeed.

Implementations shall support values of Connection_Queue_Length up to Connection_Queue_Length_Maximum. Implementations may impose a limit on Connection_Queue_Length and silently reduce the specified value.

18.4.11.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Permission_Denied`
: The calling process does not have the appropriate privileges.

`Bad_File_Descriptor`
: `Socket` os not a valid descriptor.

`Not_A_Socket`
: The file descriptor `Socket` does not refer to a socket.

`Option_Not_Supported`
: The file descriptor refers to a socket that does not support Listen.

18.4.12 Receive Data From a Socket

18.4.12.1 Synopsis

```
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Masked_Signals   : in  POSIX.Signal_Masking;
    Options          : in  Message_Option_Set := Empty_Set);
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    Options          : in  Message_Option_Set := Empty_Set);
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    From             : in  Socket_Address_Pointer;
    Masked_Signals   : in  POSIX.Signal_Masking;
    Options          : in  Message_Option_Set := Empty_Set);
procedure Receive
   (Socket           : in  POSIX_IO.File_Descriptor;
    Buffer           : in  System.Address;
    Octets_Requested : in  POSIX.IO_Count;
    Octets_Received  : out POSIX.IO_Count;
    From             : in  Socket_Address_Pointer;
    Options          : in  Message_Option_Set := Empty_Set);
procedure Receive_Message
   (Socket          : in     POSIX_IO.File_Descriptor;
    Message         : in out Socket_Message;
    Octets_Received : out    POSIX.IO_Count;
    Masked_Signals  : in     POSIX.Signal_Masking;
    Options         : in     Message_Option_Set := Empty_Set);
procedure Receive_Message
   (Socket          : in     POSIX_IO.File_Descriptor;
    Message         : in out Socket_Message;
    Octets_Received : out    POSIX.IO_Count;
    Options         : in     Message_Option_Set := Empty_Set);
```

18.4.12.2 Description

The Receive_Message procedure and the Receive procedure with the From parameter present are used to receive data from the socket specified by Socket, whether it is a connection-mode socket.

Receive without the From parameter is normally used on a connected socket.

For the Receive procedure, the received data are placed in the object specified by Buffer. Receive_Message shall place the received data in the data segments described by the Socket_Message object specified by the Message parameter. The amount of data requested is specified by the Octets_Requested parameter for Receive and the Size attribute of the socket message segment for Receive_Message.

NOTE: Applications may use pointers to the POSIX.Octet_Array type to ensure proper data width for network I/O operations. When using pointers to other data types, byte width and ordering issues (i.e., big endian, little endian) are the responsibility of the application.

The Socket_Address_Pointer object designated by the From parameter shall be a conversion of an access to an object of a protocol-specific address type allocated by the application before the call to Receive (see 18.4.1.2). Alternately, the Null_Socket_Address constant may be passed in From causing this parameter to be ignored.

For a connectionless-mode socket, the From parameter in the call to Receive shall designate an address object of a specific type that corresponds to the protocol of the socket. Otherwise, Receive may return an error. Whether this error is detected is implementation dependent. For a connectionless-mode socket, the address of the peer sending the data or message is returned in the address object designated by the From parameter. For a connection-mode socket, the value referenced by From is undefined.

The number of octets actually received are returned in the Octets_Received parameter.

The Options parameter is a Message_Option_Set object used to specify allowable options on calls to Receive and Receive_Message. The operations "+", "-", ">", "<", ">=", "<=", and Empty_Set are available on the type Message_Option_Set via the derived type semantics of Ada, from the operations available for POSIX.Option_Set. The appropriate operations can be used to create and examine a set containing the receive options.

Signals are masked for the duration of the call, as specified in the Masked_Signals parameter. The original signal mask is restored when the procedure returns.

If the Peek_Only option is specified, Receive shall operate as if the option were not specified, returning data or errors, but the state of the socket, its received queue and any pending errors shall be unchanged by this call.

If out-of-band data are supported by the communications provider and if the socket contains out-of-band data, a call specifying the Process_OOB_Data option requests that out-of-band data be returned. It is an error if the Process_OOB_Data option is set and no out-of-band data exist in the receive queue.

The `Wait_For_All_Data` option requests that the subprogram block until the full amount of data requested can be returned. The subprogram may return a smaller amount of data if a signal is caught, the connection is terminated, or an error is pending for the socket. The effect of the `Wait_For_All_Data` option is similar to the effect of setting the Receive Low Water Mark socket option for the socket temporarily to the amount of data requested.

If the first element on a socket receive queue is an out-of-band data mark and neither the `Peek_Only` or `Process_OOB_Data` options is specified, the mark shall be discarded, and subsequent elements in the queue (if any) shall be processed.

A call to `Receive` in which the `Process_OOB_Data` option is not specified shall return immediately when one of the following conditions is true:

(1) A data segment containing out-of-band data is present at the beginning of the receive queue. See 18.2.5.5 for a definition of data segment and a description of socket buffer queuing [1)].

(2) A data segment containing normal data is present at the beginning of the receive queue, and either the segment is terminated or the amount of data in the segment is at least as much as the smaller of the amount of data requested and the Receive Low Water Mark socket option for the socket.

(3) The socket has a pending error.

(4) For a connection-mode socket, an end-of-file (indicated by `Octets_Received = 0` has been received, or the connection has been terminated.

(5) `Receive` is interrupted by a signal.

If none of these conditions is true and the socket is set to nonblocking, `Receive` returns with the error `Would_Block` indicating no data are available. If none of these conditions is true and the socket is set to blocking, `Receive` shall wait until one of the conditions becomes true.

If any of the conditions is true and a data segment is present in the receive queue, `Receive` shall return as much data as possible from the first segment in the queue. Otherwise, if more than one of the conditions are true, the condition listed first shall be processed.

If the function returns because data are available, the amount of data returned shall be the smaller of the data available in the segment or the amount requested. If the `Peek_Only` option is not specified, the data returned by `Receive` shall be removed from the socket queue. Otherwise the receive queue shall not be modified.

If the socket is of type `Datagram_Socket` or `Raw_Socket`, the amount of data requested is smaller than the amount of data contained in the segment, and the `Peek_Only` option is not specified, then the remainder of the data in the segment shall be discarded.

If `Receive` returns because of a pending error for the socket and the `Peek_Only` option is not specified, the pending error value shall be cleared.

---

1) This data segment is different from a TCP segment as described in RFC 793 {14}.

The Message Status attribute of a `Socket_Message` is a `Message_Status_Set` object that is set on return from `Receive_Message` to indicate conditions associated with the received data and other status information. The operations `"+"`, `"-"`, `">"`, `"<"`, `">="`, `"<="`, and `Empty_Set` are available on the type `Message_Status_Set` via the derived type semantics of Ada, from the operations available for `POSIX.Option_Set`. The appropriate operations can be used to examine a set containing the received socket message status for the following values:

`Received_OOB_Data`
: Out-of-band (expedited) data were received.

`End_Of_Message`
: The end of a record is indicated.

`Message_Truncated`
: Some trailing datagram data were discarded due to a lack of space or the amount of data requested was smaller than the data contained in the segment.

`Ancillary_Data_Lost`
: Some ancillary data were discarded due to lack of space.

The steps needed to create a `Socket_Message` for use in calls to `Receive_Message` are described in 18.4.1.3.

18.4.12.3 Error Handling

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Incorrect_Address_Type`
: The type of the address object designated by the `From` parameter is not of the appropriate type for the address format of this socket.

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Would_Block`
: The socket is marked as nonblocking and the receive operation would block; or the `Process_OOB_Data` option was selected, and the implementation does not support blocking to await out-of-band data.

`Bad_File_Descriptor`
: `Socket` os not a valid descriptor.

`Interrupted_Operation`
: The operation was interrupted by a signal before any data were available for the receive.

`Invalid_Argument`
: The `Process_OOB_Data` option was selected, and the protocol or socket state does not allow this operation.

No_Buffer_Space

Insufficient resources were available in the system to perform the operation.

Not_Connected

The socket is a Stream_Socket or a Sequenced_Packet_Socket and has not been connected.

Not_A_Socket

The file descriptor Socket does not refer to a socket.

Option_Not_Supported

The socket type specified does not support one or more of the options selected.

Message_Too_Long

The number of message segments in the Socket_Message object exceeds the system limit.

18.4.13 Send Data Over a Socket

18.4.13.1 Synopsis

```
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    Masked_Signals : in  POSIX.Signal_Masking;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    To             : in  Socket_Address_Pointer;
    Masked_Signals : in  POSIX.Signal_Masking;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send
   (Socket         : in  POSIX_IO.File_Descriptor;
    Buffer         : in  System.Address;
    Octets_To_Send : in  POSIX.IO_Count;
    Octets_Sent    : out POSIX.IO_Count;
    To             : in  Socket_Address_Pointer;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send_Message
   (Socket         : in  POSIX_IO.File_Descriptor;
    Message        : in  Socket_Message;
    Octets_Sent    : out POSIX.IO_Count;
    Masked_Signals : in  POSIX.Signal_Masking;
    Options        : in  Message_Option_Set := Empty_Set);
procedure Send_Message
   (Socket      : in  POSIX_IO.File_Descriptor;
    Message     : in  Socket_Message;
    Octets_Sent : out POSIX.IO_Count;
    Options     : in  Message_Option_Set := Empty_Set);
```

18.4.13.2 Description

The Send and Send_Message procedures are used to transmit data to a peer via the socket specified by Socket.

The peer socket address may have been specified in advance (by the use of the Connect procedure or the Specify_Peer procedure), in which case no destination address shall be specified. If the peer socket address has not been prespecified, a socket address shall be provided by the To parameter to the Send procedure or via the Message parameter to the Send_Message procedure.

The amount of data to send is specified by the Octets_To_Send parameter for Send and the Size attribute of the socket message segment for Send_Message. On return from Send, Octets_Sent indicates the number of octets accepted for transmission.

NOTE: Applications may use pointers to the POSIX.Octet_Array type to ensure proper data width for network I/O operations. When using pointers to other data types, byte width and ordering issues (i.e., big endian, little endian) are the responsibility of the application.

The Options parameter is a Message_Option_Set object used to specify allowable options on calls to Send and Send_Message. The operations "+", "-", ">", "<", ">=", "<=", and Empty_Set are available on the type Message_Option_Set via the derived type semantics of Ada, from the operations available for POSIX.Option_Set. The appropriate operations can be used to create and examine a set containing the send options.

Signals are masked for the duration of the call, as specified in the Masked_Signals parameter. The original signal mask is restored when the procedure returns. No indication of failure to deliver is implicit in the Send operation.

For connectionless-mode sockets (type Datagram_Socket or Raw_Socket) the specified data shall be transmitted as a single datagram. If the message is too long to pass atomically through the underlying protocol, then the error Message_Too_Long shall be returned, and the message is not transmitted. For connection-mode sockets (type Stream_Socket or Sequenced_Packet_Socket) transmission is not necessarily atomic, and part or all of the data may be transmitted.

If the socket has a pending error, Send shall raise the POSIX_Error exception, and the pending error shall be cleared. No data shall be transmitted in this case.

If the message cannot be immediately accepted by the communications provider because of a transient resource constraint (such as lack of buffer space), then Send normally blocks until all data can be transmitted, unless nonblocking has been set. If a call to one of these functions is interrupted by delivery of a signal after transmitting some data, the return status shall indicate the amount of data accepted for transmission.

The option Process_OOB_Data is used to send out-of-band data on sockets that support this notion (*e.g.*, Stream_Socket); *e.g.*, it can only be used if the underlying protocol also supports out-of-band data. The option End_Of_Message terminates a record for protocols that support that concept (end-of-record follows specified data).

The option Do_Not_Route specifies that the data being sent should bypass the standard routing facilities. This option may also be controlled using via Set_Socket_-Routing.

The steps needed to create a Socket_Message for use in calls to Send_Message are described in 18.4.1.3.

18.4.13.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Permission_Denied
: The calling process does not have the appropriate privileges.

Bad_File_Descriptor
: Socket os not a valid descriptor.

Interrupted_Operation
: The operation was interrupted by a signal before data or other indications were available.

Message_Too_Long
: The socket requires that the message be sent atomically, and the size of the message made this impossible.

Network_Down
: The local network connection is not operational.

Network_Unreachable
: The destination network is not reachable.

No_Buffer_Space
: The system was unable to allocate the internal buffer. The operation may succeed when buffers become available.

Not_Connected
: The socket is not connected or otherwise has not had the peer prespecified.

Not_A_Socket
: The file descriptor Socket does not refer to a socket.

Broken_Pipe
: The socket is of type Stream_Socket or a Sequenced_Packet_Socket and is no longer connected.

Would_Block
: The socket is marked as nonblocking, and the request would block.

Incorrect_Address_Type
: The type of the designated address object is incorrect for this socket (only on calls to Send_Message or Send with the To parameter).

Is_Already_Connected
: The socket is already connected and a destination address was specified (only on calls to Send_Message or Send with the To parameter).

Invalid_Argument

The To parameter is not valid for the specified address type (only on calls to Send_Message or Send with the To parameter).

Message_Too_Long

The socket requires that the message be sent atomically, and the size of the message made this impossible; or the number of segments in a multisegment message exceeds the system limit.

18.4.14 Shut Down Part of a Full-Duplex Connection

18.4.14.1 Synopsis

```
type Shutdown_Mode is
   (Further_Receives_Disallowed,
    Further_Sends_Disallowed,
    Further_Sends_And_Receives_Disallowed);
procedure Shutdown
   (Socket : in POSIX_IO.File_Descriptor;
    Mode   : in Shutdown_Mode);
```

18.4.14.2 Description

Shutdown shall cause all or part of a full-duplex connection on a socket to be shut down.

The Mode parameter further qualifies how the shutdown is to occur. If Mode is Further_Receives_Disallowed, then further receives shall be disallowed. If Mode is Further_Sends_Disallowed, then further sends shall be disallowed. If Mode is Further_Sends_And_Receives_Disallowed, then further sends and receives shall be disallowed.

The effects of Shutdown are protocol-specific.

18.4.14.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor

Socket os not a valid descriptor.

Invalid_Argument

Mode is not valid.

No_Buffer_Space

The system was unable to allocate the internal buffer. The operation may succeed when buffers become available.

Not_Connected

The socket is not connected.

Not_A_Socket

The file descriptor Socket does not refer to a socket.

18.4.15 Determine Whether a Socket is at the Out-of-Band Mark

18.4.15.1 Synopsis

```
function Socket_Is_At_OOB_Mark (Socket : POSIX_IO.File_Descriptor)
  return Boolean;
```

18.4.15.2 Description

If the protocol for the socket supports out-of-band data by marking the stream with an out-of-band data mark, `Socket_Is_At_OOB_Mark` shall return `True` if all the data preceding the out-of-band data have been read and the out-of-band data are next in the receive queue.

The `Socket_Is_At_OOB_Mark` function call shall not remove the mark from the stream. The use of this function between receive operations allows an application to determine which received data precede the out-of-band data and which follow the out-of-band data.

18.4.15.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`

: `Socket` os not a valid descriptor.

`Inappropriate_IO_Control_Operation`

: The file descriptor does not refer to a socket.

c

Section 19: Event Management

This section describes the event management facilities available under this standard.

19.1 Package POSIX_Event_Management

This package provides support for the poll event management and select event management functions.

```
with POSIX,
     POSIX_IO,
     POSIX_Limits,
     POSIX_Signals;
package POSIX_Event_Management is
   -- 19.1.1 Poll for File Descriptor Events
   type Poll_Events is new POSIX.Option_Set;
   Read_Not_High  : constant Poll_Events := implementation-defined;
   Read_Normal    : constant Poll_Events := implementation-defined;
   Read_Priority  : constant Poll_Events := implementation-defined;
   Read_High      : constant Poll_Events := implementation-defined;
   Write_Normal   : constant Poll_Events := implementation-defined;
   Write_Priority : constant Poll_Events := implementation-defined;
   Poll_Error     : constant Poll_Events := implementation-defined;
   File_Not_Open  : constant Poll_Events := implementation-defined;
   type Poll_FD is private;
   function Get_File (Poll_Item : Poll_FD)
      return POSIX_IO.File_Descriptor;
   procedure Set_File
      (Poll_Item : in out Poll_FD;
       File      : in     POSIX_IO.File_Descriptor);
   function Get_Events (Poll_Item : Poll_FD)
      return Poll_Events;
   procedure Set_Events
      (Poll_Item : in out Poll_FD;
       Events    : in     Poll_Events);
   function Get_Returned_Events (Poll_Item : Poll_FD)
      return Poll_Events;
   procedure Set_Returned_Events
      (Poll_Item : in out Poll_FD;
       Events    : in     Poll_Events);
   subtype Poll_FD_Array_Range is Positive
      range 1 .. POSIX.Open_Files_Maxima'Last;
   type Poll_FD_Array is array
      (Poll_FD_Array_Range range <>) of Poll_FD;
   Wait_Indefinitely : constant Duration := implementation-defined;
   procedure Poll
      (Files          : in out Poll_FD_Array;
       Response_Count : out    Natural;
       Timeout        : in     Duration := Wait_Indefinitely);
   -- 19.1.2 Select From File Descriptor Sets
   type File_Descriptor_Set is private;
   Empty_File_Descriptor_Set : constant File_Descriptor_Set;
   procedure Make_Empty (Set : in out File_Descriptor_Set);
   subtype Select_File_Descriptor is POSIX_IO.File_Descriptor
      range POSIX_IO.File_Descriptor'First .. POSIX_IO.File_Descriptor
            (POSIX_Limits.FD_Set_Maxima'Last - 1);
```

```
   procedure Add
      (Set  : in out File_Descriptor_Set;
       File : in     Select_File_Descriptor);
   procedure Remove
      (Set  : in out File_Descriptor_Set;
       File : in     Select_File_Descriptor);
   function In_Set
      (Set  : File_Descriptor_Set;
       File : Select_File_Descriptor)
      return Boolean;
   procedure Select_File
      (Read_Files     : in out File_Descriptor_Set;
       Write_Files    : in out File_Descriptor_Set;
       Except_Files   : in out File_Descriptor_Set;
       Files_Selected :    out Natural;
       Timeout        : in     Duration := Wait_Indefinitely);
   procedure Select_File
      (Read_Files     : in out File_Descriptor_Set;
       Write_Files    : in out File_Descriptor_Set;
       Except_Files   : in out File_Descriptor_Set;
       Files_Selected :    out Natural;
       Signals        : in     POSIX_Signals.Signal_Set;
       Timeout        : in     Duration := Wait_Indefinitely);
   generic
      with procedure Action
         (File : in     Select_File_Descriptor;
          Quit : in out Boolean);
   procedure For_Every_File_In (Set : in File_Descriptor_Set);

private
   implementation-defined
end POSIX_Event_Management;
```

19.1.1 Poll for File Descriptor Events

19.1.1.1 Synopsis

```
   type Poll_Events is new POSIX.Option_Set;
   Read_Not_High  : constant Poll_Events := implementation-defined;
   Read_Normal    : constant Poll_Events := implementation-defined;
   Read_Priority  : constant Poll_Events := implementation-defined;
   Read_High      : constant Poll_Events := implementation-defined;
   Write_Normal   : constant Poll_Events := implementation-defined;
   Write_Priority : constant Poll_Events := implementation-defined;
   Poll_Error     : constant Poll_Events := implementation-defined;
   File_Not_Open  : constant Poll_Events := implementation-defined;
   type Poll_FD is private;
   function Get_File (Poll_Item : Poll_FD)
      return POSIX_IO.File_Descriptor;
   procedure Set_File
      (Poll_Item : in out Poll_FD;
       File      : in     POSIX_IO.File_Descriptor);
   function Get_Events (Poll_Item : Poll_FD)
      return Poll_Events;
   procedure Set_Events
      (Poll_Item : in out Poll_FD;
       Events    : in     Poll_Events);
   function Get_Returned_Events (Poll_Item : Poll_FD)
```

```
   return Poll_Events;
procedure Set_Returned_Events
   (Poll_Item : in out Poll_FD;
    Events    : in     Poll_Events);
subtype Poll_FD_Array_Range is Positive
   range 1 .. POSIX.Open_Files_Maxima'Last;
type Poll_FD_Array is array
   (Poll_FD_Array_Range range <>) of Poll_FD;
Wait_Indefinitely : constant Duration := implementation-defined;
procedure Poll
   (Files          : in out Poll_FD_Array;
    Response_Count : out    Natural;
    Timeout        : in     Duration := Wait_Indefinitely);
```

19.1.1.2 Description

The functionality described in this subclause is optional. If the Poll option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

`Poll` provides applications with a mechanism for multiplexing input/output over a set of file descriptors that reference open files. It identifies and selects the files on which an application can send or receive messages or on which certain events have occurred. The `Files` parameter specifies the file descriptors to poll. The number of files selected is returned in the `Response_Count` parameter. The `Timeout` parameter provides an optional timeout mechanism.

The type `Poll_FD` shall be used to represent an object with at least the following attributes:

File
: The file descriptor to poll

Events
: A `Poll_Events` object representing a set of events to examine

Returned Events
: A `Poll_Events` object representing a set of events returned by the poll

The `Files` parameter specifies the file descriptors to be examined and the events of interest for each file descriptor. It specifies an array with one element for each open file descriptor of interest. The elements of the array are `Poll_FD` objects. The File attribute specifies an open file descriptor. Attributes Events and Returned Events are sets of event flags of type `POSIX.Option_Set` containing any combination of the event flags `Read_Not_High`, `Read_Normal`, `Read_High`, `Write_Normal`, `Write_Priority`, `Poll_Error`, and `File_Not_Open`.

The functions that the application can use for input and output depend on the type of file descriptor. For regular files, they shall include `Read` and `Write`. For character special files for use with XTI calls, they shall include `Listen`, `Receive`, `Confirm_Connection`, `Retrieve_Disconnect_Info`, `Acknowledge_Orderly_Release`, `Acknowledge_Orderly_Release_With_Data`, `Receive_Data_Unit`, `Retrieve_Data_Unit_Error`, `Receive_And_Scatter_Data`, `Receive_And_Scatter_Data_Unit`, `Send`, `Send_Disconnect_Request`, `Initiate_Orderly_Release`, `Initiate_Orderly_Release_With_Data`, `Send_Data_Unit`, `Gather_And_Send_Data`,

Gather_And_Send_Data_Unit. For use with sockets calls, they shall include Read, Receive, Receive_Message, Send, Send_Message, and Write.

In the this subclause, messages as well as other file type specific events (*e.g.*, end-of-file, disconnection indication) are referred to as data.

All data fall into one of three categories: normal data, priority data, and high priority data. The meanings of these categories depend on the file type and can also in part be implementation dependent. Each file type shall support at least the normal data category.

The type Poll_Events shall denote a set of poll event flags returned for a file descriptor. The operations "+", "-", ">", "<", ">=", "<=", and Empty_Set are available on the type Poll_Events via the derived type semantics of Ada, from the operations available for POSIX.Option_Set. The appropriate operations can be used to create and examine a set containing the required flags. The following Poll_Events flags shall be defined:

Read_Not_High
: Data other than high-priority data can be read without blocking.

Read_Normal
: Normal data can be read without blocking.

Read_Priority
: Priority data or high-priority data can be read without blocking.

Read_High
: High-priority data can be read without blocking.

Write_Normal
: Normal data can be written without blocking.

Write_Priority
: Priority data can be written without blocking.

Poll_Error
: An I/O error has occurred.

File_Not_Open
: The specified File attribute does not belong to an open file.

Read_Normal, Read_Priority, or Read_High shall be set in Returned_Events when a call to an input function with POSIX_IO.Non_Blocking clear would not block and normal data, priority data, or high-priority data (respectively) have been received, whether or not the function would transfer data successfully.

NOTE: The data might not be returned by the input function under some circumstances. For example, an error might be returned if an invalid parameter has been supplied.

Read_Not_High shall be set in Returned_Events when a call to an input function with POSIX_IO.Non_Blocking clear would not block, and normal data or priority data have been received, whether or not the function would transfer data successfully.

`Write_Normal` shall be set in `Returned_Events` when a call to an output function with normal data and with `POSIX_IO.Non_Blocking` clear would not block.

NOTE: The data might not be output under some circumstances. For example, an error might be returned if an invalid parameter has been supplied.

`Write_Priority` shall be set in `Returned_Events` when priority data have already been output for the file descriptor and a call to an output function with priority data and with `POSIX_IO.Non_Blocking` clear would not block.

Flags `Poll_Error` and `File_Not_Open` are only valid in the `Returned_Events` parameter; they shall not be set in the `Events` parameter. `Poll_Error` shall be set in `Returned_Events` when an error has occurred on the open file specified by the File attribute. `File_Not_Open` shall be set in `Returned_Events` when the File attribute does not belong to an open file.

For each element of the array specified by `Files`, `Poll` examines the given file descriptor for the event(s) specified in `Events`.

If an event is meaningless for a particular file type, the behavior of `Poll` for that file type and event is unspecified.

The results of the `Poll` query are stored in the `Returned_Events` attribute in the `Poll_FD` object. `Returned_Events` can be examined as described above using the operations for `POSIX.Option_Set` to indicate which of the requested events is true. If none are true, `Returned_Events` is set to `Empty_Set` when `Poll` returns. The event flags `Poll_Error` and `File_Not_Open` are always set in `Returned_Events` if the conditions they indicate are true, even though these flags were not set in `Events`.

If none of the defined events has occurred on any specified file descriptor, `Poll` shall wait until an event has occurred on any of the specified file descriptors or until at least the time specified by `Timeout` has elapsed and should then return as soon as possible.

NOTE: Timing accuracy varies on different systems.

If the `Timeout` parameter is used and the value of `Timeout` is zero, `Poll` shall return immediately. If the `Timeout` parameter is used and the value of `Timeout` is greater than zero, `Poll` shall block until a requested event occurs, the time in `Timeout` elapses, or the call is interrupted. If the default `Timeout` value (`Wait_Indefinitely`) is used, `Poll` shall block until a requested event occurs or until the call is interrupted. `Poll` shall not be affected by the `POSIX_IO.Non_Blocking` flag.

File descriptors for use with XTI and sockets shall support normal data and priority data.

For XTI the following apply:

— Whether expedited data are normal data or priority data is implementation defined for each communications provider.

— Whether `Disconnect_Request_Received` and `Connect_Request_Received` are normal or priority data is implementation defined for each communication provider.

— If expedited data are priority data or if Disconnect_Request_Received and Connect_Request_Received are priority data, then Connect_Response_-Received shall be priority data. Otherwise, whether Connect_Response_-Received are normal or priority data is implementation defined.

— Normal_Data_Received, Orderly_Release_Request_Received, Error_-In_Previously_Sent_Datagram, and other data shall be normal data.

— If Events is set to Read_Normal, then Poll shall return with Read_Normal set in Returned_Events when normal data have been received.

— If Events is set to Read_Priority, then Poll shall return with Read_Priority set in Returned_Events when priority data have been received.

— If Read_Not_High is set in Events, then Poll shall return with Read_Not_High set in Returned_Events when any of Connect_Response_Received, Normal_Data_Received, Disconnect_Request_Received, Expedited_Data_-Received, Connect_Request_Received, Orderly_Release_Request_Received, or Error_In_Previously_Sent_Datagram have been received.

— If Write_Normal is set in Events, then Poll shall return with Write_Normal set in Returned_Events unless an attempt to send normal data would block (or return Flow_Control_Error).

— If Write_Priority is set in Events, then Poll shall return with Write_Priority set in Returned_Events if expedited data are priority data, expedited data have been sent, and an attempt to send expedited data would not block (or return Flow_Control_Error).

— Poll_Error may be set in Returned_Events for implementation-defined reasons and the error reason shall then be returned by any XTI call that would otherwise be valid.

— The only XTI call that can succeed once Poll_Error has been returned is Close.

— When Poll returns with Read_Not_High, Read_Normal or Read_High set in Returned_Events, the application can determine which event has occurred by calling Look.

For sockets the following apply:

— Whether connection indications (consumed by Accept_Connection) are normal or priority data is implementation defined for each communications provider.

— Out-of-band data shall be priority data.

— Other data shall be normal data.

— If Read_Not_High is set in Events and the socket is in the Listening state, then Poll shall return with Read_Not_High set in Returned_Events when an incoming connection indication has been received so that a call to Accept_Connection would complete without blocking. If connection indications are normal data, then Read_Normal can be used instead of Read_Not_High. If they are priority data, then Read_Priority can be used.

— If Write_Normal is set in Events and the socket is in the Connecting state, then Poll shall return with Write_Normal set in Returned_Events when connection establishment is complete.

— `Poll_Error` may be set in `Returned_Events` for implementation-defined reasons and the error reason shall then be returned by any sockets call that would otherwise be valid.

— The only socket call that can succeed once `Poll_Error` has been returned is `Shutdown`.

— If `Events` is set to `Read_Priority`, then `Poll` shall return with `Read_Priority` set in `Returned_Events` when any of the following becomes true:

(1) Out-of-band data have been received and `OOB_Data_Inline` is not set.
NOTE: If `OOB_Data_Inline` is set, then the out-of-band data would be returned by an input function with `POSIX_IO.Non_Blocking` set, and this condition is indicated by `Read_Normal` and `Read_Not_High`.

(2) An out-of-band data mark is present in the receive queue.

(3) `Connect` has been called with `POSIX_IO.Non_Blocking` set, and the connection attempt has failed.

(4) A connection has been broken by the peer or due to expiration of the Socket Keep Alive timeout.

On success, `Poll` returns the total number of file descriptors that have been selected in `Response_Count` (that is, file descriptors for which the `Returned_Events` field is not `Empty_Set`). A value of zero indicates that the call timed out and no file descriptors have been selected.

19.1.1.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Resource_Temporarily_Unavailable`
: Allocation of internal data structures failed, but the request may be attempted again.

`Bad_Address`
: A parameter points outside the allocated address space.

`Interrupted_Operation`
: A signal was caught during the `Poll` system call.

19.1.2 Select From File Descriptor Sets

19.1.2.1 Synopsis

```
type File_Descriptor_Set is private;
Empty_File_Descriptor_Set : constant File_Descriptor_Set;
procedure Make_Empty (Set : in out File_Descriptor_Set);
subtype Select_File_Descriptor is POSIX_IO.File_Descriptor
   range POSIX_IO.File_Descriptor'First .. POSIX_IO.File_Descriptor
         (POSIX_Limits.FD_Set_Maxima'Last - 1);
procedure Add
   (Set  : in out File_Descriptor_Set;
    File : in     Select_File_Descriptor);
procedure Remove
   (Set  : in out File_Descriptor_Set;
    File : in     Select_File_Descriptor);
```

```
function In_Set
   (Set  : File_Descriptor_Set;
    File : Select_File_Descriptor)
   return Boolean;
procedure Select_File
   (Read_Files     : in out File_Descriptor_Set;
    Write_Files    : in out File_Descriptor_Set;
    Except_Files   : in out File_Descriptor_Set;
    Files_Selected :    out Natural;
    Timeout        : in     Duration := Wait_Indefinitely);
procedure Select_File
   (Read_Files     : in out File_Descriptor_Set;
    Write_Files    : in out File_Descriptor_Set;
    Except_Files   : in out File_Descriptor_Set;
    Files_Selected :    out Natural;
    Signals        : in     POSIX_Signals.Signal_Set;
    Timeout        : in     Duration := Wait_Indefinitely);
generic
   with procedure Action
      (File : in     Select_File_Descriptor;
       Quit : in out Boolean);
procedure For_Every_File_In (Set : in File_Descriptor_Set);
```

19.1.2.2 Description

The functionality described in this subclause is optional. If the Select option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

The `Select_File` procedure examines a set of file descriptors passed in the `Read_Files`, `Write_Files`, and `Except_Files` parameters to see whether some of the descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively.

The set of file descriptors is passed in an object of type `File_Descriptor_Set`. The constant `POSIX_Limits.FD_Set_Maxima'Last` shall specify the maximum number of file descriptors in `File_Descriptor_Set`. This value is implementation defined and shall be greater than or equal to `POSIX_Limits.Portable_FD_Set_Maximum`. All objects of type `File_Descriptor_Set` shall have the initial value `Empty_File_Descriptor_Set`. The procedure `Make_Empty` shall set the descriptor set indicated by `Set` to the value `Empty_File_Descriptor_Set`, freeing any dynamically allocated storage associated with the object.

`Add` shall add the file descriptor `File` to the set indicated by `Set`. If the file descriptor `File` is already in this set, there shall be no effect on the set, nor will an error be returned. `Remove` shall remove the file descriptor `File` from the set indicated by `Set`. If `File` is not a member of this set, there shall be no effect on the set, nor will an error be returned. The function `In_Set` shall evaluate to `True` if the file descriptor `File` is a member of the set indicated by `Set` and shall evaluate to `False` otherwise.

The application program instantiates the generic procedure `For_Every_File_In` with an actual procedure for the generic formal procedure `Action`. When called, the instance shall call the actual procedure supplied for `Action` once for each file

descriptor in the associated File_Descriptor_Set. The order of traversal is unspecified.

Action shall be able to force termination of the generic instance either by setting Quit to True or by raising an exception. Prior to calling Action, the instance shall set Quit to False. Exceptions raised by Action shall terminate iteration and shall be propagated back to the caller of the instance. After an exception is raised by Action or Action returns with Quit set to True, no more calls to Action shall occur.

Select_File shall examine all file descriptors in each set. On return, Select_File shall replace each of the three input descriptor sets with a subset of the corresponding input set that includes only the descriptors that are ready for the requested operation, and shall return the total number of ready descriptors in all the output sets in the Files_Selected parameter. If a file descriptor is in more than one output set, it is counted as many times as it occurs.

The Signals parameter is an object of type POSIX_Signals.Signal_Set. If the Signals parameter is included, then Select_File shall replace the signal mask of the process by the set of signals indicated by Signals before examining the descriptors, and shall restore the signal mask of the process before returning.

A descriptor shall be considered ready for reading when a call to an input function with POSIX_IO.Non_Blocking clear would not block, whether or not the function would transfer data successfully. (The function might return data, an end-of-file indication, or an error other than one indicating that it is blocked; and in each of these cases the descriptor shall be considered ready for reading.)

A descriptor shall be considered ready for writing when a call to an output function with POSIX_IO.Non_Blocking clear would not block, whether or not the function would transfer data successfully.

If the operation is meaningless for a particular file type, Select_File shall indicate that the descriptor is ready for read or write operations and shall indicate that the descriptor has no exceptional condition pending.

For XTI, whether expedited data are normal data or priority data is implementation defined for each communications provider.

A file descriptor for use with XTI calls shall be considered ready for reading when any of the events Normal_Data_Received, Orderly_Release_Request_Received, or Error_In_Previously_Sent_Datagram has occurred. Such a file descriptor shall also be considered ready for reading when any of the events Connect_Response_Received, Disconnect_Request_Received, Expedited_Data_Received, or Connect_Request_Received has occurred and the event is not indicated as an exceptional condition. Whether these events are indicated as exceptional conditions is implementation defined.

NOTE: These conditions are the same as those under which Poll returns with Read_Normal set.

A file descriptor for use with XTI calls shall be considered ready for writing if the output functions (Send, Initiate_Orderly_Release, Initiate_Orderly_Re-

lease_With_Data, Send_Data_Unit, Gather_And_Send_Data, and Gather_-And_Send_Data_Unit) would not block or return Flow_Control_Error.

NOTE: These conditions are the same as those under which Poll returns with Write_Normal set.

A file descriptor for use with XTI calls shall be considered to have an exception condition when any of the following occur:

— An implementation-defined error has occurred.

— An Expedited_Data_Received event has occurred, and this event does not set the file descriptor ready for reading.

— Either of the events Disconnect_Request_Received or Connect_Request_-Received has occurred, and these events do not set the file descriptor ready for reading.

— A Connect_Response_Received event has occurred, and this event does not set the file descriptor ready for reading.

A Connect_Response_Received event shall be indicated as an exceptional condition if an Expedited_Data_Received event is indicated as an exceptional condition or if a Disconnect_Request_Received or Connect_Request_Received event is indicated as an exceptional condition. Otherwise, it is implementation defined which of the Expedited_Data_Received, Disconnect_Request_Received, Connect_Request_Received, and Connect_Response_Received events set the file descriptor ready for reading.

NOTE: These are the same circumstances under which Poll returns with Read_Priority or Poll_Error set. The Select_File procedure does not detect the circumstances under which Poll returns with Read_High or Write_Priority set.

If a descriptor refers to a socket, the implied input function is the Receive_Message procedure with parameters requesting normal and ancillary data, such that the presence of either type shall cause the socket to be marked as readable. The presence of out-of-band data will be checked if the socket option OOB_Data_Inline has been enabled, as out-of- band data are enqueued with normal data. If the socket is currently in the Listening state, it will be marked as readable if an incoming connection request has been received, and a call to the Accept_Connection procedure is guaranteed to complete without blocking.

If a descriptor refers to a socket, the implied output function is the Send_Message procedure supplying an amount of normal data equal to the current value of the Send_Low_Water_Mark option for the socket. If a nonblocking call to the Connect procedure has been made for a socket and the connection attempt has either succeeded or failed (leaving a pending error), the socket shall be marked as writable.

A socket shall be considered to have an exceptional condition pending if a receive operation with POSIX_IO.Non_Blocking clear for the open file description and with the Process_OOB_Data option set would return out-of-band data without blocking. (It is protocol-specific whether Process_OOB_Data would be used to read out-of-band data.) A socket shall also be considered to have an exceptional condition pending if an out-of-band data mark is present in the receive queue. Other circumstances

under which a socket may be considered to have an exceptional condition pending are protocol specific and implementation defined.

If none of the selected descriptors are ready for the requested operation, `Select_File` may block until at least one of the requested operations becomes ready. The parameter `Timeout` controls how long `Select_File` may take to complete. The `Timeout` parameter specifies a maximum interval to wait for the selection to complete. If the specified time interval expires without any requested operation becoming ready, the procedure shall return. If the default `Timeout` value (`Wait_Indefinitely`) is used, then the call to `Select_File` shall block indefinitely until at least one descriptor meets the specified criteria. A `Timeout` parameter value of zero can be used to effect a poll. A blocked `Select_File` operation may be interrupted by a signal.

Any of the parameters `Read_Files`, `Write_Files`, and `Except_Files` may be empty sets if no descriptors are of interest.

19.1.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: One of the descriptor sets specified an invalid descriptor.

`Bad_Address`
: One or more parameters specified an invalid address.

`Interrupted_Operation`
: A signal was delivered before the time limit expired and before any of the selected events occurred.

`Invalid_Argument`
: The parameter `Timeout` is invalid (negative or too large).

c

Annex A
(informative)
Bibliography

The following documents are related to this standard in an informative manner or were used as references in its preparation.

{B1} Adams, Douglas, *The Hitchhikers Guide to the Galaxy*. New York: Crown Books, 1989.

{B2} Blair, Byron E., Editor *Time and Frequency: Theory and Fundamentals*. U.S. Dept. of Commerce, National Bureau of Standards, May 1974.

{B3} ISO/IEC 7498-1:1994, *Information technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*.

{B4} ISO/IEC 8348/AD2, *Addendum to the Network Service Definition Covering Network Layer Addressing*.

{B5} ISO/IEC 8652:1987, *Programming Languages— Ada* (endorsement of ANSI-/MIL-STD 1815A-1983).

{B6} ISO/IEC 8859-1:1998, *Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin alphabet No. 1*.

{B7} ISO/IEC JTC 1/SC22/WG9/Ada Rapporteur Group, *GET_LINE for interactive devices*, Ada Issue AI-00172, August 1986.[1)]

{B8} ISO/IEC JTC 1/SC22/WG9/Ada Rapporteur Group, *Data can be appended to the end of an existing file*, Ada Issue AI-00278, June 1989.

{B9} ISO/IEC JTC 1/SC22/WG9/Ada Rapporteur Group, *NEW_PAGE can raise USE_ERROR for certain FORMs of File*, Ada Issue AI-00886, January 1991.

{B10} Jacobson, V., “Congestion Avoidance and Control”, ACM SIGCOMM-88, August 1988.

{B11} McJones, Paul R., and Swart, Garret F., “Evolving the UNIX System Interface to Support Multithreaded Programs,” Digital Equipment Corp., Palo Alto, CA, DEC-SRC Report 21, September 1987.

{B12} IEEE Standard 1003.1b-1993, *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]—Amendment 1: Realtime Extension*.[2)]

{B13} IEEE Std 1003.1c-1995, *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API)—Amendment 2: Threads Extension [C Language]*.

{B14} IEEE P1003.1g/D6.6, *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API)—Amendment xx: Protocol Independent Interfaces (PII)*.

1) Ada Issues can be obtained from the Ada Information Clearinghouse, c/o IIT Research Institute, 4500 Forbes Boulevard, Lanham, MD 20706-4312, USA.

2) IEEE publications can be obtained from the Institute of Electrical and Electronic Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, JN 08855-1441, USA (http://www.standards.ieee.org/).

{B15} IEEE Std 1003.1i-1995, *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API)—Amendment: Technical Corrigenda to Realtime Extension [C Language].*

{B16} IEEE Std 1003.1i-1995, *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities.*

{B17} IEEE Std 1003.5-1992 (R 1990), *IEEE Standard for Information Technology—POSIX® Ada Language Interfaces—Part 1: Binding for System Application Programming Interface (API).*

{B18} IEEE Std 1003.5b-1996 (R 1990), *IEEE Standard for Information Technology—POSIX® Ada Language Interfaces—Binding for System Application Program Interface (API)—Amendment 1: Realtime Extensions.*

{B19} IEEE Std 1003.13-1998, *IEEE Standard for Information Technology—Standardized Application Environment Profile—POSIX Realtime Application Support (AEP).*

{B20} Scheifler, Robert, *X Window System Protocol, Version 11, Release 5* (from the X11R5 distribution).

{B21} IETF RFC 952:1985, *DoD Internet Host Table Specification.*

{B22} IETF RFC 1020:1987, *Internet Numbers.*

{B23} IETF RFC 1983:1996, *Internet Users' Glossary.*

c

Annex B
(informative)
Rationale and Notes

B.1 General

This rationale is for the POSIX/Ada binding as a whole. There is also a rationale clause corresponding to each section of the standard. This rationale does not attempt to justify or explain the choice of interfaces or functionality in the C-language systme API standards on which this standard is based. See those standards for that rationale.

In what follows, it is sometimes useful to distinguish between the original version of this standard, denoted by POSIX.5, the realtime amendment of this standard, denoted by POSIX.5b, and the sockets/XTI amendment, which is denoted by POSIX.5c. |c

B.1.1 Purpose and Audience

The purpose of the POSIX/Ada binding is to promote application portability at the Ada source-code level. The goal is that Ada applications using this binding be portable among POSIX-conforming Ada implementations.

Application programs that are intended to be portable to all POSIX implementations may be checked for portability by examining their **`with`** clauses.

This binding has been designed so that it may easily be used by Ada programmers, who do not necessarily have prior experience with either POSIX or C.

Originally, this rationale served an archival and advocacy purpose, recording the resolution of conflicts among the developers and providing supporting arguments for the benefit of balloters. As part of the POSIX.5b revision, the rationale was totally rewritten to serve a new audience:

— Application programmers interested in making effective use of this standard.

— Implementors interested in subtleties and intentions of features.

— Developers of future revisions to this standard interested in preserving the overall approach and philosophy of this Ada language binding.

B.1.2 Relation to Other POSIX Standards

This binding is dependent on the corresponding POSIX C-language system API standards in the sense that the semantics defined in that document must be followed by this standard. In some cases, it might have been desirable to change the semantics of some feature of POSIX to accommodate Ada better, but that was considered beyond the scope of a language binding. |c

This standard does *not* require that the system support any C-language interfaces. (See B.1.4.) In particular, the system is not required to support C-language thread services. Ada already has multiple threads of control within a process. This standard just defines the interactions of tasks with the facilities of POSIX. Support for tasks is required of all implementations of the Ada language and, therefore, is required for

all implementations of this standard. The developers of this standard have expended great effort to make it transparent to the Ada application whether the POSIX/C thread interfaces are supported by the underlying system.

This standard includes a binding to the POSIX thread services in the sense that several changes have been made from POSIX.5 to achieve semantic compatibility with an implementation in which both Ada tasks and POSIX threads map to the same type of entity in the underlying operating system. Moreover, some interfaces are provided for services defined by POSIX.1 for threads, which are not defined for Ada tasks by the Ada RM {1} but should be implementable by an Ada runtime system, regardless of whether the system supports POSIX threads.

B.1.3 Ada Language

The original version of this standard, POSIX.5, defined a binding to Ada 83, since that was the recognized standard for the Ada language at the time the binding was developed. Concurrent with the revision POSIX.5b, the Ada language standard was also being revised. It was decided to retain compatibility in the binding with Ada 83, but also to support migration to Ada 95. This decision was taken because production implementations of Ada 95 were not expected to be available immediately, and there was an immediate demand for the revised POSIX/Ada binding. Moreover, implementations of Ada 83 were expected to continue to be used for maintenance of existing applications.

In some cases, new functionality of Ada 95 conflicts with or overlaps functionality defined by POSIX.1 for threads. Such cases have been resolved individually, taking into account the degree of implementation freedom given by the Ada language. The most extreme case is the model for thread cancelation, which is too restrictive to support the new Ada semantics for abortion, exception propagation, and object finalization. In this case, it was decided to regard the POSIX thread feature as specific to the C-language binding and, therefore, to provide no Ada binding for it. In some other cases, such as thread scheduling, the Ada language provides enough flexibility that the POSIX.1 scheduling model can be provided as a user-selectable alternative to the standard Ada facility, with the implementation being left to define any interaction between the two.

In some cases, new Ada 95 features are needed to make effective use of the POSIX interface. A good example is the extended support for the type `Address` in package `System` and its new child package `System.Storage_Elements`. In this case, to provide the capability during transition from Ada 83 to Ada 95, this standard defines the package `System_Storage_Elements` to be a renaming of `System.Storage_Elements` and gives the implementation explicit permission to replace the renaming with a package specification and body with similar functionality, as close as possible to the Ada 95 package, without depending on any new language features.

The first few drafts of POSIX.5c used the full Ada 95 language, including child packages, general access types, access parameters, and tagged types. This decision was made based on the expected publication date for this standard was long enough after the adoption of Ada 95 to allow for mature Ada 95 compiler technologies.

Some ballot objections were raised concerning compatibility with some large applications that are in maintenance mode, and have frozen the compilation system at C

Ada 83. Apparently, the maintainers of those systems were interested in converting from C interface code to the POSIX/Ada socket bindings.

There were also objections to the use of tagged types from balloters on grounds other than compatibility with Ada 83. There were concerns regarding the complexity of dynamic dispatching operations. There were also stylistic disagreements about how tagged types should be used. The latter disagreements had a religious flavor, and since tagged types were new enough in the Ada language that a strong community consensus had not developed, there seemed to be no way to resolve the controversy.

Due to these objections, in Draft 4 of POSIX.5c the approach to was modified as follows:

— Ada 95 remains the normative reference for this standard for the rationale described above.

— Ada 83 fallback approaches are defined for all uses of Ada 95 features. These approaches are documented in 1.3.1.1.

— The standard uses Ada 95 child packages to organize the hierarchy of protocol independent and protocol specific features. For compatibility with earlier POSIX/Ada binding standards (and to avoid changing thousands of references), the Ada 83 fallback approach for child packages is used for all package names. In other words, packages designed to use names with the form `POSIX.Foo.Bar` are renamed to use the Ada 83 form `POSIX_Foo_Bar`. The conformance instructions in 1.3 reflect this decision.

— All the access parameters in earlier drafts of POSIX.5c were replaced by **`in`** parameters of an Ada 95 general access type. The Ada 83 fall back approach is to use **`in`** parameters of an ordinary access type, dynamic allocation, or unchecked conversion.

— The standard still uses Ada 95 general access types. The Ada 83 fall back approach is to use ordinary access types, dynamic allocation, or unchecked conversion.

— After much thought and consideration, the uses of tagged types (for base communications addresses) have been replaced by private pointer types. Each child package defines a private protocol specific address type along with an Ada 95 general access type pointing to the protocol-specific address type. The Ada 83 fallback for these types are ordinary access types or the use of dynamic memory allocation or unchecked conversion. The protocol-specific child packages also provide functions to convert the protocol-specific pointers to and from the base package private pointer type (to be used by the base package services). One extra function for each child package is provided to perform an "`Is_A_`" operation to query for a protocol specific address type given the base address type. C

B.1.4 Implementation Model

The POSIX/Ada binding is intended to be implementable on a variety of systems, in a variety of ways. In particular, the binding has been developed to allow for the following implementation possibilities:

— An implementation on a bare machine, with no underlying operating system support (*e.g.*, corresponding to the Minimal Realtime Profile of POSIX.13).

— An implementation based on an underlying operating system conforming to POSIX.1 but not supporting the optional _POSIX_THEADS functionality.

— An implementation based on an underlying operating system conforming to POSIX.1, including all the optional functionalities.

The only means by which the application might distinguish between these possibilities is the support or nonsupport for particular options. For example, blocking behavior might not be per task in an implementation based on an underlying operating system that does not support the _POSIX_THREADS optional functionality of POSIX.1. Thus, Strictly Conforming POSIX.5 applications should be portable across all implementations.

B.1.5 Level of Binding

The developers of POSIX.5 faced a fundamental choice between a direct *versus* an abstract binding of the POSIX.1 functions into Ada. A direct binding maps the POSIX/C functions as closely as possible to the corresponding Ada facilities. Arguments in favor of a direct binding are that it is less expensive in time and effort to produce and the implementation is more likely to be able to call the underlying POSIX.1 functions directly.

An abstract binding tries to group POSIX functions into abstract data types, operations on those types, and packages of logically related operations. Arguments in favor of an abstract binding are that it allows a more natural style and bindings may be omitted more easily for POSIX functions that duplicate already existing features of Ada.

The abstract binding approach was chosen for POSIX.5. Examples of ways in which the Ada binding differs from a direct translation of C into Ada include the following:

— Using packages to organize declarations that are logically related.

— Using generics to generalize certain C functions.

— Using more expressive Ada names.

— Splitting multipurpose C functions into several Ada subprograms.

— Mapping C *errno* codes to the Ada exception mechanism.

— Omitting POSIX/C functions that duplicate existing Ada language features.

The model is that there is a basic abstraction, consisting of the set of POSIX types, values, and operations. The POSIX.1 binding is one expression of the POSIX abstraction in the C language. This standard represents another expression of the same POSIX abstraction, in the Ada language. The approach used to develop the Ada binding was to uncover the basic abstraction and then to express it as well as possible in Ada.

The POSIX.5b revision did not seriously reconsider this question. To do so would have required a total rewrite of POSIX.5. Besides, input from the balloting group seemed to favor continuation of the abstract binding approach.

B.1.6 Form of Document

A related, but separate, issue is the form of this document. Historically, POSIX standards are first expressed in the C language. Bindings of these standards to other languages then face a choice between a thin document and a thick document. A thin binding document is one that defines only the syntax of the operations in the new language. Readers of the language binding are referred to the POSIX/C standards for an explanation of the required behavior. Arguments in favor of a thin binding document are that it less expensive to produce and maintain and it is less likely to introduce inadvertent discrepancies between the documents defining the C-language binding and the other language bindings.

A thick binding document, on the other hand, is self-contained. The required behavior of each interface in the new language is fully described, so that readers need not refer to the POSIX/C stanards. Arguments in favor of a thick document are that users can find all the needed information in one place and it is likely to be more understandable.

The original version of this standard is a thick binding document. The required behavior of the POSIX/Ada interface is fully documented; users need not refer to POSIX.1 and, therefore, need not be conversant with C.

At the time work started on POSIX.5b, the IEEE PASC officially adopted the language-independent standards paradigm promoted by ISO. It was believed that the development and maintenance of multilingual interfaces for POSIX would be easier if there were a common LIS of POSIX services. Language bindings then would be thin specifications of syntax and would refer back to the LIS for the required behavior of the interfaces. Considerable effort was expended within POSIX to make this LIS paradigm work, but several difficulties were encountered.

First, the POSIX/C standards were all intended to comprise one standard, but were being developed piecemeal. It was difficult to synchronize the development of LIS documents with the development of extensions to C language standards. Several extensions of POSIX.1 were being developed as C-language bindings, most of which were already in ballot. Conversion to thin bindings and LIS would mean essentially going back and starting over.

Second, even for new work, it was difficult to find volunteers willing and able to create LIS documents. Most of the POSIX work is based on established practice in the C/UNIX market, is done by individuals who are experienced in C, and is funded by organizations who regard C as the predominate language in their markets. Finding people willing to develop LIS documents is much harder. The notation is unfamiliar, and the connection to practice is tenuous. Thus, progress on LIS standards has been very slow.

A final problem is end-user hostility to the LIS model. The combination of LIS and thin language binding is much harder to read and understand than a single self-contained document. To understand the behavior required by such a standard, the user must master both the LIS notation and the specific programming language, and mentally keep track of the correspondence of names and concepts between the two documents — using one set of names for syntax and another set of names for semantics.

The first balloted draft of POSIX.5b was the result of a good-faith effort to develop a thin Ada binding to an LIS. Since there was no LIS for POSIX.1b or POSIX.1c, the C-language interfaces were used in place of the LIS. The response to the first ballot of POSIX.5b was overwhelmingly against changing over to the thin binding format. The consensus of POSIX.5b balloting group coincided with a broader consensus within IEEE PASC that the LIS paradigm was an impediment to the development of needed standards, and so the LIS requirement was deleted. The next draft of POSIX.5b was converted to the thick format consistent with POSIX.5, and this format has been maintained through POSIX.5c.

B.1.7 Global Issues

B.1.7.1 Document Structure

The document structure of this standard closely follows that of the corresponding POSIXC system API standards. Each section consists of one or a few Ada package specifications, containing declarations for logically related types, constants, and operations. A few operations have been moved from the section where they appear in the C interface to a different section in the Ada binding to achieve Ada packaging consistency; these differences date back to POSIX.5. To aid users and implementors in relating Ada binding entities to the C interface entities, Annex C provides a C-to-Ada cross-reference.

Originally, the XI and Sockets interfaces were separated into two distinct documents. These two documents were combined into one standard for the following reasons:

— Although the XTI and Sockets interfaces are both required for P1003.1g compliance, the Ada binding continues to specify these interfaces as options. Therefore, the separation of the two interfaces into different documents does not provide any difference (or advantage) in the way these options are managed.

— Other than the sections containing the actual bindings for the XTI and Sockets DNI (Section 17 and Section 18), many of the normative sections and annexes of P1003.1g apply to both XTI and Sockets. If these sections were split into two documents, extra time and effort would be required to coordinate changes that apply to both. In addition, tracking any further refinements to P1003.1g is simpler if the POSIX/Ada bindings are all in a single document.

— One standards document would reduce the risk of ballot pool fatigue. It also eliminates coordination problems, where an objection in a common section of one document may or may not apply to both, or where the resolution of an objection in one document causes a new objection to be raised in the other. c

B.1.7.2 Packaging

The developers of POSIX.5 wanted to produce an object-oriented packaging for POSIX, but could not because of the overriding decision to remain reasonably consistent with the organization of the POSIX/C system API standards.

Four packaging styles received consideration:

— Single package: specify the entire binding within a single package with no subpackaging.

— Multiple packages: create several separate packages, each centered around a small set of logically related types and operations.

— Subpackaged: the same as multiple packages, except that the entire binding is wrapped by a single outer package.

— Limited subpackaged: place nearly all of the binding in a single package, but place a few selected portions (*e.g.*, unsafe operations) in subpackages.

After considering the arguments for and against each alternative, the multiple packages approach was selected, primarily because it could be converted easily into one of the other styles, should that become desirable.

Arguments in favor of multiple packages include

— Clear identification of interface usage, by means of examining the **`with`** clauses of an application; and

— Use of Ada to convey the structure of the binding, which permits understanding of the binding in smaller, relatively independent parts.

Disadvantages of multiple packages include

— Complicated visibility rules and

— Large and segmented name space, so that it is not obvious where types, objects, and operations are located unless fully dotted notation is used.

— Increased potential for elaboration order and compilation dependency problems

— Compartmentalization of implementation details, so that Ada visibility rules require unchecked conversion between private types in different packages that have the same underlying representation.

For POSIX.5b, the availability of child packages in Ada 95 offered another choice. The decision was made to take advantage of this new Ada language feature, but to preserve compatibility with POSIX.5 and Ada 83. Therefore, the policy is that if a package has a name of the form `POSIX_XXX` and the implementation supports child packages, `POSIX_XXXX` should be declared as a renaming of `POSIX.XXXX`, a child package of `POSIX`. This renaming scheme mitigates some of the disadvantages of the multiple packages approach.

In POSIX.5c multiple packages were originally defined, organized hierarchically as child packages corresponding to the various POSIX implementation options. For example, the bindings in package `POSIX.Sockets` are applicable when the Sockets Detailed Network Interface option is supported. If, in addition, the Internet Protocol Support option is supported, then the bindings in the child package `POSIX.Sockets.Internet` are applicable. In some cases, there are additional capabilities governed by options at the lowest child package level. For example, the TCP/IP protocol specific bindings in package `POSIX.Sockets.Internet` are applicable only if the Internet Stream Support option is supported. In Draft 4 of POSIX.5c this approach was modified somewhat to also accommodate Ada 83 implementations as described in the original rationale of POSIX.5b. C

B.1.7.3 Coding style

To keep consistency with previous POSIX/Ada binding standards, the same coding style has been maintained through the amendments. In matters of taste/style that involve subjects like spacing, punctuation, blank lines, and comments, consensus is difficult to reach. Using a different style in some sections of an amendment could result in further objections to that section and/or sections of the standard that were not subject to ballot. Therefore, coding style issues were decided by the editor. Also, to enforce a consistent style, the code sections have been test compiled using the Ada 95 style checking options of the Gnu NYU Ada Translator (GNAT).

B.1.7.4 Package sections

The order of the Ada binding closely follows that of corresponding POSIX/C system API standards. Functions are grouped alphabetically by major function/procedure name. Where appropriate, data objects (and any associated access functions and procedures) have been moved from their original location in the POSIX/C system API standards to directly precede the subprograms that use them.

B.1.8 Naming

The POSIX name space is very large. One of the goals of POSIX.5b was to regularize this large name space by following consistent rules for naming types, constants, and operations.

For example, POSIX defines many optional facilities and implementation limits. POSIX.5b introduced a simple set of rules for generating the names of these options and limits, and the names of the associated types, constants, and inquiry functions. However, names used in POSIX.5 do not follow these rules. They have been retained for compatibility; therefore, duplicate names occur in package `POSIX` and in either `POSIX_Options` or `POSIX_Limits`. Also, some duplication occurs within packages `POSIX_Configurable_System_Limits` and `POSIX_Configurable_File_Limits`.

Another example is the naming of operations to retrieve, and sometimes to set, the values of attributes of abstract objects in the interface (*e.g.*, processes, files, semaphores, shared memory objects). The names of such operations are always `Get_`X and `Set_`X, respectively, where X is the name of the attribute.

Any exceptions to the consistent naming rules are carry-overs from earlier POSIX/Ada binding standards. In some cases (*e.g.*, `Change_Owner_Restriction` does not conform to the naming rules for options), this decision was conscious. If other cases, yet to be discovered, do not conform to the rules, it was simple oversight.

For POSIX.5c the C-language names in P1003.1g have been mapped to descriptive Ada names using the styles defined in POSIX.5b. In certain cases, some names that became unreasonably long were shortened using often used abbreviations and acronyms. c

B.1.8.1 Form of Iterators

An iterator is an operator that permits all parts of a structured object to be visited. How iterators should be presented was debated at length during development of

POSIX.5. Programs need to be able to iterate over several kinds of structures, such as string lists (2.4.4), environment lists (4.3.2), directories (5.2.4), and group databases (9.2.2). A form of iteration was desired that would be proper Ada style, easy to use, and consistent among the different parts of the binding.

Two candidates emerged. One candidate consisted of a set of operations to start the iteration, get the next item, test for end, rewind or start over, and close or stop the iteration. The other candidate was a generic procedure. Some programming examples were tried with both approaches, and the generic procedure was preferred.

The generic procedure takes as a parameter a user-supplied procedure and performs iteration on the structured object, applying the user-supplied procedure to each item that is found. This approach is conceptually elegant and results in compact code. Other advantages of the generic iterator are that it is more tasking-safe and easier to program finalization, for example, closing a directory or deallocating storage. The disadvantage is that it may be unfamiliar to programmers who have not used generic units.

Exit from the iterator was studied in detail. The two main ways to exit are via an exception or via setting a `Boolean` return value. The latter approach was chosen, since use of exceptions is generally reserved for error situations. Often, exit from an iterator is not because of an error (for example, exiting when a search completes successfully).

B.1.8.2 Restrictions on Implementation Extensions

The intent of this binding is to permit implementations to provide extensions that provide Ada applications with access to specific services of each particular underlying operating system. However, such extensions should be done in a way that does not break an application that uses only the standard POSIX/Ada interface. To this end restrictions are placed on the form of implementation extensions.

In order to allow **`with`** clause analysis of an application to identify the use of extensions, implementation-defined extensions are expected to be placed in separate implementation-provided packages. No alterations of the package specifications defined in this standard are permitted except those listed in 1.3.1.1.

Also, it is required that if any implementation-defined extensions alter the behavior of interfaces defined in this standard, that effect must not happen unless the user explicitly activates the extension.

B.1.9 Mapping C Features to Ada

B.1.9.1 Error Reporting

Upon failure, the POSIX/C functions either return the value `-1` to indicate that an error has occurred (in this case the error code that indicates the exact nature of the error may be retrieved from *errno*) or eturn the error code itself. Preferred Ada style is to use exceptions to indicate errors. Exceptions are considered to be more reliable since they eliminate the possibility that an application may unintentionally miss errors by forgetting to check the function return value. Thus, operations in this POSIX/Ada binding raise an exception, rather than return an error code. The application may call `Get_Error_Code` to obtain the exact error code.

Two design decisions are worth mentioning here. First, a single exception `POSIX_Error` is used to indicate all possible POSIX errors. Using a single exception for all POSIX errors allows greater flexibility in writing exception handlers to treat errors according to the needs of the application and requires minimal implementation support over and above what Ada already provides. In particular, an application can use of a single exception handler for all POSIX errors (including any implementation-defined errors) and use a handler for **`others`** to catch non-POSIX errors. Second, POSIX error codes are named numbers, rather than enumeration values. The reason is to facilitate the addition of new error codes and also to discourage applications from depending upon any particular ordering of error code values.

One annoying consequence of strict adherence to this general error handling approach is that it makes a common situation rather awkward. It is common practice when using a POSIX/C function to place the call in a loop and then to repeat the call until it succeeds. In Ada, each POSIX call must be placed within an exception frame (to handle `POSIX_Error`) and the exception frame must be placed within a loop. Examples of error codes for which this is likely to be done (since these *errno* values do not represent true errors, but rather the status of the system or of some ongoing operation) are

— `Resource_Temporarily_Unavailable`

— `Operation_In_Progress`

— `Operation_Canceled`

Therefore, special provision has been made in this standard for a few operations that are very likely to be used in such a polling mode. An alternative form of the operation is provided in the form of a function. The return value of the function is an enumeration type, which enumerates the expected kinds of status values that are not actually errors.

For example, the function `Try_Wait` on semaphores and the function `Try_Lock` on mutexes return a `Boolean` value, which is `True` if they succeed and `False` if the semaphore or mutex is busy. Actual errors, such as an uninitialized semaphore or mutex, still cause `POSIX_Error` to be raised. Another example is the function `Get_AIO_Error_Code`, which allows an application to poll the completion status of an ongoing AIO operation that might have terminated in error.

In POSIX.5c C-language functions, procedures, and data structures that are categorized as *obsolescent* in P1003.1g are omitted, including the following functions: *gethostbyaddr(), gethostbyname(), sethostent(), endhostent(), getservbyname(), getservbyport(), setservent(), endservent(),* and *gethostname()*.

Certain C-language data structures have been omitted from the binding when a more appropriate construct is available in Ada. For example, the C *timespec* and *timeval* data structures are replaced with the Ada `Duration` type.

Some coarse timers from the C binding retain an integral type in the Ada binding. They are included because the resolution of these timers is coarse (usually an integral number of seconds or minutes), or because they have an overloaded meaning in the network interface (*e.g.*, the IP Time To Live option is also a gateway hop count). c

Based on additional comments to Draft 3 of POSIX.5c, this approach was refined further. The rationale for not using `Duration` for these time values still holds, since using `Duration` could mislead the programmer by implying a finer control over time values than is available for certain operations. However, these occurrences now use subtypes of the integral time types defined in package `POSIX`. For example, Socket Linger Time (which originally was of type `Natural`) is now implemented as type `Linger_Time`, which is added to `POSIX.Sockets` as a subtype to `POSIX.Seconds` with a natural range. C

B.1.9.2 Extensible Types

The POSIX/C system API standards consider some types to be extensible by a POSIX implementation or by future revisions of the standard. Examples of extensible types include error codes, signal numbers, option sets, permissions, and the character set.

Ada offers three alternatives for defining extensible types:

— Enumeration type, listing the standard values.

— Constrained integer type, with constants for the standard values.

— Private type, with constants for the standard values.

It is readily apparent that enumeration types are unsuitable. Enumeration types cannot be extended without changing the standard package specification. Extending an enumeration type is forbidden in 1.3.1.1, because it introduces new visible names (thus possibly breaking applications that have a **`use`** clause for the package) and a simple **`with`** clause check for uses of implementation-dependent extensions is no longer adequate. There was also concern that an application might depend on the enumeration order, which would not be portable. In order to map enumeration values directly to the representation of the underlying operating system, the ordering of the enumeration type would have to agree with the ordering of the underlying representation (and, therefore, would be implementation dependent. A call to an underlying system service might return a value that cannot be represented in the standard enumeration type (that is, there might be holes in the standard values). Checking for this situation (and then handling the error in a reasonable way) is a burden on implementors and a performance penalty for users. For all these reasons, enumeration types are seldom used in the POSIX/Ada binding.

Integer types are preferred in situations where the values may be used as indices into arrays or for case statements. Also, integer types are convenient for values that must be passed across language boundaries (*e.g.*, from an Ada application to an underlying C-based operating system). Thus, integer types are used for error codes, for signal values, and for file descriptors. To provide some of the convenience of enumeration types for such integer types, value and image functions are provided.

This approach has a small risk. For example, consider `Error_Code`. The intention is that the implementation-defined range includes all values of the type (both POSIX-defined and implementation-defined), but only the names of the POSIX-defined error codes are visible. There is some risk that an application might use an integer value that either represents an implementation extension or a value that is completely invalid (*i.e.*, no meaning either in POSIX or in the implementation.) However, the

application programmer could only use such a value via a facility other than a predefined name (*e.g.*, an integer literal). The current approach seems to provide reasonably strong typing and compile-time checking while still permitting extensions by other POSIX standards-development groups.

Private types are preferred for situations where no functionality of discrete types is desired. For example, array indexing by `Process_ID` was not thought to be common, so this type was made private. Also, the opaque and structure types in the POSIX C-language system API are often mapped to private types in the POSIX/Ada binding. Thus, private types are used for option sets and for the time types *timespec* and *itimerspec*.

B.1.9.3 Private Types

C-language structured types are usually mapped to private types in the POSIX/Ada interface, rather than to explicit record types. The use of private types allows implementors to add extension fields without changing the visible part of the standard package specification. Use of private types prevents the application from using aggregate notation to represent objects of the private types and thereby insulates the user from implementation-defined extensions or from implementation-dependent ordering of the components of the structure.

The alternative of using explicit record types with record representation clauses to map to the underlying C structures is unlikely to work in any case, since there is no guarantee that the Ada records would have the same layout as the corresponding C structures (even with the use of record representation specifications). Also, the POSIX/C system API standards do not specify the order of the components in structures, and allow additional implementation-defined components.

B.1.9.4 Multipurpose C Functions

Multipurpose C functions are decomposed into several Ada subprograms, each of which performs a single, well-defined operation. Sometimes overloading was used, resulting in two Ada subprograms with the same name, but with different parameter-result profiles. For example, the single C function *kill()* is mapped to three separate procedures in the Ada binding, all named `Send_Signal`.

Usually, however, different names are used to suggest the distinct operations being performed. The use of distinct meaningful names is considered to be good Ada style in any case and is required in situations where C functions have a variable number of parameters or have parameter values with special meanings.

B.1.9.5 Composite Return Values

Some declarations used in the POSIX/C system API standards are awkward because of C-language limitations (just as some Ada declarations are awkward because of Ada language limitations). A frequent case is illustrated by the following C function:

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

A C program may take its return value in two ways. The call *getcwd(NULL)()* returns a pointer to a string that contains the current directory. The returned string

is in static memory and will be overwritten by subsequent calls to *getcwd()*. The call *getcwd(mybuf, length)* stores a copy of the current directory in the caller-supplied buffer *mybuf*. The *getcwd()* function is declared this way apparently because of limitations in C. There is no clear way in C to declare a function that returns a string. A *char** function can return either a pointer to an area allocated with *malloc()* or a pointer to an area in static data. The former is considered bad C practice because it wastes space on the heap, and the latter has the disadvantage that subsequent calls overwrite the first value that was returned. As a result, the C declaration was made to give users the choice of supplying a buffer or using static memory.

Unlike C, Ada allows a function to return a composite type, such as a string, without requiring use of the heap or static storage. Thus, in the package `POSIX_Process_Environment`, `Get_Working_Directory` is simply a function returning a `POSIX_String`. The static buffer in *getcwd()* (and, in general, the use of global variables) could cause troubles in multithreaded versions of POSIX.

B.1.9.6 Omitting C Functions

Some POSIX/C functions are omitted from the POSIX/Ada binding because they duplicate existing features of the Ada language. Examples include the following:

— Functionality similar to *sigsuspend()* is provided by the Ada binding to *sigwait()*.

— The capability of *alarm()* is provided by the **`delay`** statement. Also, the Ada runtime system may use *alarm()*, and the application cannot be permitted to interfere.

— The C function *time()* is equivalent to the Ada function `Calendar.Clock`.

— The capability of *sleep()* and *nanosleep()* are provided by the **`delay`** statement.

In the case of POSIX.1c, most of the new C functions already had equivalent facilities within Ada. Thus, the only C functions introduced by POSIX.1c that have bindings defined by this standard are those that apply to mutexes and condition variables and certain thread scheduling functions.

B.1.9.7 Handles

The POSIX/C interface makes extensive use of pointers. In general the POSIX/Ada binding avoids use of Ada access types to correspond to C pointers. One reason is that use of pointers is inherently unsafe; that is, there will always be a chance of dangling references and storage leakage. But, more importantly, it unnecessarily constrains implementations if handles are required to be implemented as access types.

Access types in Ada imply dynamic (heap) memory allocation. Realtime applications avoid any use of dynamic, if possible, because of the possibility of runtime errors that cannot be detected and removed by prior testing and because unpredictable response times may occur due to storage recovery being performed at unexpected times.

Realtime implementations of POSIX, especially those conforming to the Minimal Realtime profile of POSIX.13, may avoid dynamic allocation by preallocating a limited number of system resources in an array data structure. The C pointer would then correspond to an index into the array in the Ada implementation.

Other implementations may choose to use dynamic allocation in order to avoid imposing such hard *a priori* limits on system resources. For these implementations, the C pointer type would indeed correspond to an Ada access type.

To allow for both of these implementation choices (array indices and access types) for handles to system objects, the Ada binding uses descriptors declared as private types.

Every occurrence of a C pointer parameter potentially corresponds to two objects in the Ada interface: the handle or descriptor itself, and the object being referenced by the handle or descriptor. Sometimes, both are made visible — as in the case of semaphores. (See B.11.1.1 for the reasons.)

```
type Semaphore is limited private;
type Semaphore_Descriptor is private;
function Descriptor_Of(Sem: Semaphore) return Semaphore_Descriptor;
```

If the underlying object is exposed (*e.g.*, `Semaphore`), it is always an Ada limited private type. The descriptor objects are private to allow copying. Given an object (*e.g.*, `Semaphore`), the function `Descriptor_Of` returns a handle that can be used in subsequent operations to refer to the object.

At other times, only the handle is needed, as in the case of AIO control blocks. Then the Ada binding omits the underlying object type declaration and the function `Descriptor_Of`. Thus, for AIO control blocks, only the descriptor is visible in the interface:

```
type AIO_Descriptor is private;
```

In some cases, such as I/O buffers, the proper semantics are achieved by mapping a usage of a C pointer type to an Ada access type, but such cases are rare.

B.1.9.8 Avoiding Storage Leakage

The POSIX/Ada interface has been designed to try to avoid dynamic storage allocation wherever possible. In some cases, for example, AIO control blocks, the use of dynamically allocated storage seems unavoidable. However, most instances of descriptors offer the prospect of avoiding heap allocation, as discussed in B.1.9.7.

In every instance where use of dynamic storage allocation might be chosen by an implementation, separate operations to allocate and recover storage have been provided, usually with pairs of names such as `open` and `close`, `initialize` and `finalize`, or `create` and `destroy`. An application that calls the recovery operation (*e.g.*, close, finalize, or destroy) before exiting the scope of any objects created using the allocate operations (*e.g.*, open, initialize, or create) is guaranteed that any storage dynamically allocated by the implementation will be reclaimed.

These explicit allocator and deallocator operations are provided by the standard in order to minimize storage leakage and to protect the integrity of implementation data structures from inadvertent corruption by the application. However, they do leave the possibility of dangling references. Minimizing the danger of dangling references for the application was considered of secondary importance for two reasons. First, disciplined programming practices can avoid the problem altogether at the application level. Second, even if dangling references do occur and lead to erroneous

execution of the application, recovery may still be possible. However, the corruption of implementation data structures could render effective recovery impossible.

B.1.10 Conformance

The classes of conformance and the corresponding requirements are generally the same as in the POSIX/C system API standards, with appropriate changes in terminology for Ada. However, because of the interactions of certain Ada language features with features of the POSIX interface, support of this standard also imposes requirements on the Ada language implementation, as are enumerated in 1.3.1.1.

B.1.11 Year 2000 Compliance

This standard is intended to contain no specifications that conflict with Year 2000 requirements. In particular, the time types and interfaces defined in this binding are sufficient to represent times and dates within and between the 20th and 21st centuries. Further definition or specification of Year 2000 compliant features or requirements is outside the scope of this standard.

B.2 Terminology and General Requirements

B.2.1 Conventions

This clause is similar to the corresponding clause in POSIX.1, with typographical conventions changed to be appropriate for the Ada language.

B.2.2 Definitions

This clause is similar to the corresponding clause in POSIX.1. Some of the terms and definitions have been revised for consistency with the terminology of the Ada RM {1}. Some new definitions have been added, specific to the Ada language. Several of these are repeated verbatim from the Ada RM {1}.

B.2.3 General Concepts

This clause is similar to the corresponding clause in POSIX.1, with several additions unique to Ada.

B.2.3.1 Process/Active Partition Equivalence

The developers of POSIX.5 were confronted with a fundamental choice. Which Ada construct corresponds most closely to a POSIX process? Two Ada 83constructs were candidates: an entire Ada program or an individual Ada task. There were several reasons for deciding that an entire Ada program corresponds to a POSIX process. First, Ada tasks share some execution context, which might include process-level attributes, such as system resources held and virtual address mappings. Requiring this execution context to be shared across POSIX process boundaries would impose extra overhead on task switching and might not even be possible on some systems. Second, tasking interactions, such as rendezvous semantics, would impose extra requirements on process interactions that might not be supported on some systems. Third, existing practice for Ada implementations on UNIX systems typically equated

Ada programs to processes. Thus, the model adopted by POSIX.5 was that all Ada tasks within an Ada program *appeared* to execute within a single POSIX process.

This model does not prohibit implementations from creating additional hidden processes in order to improve application performance. For example, some implementations might create a hidden I/O server process to handle all synchronous I/O operations for the Ada program so that when one task is blocked on an I/O operation, the other tasks in the same process are still free to execute. All that is required is that the implementation ensure that the Ada program behave as a single POSIX process for operations of the POSIX/Ada interface.

Ada 95 requires a change to this model. An Ada program may consist of several active partitions. Each active partition has a main subprogram and may have dependent tasks. Active partitions of the same Ada program may execute concurrently, if sufficient processing resources are available. The model adopted by POSIX.5b is that an Ada active partition corresponds to a POSIX process and the environment task of the active partition corresponds to the initial thread of control of the process.

B.2.3.2 Task/Thread Equivalence

POSIX.1c introduced the concept of multithreaded POSIX processes. The obvious mapping is that an Ada task corresponds to a POSIX thread as defined by POSIX.1c. This works at the conceptual level. However, there are significant differences in the details of the behavior of tasks and POSIX threads, so that it is not possible to identify these constructs with each other at the implementation level. In other words, given the implementation freedoms allowed by the Ada language standard and by POSIX, it is not possible to guarantee that Ada tasks having the behavior required and permitted by the Ada RM {1} can be implemented using the facilities for POSIX threads defined by POSIX.1.

Where there may be differences in the detailed behavior between Ada tasks and POSIX threads, this standard regards such differences as language-specific. The philosophy has been that tasks are the Ada expression of the POSIX abstraction of light-weight concurrency within a process, and POSIX.1c threads are the C-language expression of the same abstraction. It is clear then that any C threads functions that provide facilities already present in Ada need not lead to new operations in this binding, unless there is overriding reason to do so.

B.2.3.3 Tasking-Safe Operations

All operations in the POSIX/Ada interface are intended to be tasking safe, unless specified as tasking unsafe. *Tasking safe* means that the effect of the operation shall be as specified by this standard, even if it is executed in a multitasking environment where other tasks are concurrently executing the same or other operations specified by this standard.

The only operations currently marked tasking unsafe are the operations in package `POSIX_Unsafe_Process_Primitives`.

The I/O operations defined in Annex A of the Ada RM {1} are not guaranteed to be tasking safe. This standard does not address that issue.

B.2.3.4 Interruptible Operations

Some POSIX operations may return prematurely if a signal is delivered and handled, whether by an application signal handler (as for C programs) or by the Ada language implementation. In the Ada case, the interrupted operation would propagate the exception `POSIX_Error` with the error code `Interrupted_Operation`. The application would ordinarily want to recover from this exception and restart the operation. Thus, each call to an interruptible POSIX operation would have to be placed within an exception frame, and the exception frame would have to be placed within a loop, as follows:

```
loop
   begin
    Some_POSIX_Interruptible_Operation(...);
      exit;
   exception
      when POSIX_Error =>
         if POSIX.Get_Error_Code /= Interrupted_Operation then
            raise;
         end if;
   end;
end loop;
```

The need to provide a loop such as the one above seems awkward for the user, especially since the application must assume some signals are in use by the Ada language implementation (*i.e.*, reserved signals) and, therefore, every interruptible operation is likely to be interrupted. This problem is compounded since the restarted operation may be interrupted again and again, causing the application to loop for an unbounded amount of time. Interruptions are especially likely if the operation is one that takes a long time to complete and if the Ada runtime system is using a periodic alarm signal, *e.g.*, to do time-slicing on a system that does not have direct operating system support for time-sliced scheduling of threads.

For this reason, an extra `Masked_Signals` parameter has been provided for interruptible operations, which specifies the signals that are to be blocked during the operation. (See B.2.4.6.)

B.2.3.5 Configurable Limits and Options

The POSIX/Ada interface involves a number of options and implementation limits. (See 2.5.1 and 2.6.1.) Corresponding to each option and limit usually is a static subtype indicating the range of implementation behavior permitted and a runtime callable function that returns the actual implementation behavior. (See Tables 2.4, 2.6, 4.1, and 4.2.) In addition, corresponding to each limit is a compile time constant that specifies the portable value for that limit (see Table 2.5).

Generally, the value of an option or a limit is system wide. However, for some options and limits related to files, the option or limit is pathname-specific (see 5.4.1, 5.4.2, and 5.4.3). In other words, the value of the option or limit may vary, depending on the pathname of the file (*i.e.*, where it is located in the file system).

The POSIX/Ada implementor declares the static subtype, for example, `Open_Files_Maxima`, to correspond to the lowest and highest values that might be configured on the target POSIX system(s). Thus, `Open_Files_Maxima'First` must be

at least equal to `Portable_Open_Files_Maximum`, but might be greater. `Open_Files_Maxima'Last` might be the limit for the target POSIX system, if it is known, or might be as high as `Natural'Last`.

A portable application may rely only on the portable limit, in this case, `Portable_Open_Files_Maximum`. Alternatively, it may query the runtime function, `Open_Files_Maximum`, and modify its behavior dynamically based on the value returned.

The implementation is required to guarantee the same portable limits as the POSIX C-language system API standards. The treatment of limits is unlike the treatment of signals: certain signals are reserved for the Ada language implementation, but the Ada language is not allowed to take away from the application any resources that would result in an actual limit being less than the specified portable limit. A consequence is that the Ada runtime system is prevented from using certain features internally on platforms that only support the minimum portable values. Then alternative, which is to reserve an arbitrary amount of each limited system resource for possible use by the Ada runtime system, seems less desirable, since the amount of a resource needed by the Ada runtime is not known, and will be zero in many cases. Thus, the arbitrary amount reserved is likely to be either too little or too much.

The relatively straightforward approach in the POSIX/Ada binding for treating configurable limits and options may appear more complicated to readers than it actually is because of two intruding design issues: naming and packaging.

Due to the very large number of new limits and options, and the prospect that future extensions to POSIX will add even more, POSIX.5b introduced systematic naming rules for the constants, static subtypes, and configurable limits and options. These naming rules are as follows:

— Static subtypes for options have the same name as the corresponding option, but with `_Support` appended.

— Static subtypes for limits have the same name as the corresponding limit, but with `Maximum` replaced by `Maxima`.

— Configurable functions for options have the same name as the corresponding option, but with one of the suffixes `_Is_Supported` or `_Are_Supported`.

— Configurable functions for limits have the same name as the corresponding limit.

— Constants for the portable values of limits have the same name as the corresponding limit but with the prefix "`Portable_`".

— Blanks in the names of options and limits are replaced with underscores.

Unfortunately, POSIX.5 did not follow these naming rules in all cases, so that this standard now has duplicate names for many constants, static subtypes, and configurable functions. The old name was retained for compatibility with POSIX.5, but is obsolescent; and the new name is provided for future use.

The packaging structure of POSIX.5 placed all constants and static subtypes for limits and options in package `POSIX`. However, because of the many new limits and options added by POSIX.5b, it was decided to split all options and limits out into two new packages. `POSIX_Options` and `POSIX_Limits`. POSIX.5b retains the old names in `POSIX` for compatibility with POSIX.5s, and places the new names for options in `POSIX_Options` and the new names for limits in `POSIX_Limits`. POSIX.5

already had placed the runtime callable functions for options and limits in separate packages; `POSIX_Configurable_System_Limits` contains functions for options and limits that are not pathname-specific. `POSIX_Configurable_File_Limits` contains functions for options and limits that are pathname-specific. POSIX.5b merely adds the new names to these packages.

Thus, the placement rules for constants, static subtypes, and configuration functions having to do with options and limits may be summarized as follows:

— Static subtypes for limits and constants for portable values of limits are in package `POSIX_Limits`.

— Static subtypes for options are in package `POSIX_Options`.

— Configuration functions for options and limits that are not pathname-specific are in package `POSIX_Configurable_System_Limits`.

— Configuration functions for options and limits that are pathname-specific are in package `POSIX_Configurable_File_Limits`.

B.2.4 Package `POSIX`

The package `POSIX` contains a common core of declarations that are used by the other POSIX/Ada interface packages. For implementations that support child packages, all other POSIX/Ada interface packages are permitted to be implemented as child packages of package POSIX. (See 1.3.1.1.)

B.2.4.1 Options and Limits

The preferred and complete interfaces for obtaining information about whether an implementation supports an optional feature are defined in the package `POSIX_Options`.

The preferred and complete interfaces for obtaining information about implementation limits are defined in the package `POSIX_Options`.

However, for upward compatibility, redundant interfaces for obtaining information about optional features and limits defined in POSIX.5 are still provided in the package `POSIX`. These interfaces are obsolescent.

B.2.4.2 Blocking Behavior Values

Ideally, POSIX operations that block the calling task should not block any other tasks in the process. However, per-task blocking of all system calls is impractical to achieve for some system calls on some operating systems. Therefore, to require per-task blocking for all implementations of the Ada binding would prevent use of the Ada binding on some systems.

This situation is analogous to the provision in 1.1.3(6) of the Ada RM {1} that allows an implementation to impose implementation limitations on a particular execution platform, providing that the implementor justify why it is impossible or impractical to remove the limitation. Since POSIX does not have such a rule, the best approximation seems to be to require that blocking behavior be per task, wherever per-task blocking can be implemented using the underlying system. An implementor would

be required to document the reasons for any cases in which the blocking behavior is otherwise.

Two blocking behavior values were defined by POSIX.5 to specify the two extremes of behavior:

— `Tasks`: only the calling task is blocked, reliably.

— `Program`: all tasks within the process are blocked, reliably.

A new blocking behavior `Special` was added by POSIX.5b for implementations where several Ada tasks are executed by a single system-wide schedulable entity (*e.g.*, light-weight process, kernel thread, virtual processor). In this case, blocking of the schedulable entity may block several tasks, but not all tasks, or there may be a limit beyond which blocking of another task blocks all the remaining tasks in the process.

The blocking behavior for classes of POSIX operations are defined in package `POSIX` by either static subtypes or constants of type `Blocking_Behavior`. POSIX.5 defined blocking behavior for the following classes of operations:

— Ada `Text_IO` operations.

— POSIX synchronous I/O operations.

— File locking.

— Wait for child.

POSIX.5b added only a new blocking behavior, subtype `Realtime_Blocking_Behavior`, which applies to all the realtime operations added by POSIX.5b. The alternative was to have individual blocking behaviors for the separate options (*e.g.*, semaphores and message queues). Such fine-grained control over blocking behavior was deemed unnecessary, since applications are not likely to exploit it and implementations are not likely to provide different blocking behavior for the different realtime options.

The anticipated ranges for subtype `Realtime_Blocking_Behavior` and their meanings are

— `Tasks..Tasks`: Realtime operations may be relied upon not to block more than the calling task.

— `Program..Program`: Some realtime operations may block all tasks within the process; consult the conformance document for specific information.

NOTE: Some operations are required to support task blocking behavior regardless of the range of `Realtime_Blocking_Behavior`.

— `Tasks..Special`: Some realtime operations may block some or all tasks within the process; consult the conformance document for specific information.

Other ranges are not anticipated since they would imply that all realtime operations block additional tasks as well as the calling task.

B.2.4.3 Characters, Bytes, and I/O Units

POSIX and C are really based on byte-oriented file systems. The C *char* type, bytes, and the I/O units upon which input/output operations are performed are really all the same in POSIX. File names and user names are arrays of *char*s, and file contents are streams of *char*s. POSIX.5 defined a single type `POSIX_Character` to represent all these objects. The intent was that the type `POSIX_Character` correspond to the C *char* type, and that implementations would extend it to include all possible byte values of the underlying system.

However, in the POSIX.5 ISO fast-track ballot and in the ballot of POSIX.5b, objections were raised to the use of `POSIX_Character` in contexts where POSIX.1 uses byte, mainly as a unit of untyped I/O data or or as a unit of memory. These objections touched on several issues:

— The trend toward internationalization of character sets and the associated data types implies that a graphic character is not always represented by a single byte of data.

— Concerns that the mapping of `POSIX_Character` to a memory representation might leave holes, *i.e.*, bit patterns that do not correspond to characters.

— Concerns that an Ada implementation might impose some overhead on the conversion of character values to other types of data.

— The provision in Ada 95 of standard marshalling and unmarshalling operations (`'Write` and `'Read`) for streams, which are based on the data type `Stream_Element`. It is clearly desirable to be able to use these operations in conjunction with the POSIX message passing and I/O operations, especially when complex data objects are being transmitted.

The possibility was considered of just reemphasizing the original intent of POSIX.5 that the values of the type `POSIX_Character` are just bytes; that is, each character is required to fit in one byte of storage and the range of values is required to cover all valid one-byte bit-patterns. Conversion operations between `POSIX_String` and `Stream_Element_Array` could be added. But, the use of such conversion operations was unappealing because of the ample evidence from balloters that the word `Character` in the type name led them to think of this type as a conventional character type, rather than as a byte type. There was also concern that the conversion functions might impose execution time overhead.

Therefore, the terminology has been changed, and a new data type has been introduced by POSIX.5b. Uses of POSIX character as a unit of data have been replaced by the term *byte*. The new Ada type `Stream_Element_Array` has been taken from Ada 95. New I/O operations (*e.g.*, message passing and AIO) are defined only on the new type. The I/O operations on the `POSIX_String` type are retained, for compatibility with POSIX.5, but also are overloaded on the new type.

The type `POSIX_Character` is retained for use in contexts where a string is needed, such as for file names or pathnames. However, one of the main reasons for distinguishing this type from the standard Ada type `Character` no longer holds since Ada 95 no longer restricts type `Character` to 7-bit ASCII values. Therefore, the possibility of requiring the type `POSIX_Character` to be derived from the standard

`Character` was considered. However, this change was not made, because it would mean that other native character sets, such as EBCDIC, would no longer be acceptable for `POSIX_Character` on Ada 95 implementations.

The lower bound of all values of type `POSIX_String` that are returned by explicit operations of the POSIX/Ada interface is the value 1, except for operations on the type `POSIX_String_List`. Appending a string to a `POSIX_String_List` preserves the bounds of the string so that subsequent retrieval operations return the identical string.

B.2.4.4 String Lists

A special problem occurs with command line argument lists and `Environment` variables. Most uses of these types just look up their values, so the binding could just provide functions and show no visible string list type at all. But some applications also need to build these lists, typically just before passing them to a new process. Should there be a simpler interface for the majority of applications that just look up values, or should there be a combined interface that allows for both looking up values and for creating new lists? The latter course was chosen.

There is a string list type in package `POSIX`. There are additional declarations that use this private type, for example, in `POSIX_Process_Environment`.

The string list type is not a direct mapping of the C type *char* **. Instead, the Ada binding provides a limited private type `POSIX_String_List`; an iterator over objects of this type; and operations to build, search, and manipulate POSIX string lists. This string facility is sufficient to support the needs of the POSIX/Ada interface.

The `POSIX_String_List` operations do not provide a more general list processing capability because it might be desirable for a system to implement these string lists in a way that optimizes their use with the POSIX operations, but would be less efficient for general use.

B.2.4.5 Option Sets

The type `Option_Set` and its operations provide a general set facility for specifying options to POSIX/Ada operations having an `Options` parameter. The `Options` parameter corresponds to the C *flags* parameter. The `"+"` operation is equivalent to the C-language `"|"` operator, and the binary `"-"` is equivalent to the C-language `"&~"` for removing options from the set. The function `Empty_Set` returns a set with no options set. `Empty_Set` is a function, rather than a constant, to ensure that the it is inherited for all types derived from `Option_Set`.

Packages with option sets (*e.g.*, `POSIX_IO`) derive a new type (*e.g.*, `Open_Option_Set`, described in 6.1.1) from `Option_Set`, inheriting all the set operations from package `POSIX`. They declare constants of this new type, using the constants provided in `POSIX`, to represent the individual options available (*e.g.*, `Non_Blocking` and `Exclusive`) The inherited operations and constants permit the end user to write option set expressions like the following:

```
Options => Non_Blocking + Exclusive;
```

For other examples of this technique, see type `Synchronize_Memory_Options` in package `POSIX_Memory_Mapping` (12.3.4) and type `Message_Queue_Options` in package `POSIX_Message_Queues` (15.1.1).

Another advantage of this approach, the primary reason for its adoption, is that it allows implementation extensions. An implementation can define additional options (in an implementation-defined package) by defining new constants of type `POSIX_IO.Open_Option_Set`, for example.

`Option_Set` is a private type in order to allow some implementation freedom. The most likely implementation is either a fixed-length packed array of `Boolean` or an integer. In the former case, the operations on `Option_Set` can be implemented using predefined operations on `Boolean` types and arrays of `Boolean` values provided by Ada. In the latter case, an implementation can use its facilities for manipulating integers as bit masks, similar to facilities in the C language.

In order to meet the requirement that objects of type `Option_Set` should be implicitly initialized to the empty set, the underlying representation is likely to be placed into a record type to allow use of Ada default initialization for record components.

The deferred constants with names beginning with "`Option_`" are provided to allow implementors of other POSIX packages where types are derived from `Option_Set` to define constants for singleton option sets in terms of these constants without the need to resort to `Unchecked_Conversion`.

B.2.4.6 Masked Signals Parameter

A `Masked_Signals` parameter is added to POSIX/Ada operations that might be interrupted by a signal in order to give the application the ability to prevent a call from being interrupted by the reserved signals, which cannot ordinarily be masked by the application.

For example, in package `POSIX_IO`, the file close operation is declared as follows:

```
procedure Close
   (File:           File_Descriptor;
    Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals);
```

Three values may be specified for `Masked_Signals`: `No_Signals`, `RTS_Signals` (always the default), or `All_Signals`. These values have the following meanings:

— `No_Signals`: No signals shall be masked other than those that already have been masked.

— `RTS_Signals`: Signals normally used by the Ada runtime system shall be masked during the operation and shall not cause it to be interrupted.

— `All_Signals`: All signals, including implementation-defined signals, are masked during the operation.

A `Masked_Signals` attribute is also provided for process creation templates since process creation can also involve interruptible operations.

This three-value enumeration was chosen instead of a signal set because it is not intended as a substitute for the operations that block and unblock a set of signals. The

purpose is to cover signals that cannot otherwise be masked, namely, the reserved signals. The three choices provided are adequate for most applications. If an application needs finer-grained control (for signals that are not reserved signals), it can use the facilities of package `POSIX_Signals`. Another reason for not allowing the specification of a set of signals here is that it would require moving the declaration of the signal and signal set types from package `POSIX_Signals` to package `POSIX`.

The POSIX/Ada operations that have the `Masked_Signals` parameter correspond exactly to the POSIX/C functions that can return with *errno* set to [EINTR].

B.2.4.7 Time Types

The C-language type *time_t* is mapped to the Ada type `Time` in package `Calendar`. The C-language type *clock_t* is mapped to the Ada type `Tick_Count` in several places (see 4.2).

The POSIX time-zone **TZ** functionality included in Section 8 of POSIX.1 is placed in Section 4 of this standard. Since **TZ** should not be used to modify the behavior of the predefined `Calendar` package in Ada, an analog to the Ada predefined package `Calendar` is provided. This package provides the same functionality, but applies the information contained in the **TZ** environment variable to a value of type `POSIX_Calendar.Time`. This package also provides operations to convert between values of `Calendar.Time` and `POSIX_Calendar` `POSIX_Time`.

The C-language structure *timespec* is mapped to the private type `Timespec` in the package `POSIX`. This type is placed in `POSIX` rather than in `POSIX_Timers` because it is used as the type of a parameter of function `Await_Signal_Or_Timeout` in package `POSIX_Signals`. Since `POSIX_Timers` depends on `POSIX_Signals`, putting `Timespec` in `POSIX_Timers` would create a circular compilation dependency. Moreover, it is reasonable to expect that `Timespec` may find uses in future extensions of other POSIX interfaces.

The idea of providing separate types for relative and absolute time similar to `Ada.Real_Time` in Ada 95was considered. The decision to use a single type is based on simplicity and consistency with POSIX.1.

The possibility of making `Timespec` a visible record type with explicit component fields for `Seconds` and `Nanoseconds` was considered. The C-language interface specifies *timespec* as such a structure. Making `Timespec` a visible record type was rejected on the grounds that the underlying system representation of time might not be the same as specified in POSIX.1b. For example, in an embedded realtime system, it is likely to be an integer count of clock ticks. Forcing the Ada user to go through the C-language representation might force extra type conversion and cause significant overhead. (A prototype implementation on a 86040 single board computer showed that the extra overhead took ten times as long as just reading the realtime clock.) Moreover, arithmetic on the two-part C-language *timespec* structure is much slower than on a 64-bit integer value. Making `Timespec` a private type allows the implementation to make direct use of the best representation of time to achieve maximum efficiency.

Another alternative considered was to use `Duration` directly. `Duration` was not used because of concern that it would impose too much overhead on nonrealtime ap-

plications, for some Ada implementations. If `Duration` is represented as a 32-bit signed integer, it is not possible to satisfy both the POSIX requirement for nanosecond precision and the Ada requirement for a range of plus or minus one day. But, if `Time` and `Duration` are both represented as 64-bit integers, there would be extra overhead for conversion between the Ada representation and the C representation. Moreover, supporting such types would imply support for general 64-bit fixed-point arithmetic, which was not yet common at the time POSIX.5 was balloted.

Operations are provided to convert between a value of the private type `Timespec` and the equivalent counts of whole seconds and nanoseconds. `Split` and `To_Timespec` are provided as well as the operations for just seconds and just nanoseconds. Likewise, mixed addition and subtraction are provided for `Timespec` with `Nanoseconds` to avoid imposing overhead of going through unnecessary widening conversions on the operand of an operation that is expected to occur in time-constrained contexts.

The conversion functions for `Calendar.Time` are separated from the package `POSIX`, so that the package `POSIX_Timers` does not depend on `Calendar`. Avoiding this dependency allows the possibility that `Calendar` might be implemented using `POSIX_Timers`.

The `Nanoseconds` type was not made a fixed-point type for two reasons. First, some Ada compilers do not permit exact specification of decimal `Small` values for fixed-point types. If the compiler does not support decimal `Small` values there might be error introduced in converting between true nanoseconds and the fixed-point type. Second, the Ada semantics for fixed-point types would require the implementation to support multiplication and division operations with greater precision than is required for time calculations, resulting in unnecessary overhead for the implementation.

The constant `Portable_Clock_Resolution_Minimum` is a named number, rather than a constant of type `Nanoseconds_Base`, since using the type `Nanoseconds_Base` would force the declaration of the time types to precede the portable constants in `POSIX`.

B.2.5 Package `POSIX_Options`

For POSIX.5b, the static subtypes for options were taken from `POSIX` and placed in `POSIX_Options` because of the large number of options introduced by POSIX.1b and POSIX.1c. The old interfaces are retained in `POSIX` for compatibility with POSIX.5, but are obsolescent. `POSIX_Options` is the preferred interface for obtaining information about the options supported by an implementation.

B.2.6 Package `POSIX_Limits`

For POSIX.5b, the portable constants and the static subtypes for limits were taken from `POSIX` and placed in `POSIX_Limits` because of the large number of additional implementation limits introduced by POSIX.1b and POSIX.1c. The old interfaces are retained in `POSIX` for compatibility with POSIX.5, but are obsolescent. `POSIX_Options` is the preferred interface for obtaining information about the limits imposed by an implementation.

B.2.7 Package Ada_Streams

`Ada_Streams` is a renaming of the Ada 95 package `Ada.Streams`. The type `Stream_Element_Array` is used in this standard to represent an array of bytes for I/O operations.

B.2.8 Package System

To make effective use of the memory management operations defined in Section 12, operations on the type `System.Address` are required, beyond those required by Ada 83. Therefore, this standard imposes some specific requirements on this data type from Ada 95, whether or not the Ada compilation system fully supports Ada 95.

B.2.9 Package System_Storage_Elements

Some interfaces defined in this standard, such as memory management and low-level I/O operations, require facilities of Ada 95 that are defined in the package `System.Storage_Elements`. An implementation of this standard is required to support these features, whether or not it fully supports Ada 95.

To provide a transition path from Ada 83, the package `System.Storage_Elements` is renamed as `System_Storage_Elements`. Permission is given to replace this renaming by an implementation-provided package specification and body that provide equivalent functionality.

B.2.10 Package POSIX_Page_Alignment

The package `POSIX_Page_Alignment` provides additional operations, beyond what are available in `System_Storage_Elements`, that make it convenient to compute address and file offsets that are page-aligned. These additional operations are useful in writing portable applications because implementations may require that some of the arguments of the memory management operations be page-aligned. Operations are also provided to adjust a length to the exact number of pages that will need to be locked/mapped for an object with a specified length and starting address.

For example, suppose a user wishes to map an object `X`. The address of `X` is given by `X'Address`. The size of `X` in bits is given by `X'Size`. To specify the range of addresses occupied by this object in a portable way, the address needs to be rounded down to the nearest page boundary, and the length needs to be adjusted upward to reflect the effect of rounding down the starting address. This is accomplished by using `Truncate_To_Page(X'Address)` for the page-aligned starting address and `Adjust_Length(X'Address, X'Size/System.Storage_Unit)` for the page-aligned length of the region.

The exception `Program_Error` was chosen for errors in these operations, for consistency with the Ada 95 treatment of errors in address arithmetic.

B.2.11 Environment Description

This section is very similar to Clause 2.6 of POSIX.1. It is important to maintain a direct mapping of environment variables between C and Ada since environment vari-

ables are used to communicate between C and Ada programs and between programs and shell commands.

B.2.11.1 Error Codes and Exceptions

The rationale for the general error reporting model is explained in B.1.9.1.

In POSIX.5b, linkage is made between the `Exception_Message` feature of Ada 95 and the POSIX error codes.

Implementations are free to raise `POSIX_Error` for implementation-defined reasons and may also add additional error codes for implementation-defined errors.

In addition, no Ada name corresponds to EDOM (domain error) since that is the C-language equivalent of the Ada exception `Constraint_Error`.

B.2.11.2 System Identification

These functions are a de-multiplexing of the C-language *uname()* function.

B.3 Process Primitives

B.3.1 Package `POSIX_Process_Primitives`

This standard specifies that an Ada active partition corresponds to a POSIX process so that process-level POSIX operations affect all tasks within the active partition. This choice seems to cause the least conflict between Ada language rules and POSIX, and it resolves several binding issues cleanly:

— `Exec` and `Start_Process` are well- defined as producing a new active partition execution.

— `Exit_Process` is well defined as ending an active partition execution.

— The current process ID is well defined.

— The bindings of file descriptors (and the descriptors of other file-like objects, *e.g.*, named semaphores and message queues) are well defined over the entire active partition.

— All process-specific state information is well- defined throughout a program, including process environment variables and system accounting times.

This presumed correspondence between active partitions and POSIX processes does not preclude the implementation from creating hidden processes to support more efficient execution of applications. It merely requires that these hidden processes not destroy the illusion that an Ada active partition is a single POSIX process with respect to the operations defined by this standard.

To help contain some of the problems that might happen, this standard provides a higher level template-based process creation operation, as well as an Ada binding to the unsafe C-language operations *fork()* and *exec()*.

B.3.2 Process Creation

The ability to create processes is an essential part of the POSIX interface; an Ada binding to POSIX without process creation would be incomplete. The POSIX process creation operations are isolated within the packages `POSIX_Process_Primitives` and `POSIX_Unsafe_Process_Primitives` in order to allow easy detection of their use by checking for the names of the packages in **`with`** clauses.

For creation of processes, the process template approach was chosen, in preference to the apparently simpler

fork() ... arbitrary code ... *exec()*

approach, because it is necessary for semantic reliability when a process has multiple threads of control. Permitting arbitrary Ada code to be executed after a *fork()* operation leads to numerous difficulties. The use of the `Process_Template` eliminates such problems since the actions possible between logical `Fork` and `Exec` operations are limited to those with clear single thread meanings. This approach was adopted from the approach described in McJones and Swart {B11}.

Certainly, applications may require *fork()* and *exec()* operations that are not paired, *e.g.*, a *fork()- fork()- exec()* sequence to create a detached process, or an *exec()* without a *fork()* to chain execution. Such special requirements can be satisfied using the `Fork` and `Exec` procedures from the package `POSIX_Unsafe_Process_Primitives`. Use of that package is appropriate because such applications are unlikely to be portable across all POSIX/Ada implementations.

Likewise, some applications may require changes to the process state, between `Fork` and `Exec`, that are not supported by the process template approach. The example cited most often is writing a shell program. It was decided not to extend the template model to support such applications. Few applications execute another program. Of those few, very few do any operation other than to create a pipe to a general system utility, which can be handled via mechanisms in file opening. Of those few that do program execution more complex than creating a pipe, very few do the full shell-type operations; typically, file redirection and establishing sensitivity to signals is all that is done.

The general philosophy is that the process creation template is the preferred way to create a new process for the simple situations that are likely to be found most commonly in portable Ada applications, but that it does not need to cover all situations. The unsafe operations are available for the remaining cases. This is the reason the templates specify only a few key process attributes. Also, since templates are not intended to be reused frequently with minor variations, there is no need for operations to extract the values of template attributes.

The procedure `Set_File_Action_To_Close` is needed because there is no way to close open POSIX files except with this action. Only the parent process knows the file descriptors and which ones of these should be open in the child process. The child process cannot close the others because there is no query function to find out all currently valid file descriptors. In the C-language interface the corresponding capability is provided by the FD_CLOEXEC flag, which can be set via *fcntl()*.

One concern when using templates is the danger of dangling references and storage leakage. The application is responsible for finalizing templates so their storage can be reclaimed. A negative consequence is that default values cannot be provided for operations that take templates as arguments. The implementation is encouraged, but not required, to detect invalid references to closed templates because it may not be possible to perform such checks reliably.

One deviation of `Start_Process` from the `Fork`/`Exec` pair is that the effective user and group IDs of the child process are taken from the real (rather than the effective) user and group IDs of the parent. Using the real user and group IDs is done for increased safety, *e.g.*, to reduce the risk of accidental transmission of privileges.

In order to permit the intentional transmission of privileges, the option of keeping the effective user and group IDs is provided as an attribute of the template that can be set. This template feature may be overridden (independently) by the permissions `Set_User_ID` and `Set_Group_ID` of the file. A process may actually go through three sets of effective user and group IDs during start-up:

— The effective user and group IDs of the parent, as inherited through the `Fork` operation.

— The real user and group IDs, as reset through the effect of the template (immediately after the `Fork` and before other process-state changes specified by the template).

— The effective user and group IDs of the file, as based on the permissions `Set_User_ID` and `Set_Group_ID` of the file, as part of the `Exec` operation.

If `Fork` is used to create a new process, the child can use the environment operations to modify its environment.

One awkward aspect of `Start_Process` is that if initialization of the new process fails after the `Fork` step, but before the `Exec` step, it is too late to return an error to the creator process. For this reason, the special exit status value `Failed_Creation_Exit` is provided. It is not possible for `Start_Process` to precheck for all error conditions that could be anticipated (*e.g.*, insufficient memory to load the new process image or concurrent deletion of the file to be executed).

B.3.3 Process Exit

The exit status of a process can be used to transmit information between programs written in different languages. Therefore, the type `Exit_Status` is defined to be an integer with the range 0..255 rather than a private type.

Ordinarily, exit from an Ada program is automatic due to normal completion or an unhandled exception. Standard exit status values are defined for two cases, plus the case of failed process creation. The choice of the value zero for `Normal_Exit` is traditional in UNIX systems. The choice of values for `Unhandled_Exception_Exit` and `Failed_Creation_Exit` is arbitrary, subject to the constraint that it must not be near zero or above 127 (but see *Hitchhikers Guide* {B1}). Values near zero are likely to be already in use for special purposes. Extreme high values have the same problem as low values (since they are equivalent to negative values near zero). Values

above 127 are conventionally used by UNIX system C-language applications for processes killed by signals. This practice is intentionally not followed by this standard if the signal is converted by the Ada language implementation into an exception, which propagates to the end of the active partition and causes it to terminate. In such a case, the exception is no longer a signal. Following the C-language convention for processes terminated by signals would, therefore, be misleading and incorrect. Moreover, since the same exception might be raised for other reasons, it would be an excessive burden on Ada implementations to remember whether an exception was caused by a signal and to split the exit processing into two cases accordingly. This information would have to be stacked and restored in the case of exceptions raised within exception handlers.

The requirement that main subprograms be parameterless procedures reflects established Ada practice and concern that Ada compilers should not be required to support as main subprograms functions returning values of type `Exit_Status`.

The procedure `Exit_Process` is provided to permit an application to terminate itself immediately. This procedure provides a capability that is otherwise not available in Ada—to terminate all the tasks of a partition abruptly. The definition effect of `Exit_Process` on tasks is intentionally not tied to Ada abortion or termination since those terms have established meanings that are irrelevant to this operation (which is performed by the underlying operating system rather than the Ada language implementation). The effect of this operation is not consistent with semantics of **`abort`**, *i.e.*, it does not recognize abort-deferral or perform finalization of controlled objects.

The underlying operating system can be relied upon to close any open file descriptors, but cannot be relied upon to flush any output buffers that might be in use by the application or the Ada implementation. There is no requirement that Ada output buffers be flushed; flushing all buffers was considered to be impractical since it would make the implementations of all output packages too dependent on the implementation of the POSIX/Ada interface, or *vice versa*. Also, flushing buffers is potentially too time-consuming for situations where a fast process exit is desired. Strictly Conforming POSIX.5 applications should close or flush all open files before exiting, using the operations provided for that purpose.

If the Ada language implementation creates any hidden processes to support execution of an Ada active partition, `Exit_Process` must terminate all such processes. The requirement to terminate hidden processes is a consequence of the rule that an Ada active partition execution must appear to be a single POSIX process.

B.3.4 Wait for Process Termination

The operations that request the status of a child process could have been specified as functions, returning a value of type `Termination_Status`. This alternative was rejected because the type of the result is private, and typically more than one component from the status value are required in subsequent programming so the value would have to be assigned to a variable anyway. Moreover, a procedure call is syntactically a statement and thus similar to other blocking Ada forms, such as the **`delay`** statement, the entry call, and the **`accept`**. Making the operation a procedure call warns the programmer more clearly about the side effects of this operation. The

out parameter (which cannot be omitted) is in the leading position so that the other parameters can be omitted, resulting in the default values being used.

The possibility of splitting these operations into separate procedures rather than using the parameters `Block` and `Trace_Stopped` was considered. The resulting increase in clarity of expression was not considered sufficient to outweigh the loss resulting from proliferation of procedures (from two to eight).

B.3.5 Package `POSIX_Unsafe_Process_Primitives`

It is clearly very difficult to define fully the semantics of the POSIX/C *fork()* and *exec()* functions in the presence of multiple Ada tasks in a way that is useful and easy to implement. As a consequence, an application that uses these operations directly will need to depend on special features of a particular Ada runtime system and a particular POSIX/Ada implementation and, therefore, is not likely to be portable. On the other hand, sometimes nothing less general than *fork()* or *exec()* will do. Therefore, the unsafe functions `Fork` and `Exec` (and `Exec_Search`) are provided, but their use is made more conspicuous by putting them into a separate package, labeled unsafe.

The text in 3.2 establishes conditions under which a Strictly Conforming POSIX.5 Application may use `Fork` and `Exec`.

B.3.6 Package `POSIX_Signals`

Signals are a specially troublesome feature of the POSIX interface because the behavior defined by POSIX.1 does not map directly to the Ada language. It is necessary to reserve some POSIX signals for the Ada-language implementation in order to preserve Ada rules concerning the predefined exceptions `Program_Error` and `Constraint_Error`. Also, no construct in Ada corresponds safely to the C-language asynchronous signal catching function (also called a signal handler), nor is one needed. Consequently, the POSIX/Ada binding restricts the view of C signal facilities accessible to Ada applications. All operations deemed useful to Ada applications are exposed, including the ability to take default signal actions and to wait synchronously for signals. Also, the POSIX/Ada signal interface is intended to be sufficient to achieve whatever level of signal interoperability is possible between Ada and C portions of a mixed-language application, depending upon the support provided by the underlying operating system.

B.3.7 Signal Model

The original signal model of POSIX.5 was based on the POSIX.1 single threaded signal model and the interrupt handling model of Ada 83. Since then, several things have changed:

— POSIX.1b added functions to synchronously wait for signals.

— POSIX.1c added multithreaded processes and has redefined the process-level signal masking operations to apply to the calling thread.

— Ada 95 has made the use of task entries as interrupt handlers obsolescent.

— It has become clear that signals are not interrupts and the POSIX/Ada binding should not treat them as such.

Due to these changes, much of the rationale for the treatment of signals in POSIX.5 no longer holds.

At the heart of the problem is how to treat *sigaction()*. The POSIX C interface allows an application to install C functions as asynchronous signal handlers via *sigaction()*. The C application can use *longjmp()* out of a signal handler to an earlier *setjmp()* point. This capability appears to be needed in some situations, where the desired response to a signal is to abandon a computation abruptly. Because the Ada application (or the Ada runtime system) is bound to need to execute some async-signal unsafe operations after such a *longjmp()*, it is not safe in general to allow the Ada application direct access to this capability. However, several Ada language constructs have closely related semantics, *i.e.*, propagation of an exception, **`abort`** of a task, and the asynchronous **`select`** statement. A central goal of this binding, therefore, has been to allow the Ada implementation to make use of signal handlers and *longjmp()* to implement these Ada operations.

This standard is based on the assumption that asynchronous signal handlers are used only by the Ada runtime system. Notification of synchronously generated signals to the Ada application is via Ada exceptions, raised by the runtime system. Notification of other signals is via task-level blocking operations modeled after the *sigwait()* interface of POSIX.1c and the *sigwaitinfo()* interface of POSIX.1b. The rationale for this decision is discussed further in B.3.13.

B.3.8 Signal Masking and Related Concepts

Originally, POSIX.5 was based on a model in which there is only one signal mask, whose effect is process wide. For the signal mask to be process wide was the existing practice for Ada-language implementations on POSIX.1 systems at that time, since the Ada language implementations provided their own user-level kernel for scheduling multiple threads of control within a process. Since then, some operating systems have begun to provide support for multiple threads of control within a process, and POSIX.1c defined a standard C-language interface for such services. The POSIX.1 multithreaded signal masking model requires that there be an individual signal mask for each thread of control. It defines some new operations for manipulating this mask and specifies much of the behavior of signal masking on a per-thread basis.

This standard cannot presume that the underlying operating system (if any) supports POSIX threads (see B.1.2). Therefore, the clear distinction between a thread-level signal mask and a process-level signal mask that is made in POSIX.1 is not possible for this POSIX/Ada binding. One fact is clear: This standard needed to be revised to allow for the possibility that each Ada task may have its own signal mask. Given the POSIX threads standard, it is likely that process-level signal masks will not be supported by all the operating systems over which this standard is implemented. Thus, it would be a significant burden to impose such a requirement on all implementations of this standard. Accordingly, the process-wide effect of the effects of the signal masking operations of POSIX.5 have now been defined to apply to the calling task (and possibly to other tasks as well).

Other changes to the signal masking model made by POSIX.1b and POSIX.1c include the addition of *sigwait()* and *sigwaitinfo()* operations, which allow a thread of control

to block until it *accepts* a signal. Since key operations introduced by POSIX.1c (which would need to be used by an Ada runtime system, even if they are not directly accessible to Ada applications) are not safe for execution from an asynchronous signal handler, it is clear that Ada applications should be encouraged to use a binding for the *sigwait()* and *sigwaitinfo()*.

While the effect of *sigwait()* is similar to a simple **accept** statement for a signal entry, (or, in general, to a selective wait statement with a set of **accept** alternatives), a direct mapping to Ada **accept** statements is not possible. The main reason is that the Ada **accept** statement allows complex effects, such as combining **accept** statements for normal task entries, and **delay** or **terminate** alternatives. Another reason is that *sigwait()* can be called by any task for any signal, whereas only one task may bind an entry to a given signal.

The solution chosen is to provide a direct binding for *sigwait()* and to allow an implementation optionally to support general signal entries in whatever fashion it chooses. In particular, the implementation is allowed to implement signal entries by providing a hidden thread of control that uses *sigwait()* to accept the signal first and then call the designated task entry via normal task-to-task entry call. Using a hidden signal-server thread to implement signal entries allows the user the convenience of mixing in **accept** statements for such entries without restrictions. However, the intent of the current wording is still to allow all the implementations that were allowed by the original POSIX.5 signal handling model (which was based on the spontaneous generation of a new virtual task for each signal occurrence). One change made by POSIX.5b is that use of task entries to accept signals is now controlled by the Signal Entries option.

While using *sigwait()* seems desirable, or even unavoidable, where POSIX.1 multithreading is supported, there are the following difficulties:

— When *sigwait()* is called, all the signals that the thread is ready to accept need to be masked, at least for the calling thread, or else the effect is undefined (see 3.3.8.2 of POSIX.1 {1}). Moreover, if any of the signals is unmasked in another thread, the effect is unspecified.

— After return from *sigwait()*, the action associated with each of the signals that the thread was prepared to accept is unspecified. In particular, if one wants to have the default action, one must call *sigaction()* to restore the default action. Still, unless the signal was masked before the call to *sigwait()*, there is a window of vulnerability in which arrival of a signal has an undefined effect.

Thus, in general, signals for which *sigwait()* is going to be called must be masked *in all threads*. The developers of POSIX.1c suggested that applications should routinely keep all signals masked in all threads, except for one thread per signal, that is allowed to unmask that signal. It does not seem wise to rely on applications to enforce this (or some more complicated) discipline, given the possible consequences. In a large application, no matter how well written originally, the probability is too high that routine maintenance might insert code that unmasks a signal for which another thread does a *sigwait()*. Such bugs are especially dangerous, since they are likely to remain latent for a long time. The effect will not show up very often, and when it does, it will be hard to diagnose.

For nonreserved signals (*i.e.*, those that a user might want to accept) only two cases of signal masking are of interest. Either a given signal is masked in *all* tasks, in which case one of the signal awaiting operations may be invoked for the signal, or the signal is unmasked in *at least one* task, in which case the default action is taken.

What is needed is some combination of known starting conditions for signal masks and known effects of signal masking and unmasking operations, so that an application can rely upon which of these cases holds. For this reason, this binding specifies the following:

— The environment task inherits its signal mask (for nonrealtime signals) from the parent process. This inheritance of the signal mask allows some influence over sensitivity to default actions by the creating process. The main subprogram can change this signal mask and thereby alter the signal actions of the process.

— All dependent tasks are initialized with all signals that are not reserved signals or signals bound to task entries masked. This allows dependent tasks to make free use of the signal awaiting operations for signals that are known to be masked in the environment task.

Some comments submitted during the ballot process for this standard made a convincing case that all the signals that can be accepted by an Ada application should be initially blocked, in the environment task as well as other tasks. Initial blocking of all signals would eliminate a window of vulnerability during which a new process that expects to receive and handle signals cannot protect itself against unwanted termination (which is the default action) on early arriving signals. The window is the interval between creation of the process and the first opportunity the process has to block the signal in the environment task.

For example, consider a library-level package that contains a task that uses some signal for notification of message arrival. Until the signal is blocked in the environment task, arrival of a message will terminate the entire process (the default action). If the task attempts to execute `Await_Signal`, the effect is undefined. The package body can prevent unwanted termination of the environment task by calling `Block_Signals` and then synchronizing (*e.g.*, by rendezvous) with the task that uses the signal. The task that uses the signal needs to wait for this rendezvous before it attempts to request any event notification via the signal or attempts to accept the signal. However, the logical complexity and runtime overhead of this synchronization could be eliminated if the signal were blocked from the start.

It is not desirable to initially block *all* the nonreserved signals in this fashion since one would then lose the ability to use the default signal actions for job control. Also, there is the issue of compatibility with POSIX.5. Both of these objections can be met, however, if only the real-time signals are required to be blocked initially in the environment task. Initial masking of the job control signals would remain under control of the parent process. Since real-time signals were new for POSIX.5b, there would be no existing POSIX.5 applications that use these signals.

If a requirement for initial blocking of the realtime signals is added, it will need to be stated in a form that does not impose any such requirement on non-Ada programs and that is compatible with existing operating system implementations. It follows that blocking of the signals must be allowed to be done by the new process after

creation. On the other hand, the standard needs to also be compatible with Ada-only embedded implementations, where it may be preferable to block the realtime signals from the first instant of the process creation. The blocking of signals cannot be any later than the elaboration of the body of package `POSIX_Signals` or any other package that depends on it since, after that point, the application may try to await a signal or use an operation that causes a signal to be generated for event notification. Given this dependence on the package `POSIX_Signals`, it may make sense that a requirement for initially blocking realtime signals might not apply to all Ada active partitions, but just to applications that use the package `POSIX_Signals`.

The suggestion of a requirement for initially blocking the realtime signals came late in the ballot process for POSIX.5b, after consensus had been achieved. There was some concern that it would require several ballot iterations to determine whether there was consent for such a change and to work out an acceptable formation of the details. Therefore, the issue was postponed until the next amendment or revision to this standard. However, informative notes were added to the standard to warn implementors and users that they should be prepared for the imposition of such a requirement in the future.

The signal masking and unmasking operations are required to guarantee the following: A task starts with the condition that the signal(s) of interest are masked in all other tasks. It unmasks the signal (this action might have the effect of unmasking the signal for other tasks as well). Later, the task again masks the signal. At this point one would want the signal to be masked again in all tasks. In order to guarantee this behavior, the implementation is allowed to impose a restriction that a task may not unblock a signal that is already unblocked by another task in the same process. The task that unblocks a signal becomes in effect the signal manager for that signal for the entire process.

The possibility was considered of putting additional restrictions on signal blocking, to further enforce the idea of each signal having only one manager task at a time. For example, a task might not be allowed to block a signal unless the same task had previously unblocked that signal. This restriction was not imposed due to problems dealing with details, such as the proper treatment of any signals that are already unblocked when the process starts.

NOTE: The next revision of this standard should perhaps review the possibility of defining further restrictions on signal blocking to enforce the concept of there being only one manager at a time for each signal.

The effect of the `Masked_Signals` parameter changes a little. The implementation is required to ensure that the requested signals are masked *for the calling task* for the duration of the operation, but then to restore signal masking to its prior state upon completion of the operation. Thus, the signals specified to be masked are guaranteed not to cause interruption of the operation, but may or may not be blocked from delivery to other tasks within the process.

POSIX.5 already had the operations `Block_Signals` and `Unblock_Signals`. These operations might have been deleted from POSIX.5b, since the original effect (of changing the process-wide signal mask) can no longer be required since POSIX.1c. However, deleting the signal blocking operations might break some existing applications, and would reduce the utility of `Ignore_Signal` and `Unignore_Signal` for

flushing pending signals. This standard has chosen instead to redefine these operations to apply to the calling task and to allow the POSIX/Ada implementation to restrict their use so that a task that unmasks a signal and then masks it again can be assured that it is possible to accept the signal via `Await_Signal`. Thus, the POSIX/Ada interface is implementable over a system that supports POSIX.1 multithreading. Restricting the effects of the singnal blocking operations to the calling task is also consistent with the following ballot objection from the P1003.4 Working Group (reproduced here verbatim), which maintains that process-wide signal masking cannot be used with predictable effects in a multithreaded process:

> In a parallel processing setting, with multiple threads or tasks, per-process signal blocking leads to inherent race conditions. In a threads context, suppose that one thread called a primitive to block all signal delivery to the process. After returning from this primitive, the thread might then reasonably expect that from then on no signal handlers would be run until signals are unblocked. But then suppose that a second thread was already one instruction into the execution of a signal handler when a context switch occurred, allowing the first thread to block signals. Then at some point, indefinitely far into the future, another context switch occurs which allows the second thread to continue running after the first thread had returned from the block signals primitive. Note that once this happens (A) a signal handler is run (in the second thread) (B) even while all signals are blocked. Thus, per-process signal masking can not be effectively used to deterministically block signals from being delivered. Per-thread or per-task signal masking does not suffer from this race condition. Thus, per-thread or per-task signal masking is a more appropriate signal masking model.

B.3.9 Tasking Safety

When a signal handler asynchronously preempts another thread of control within a process, the behavior of some of the operations defined by the C-language binding is unspecified if they are called from the signal handler. This restriction extends to the *longjmp()* operation and any other operations that may be performed by a thread of control after exit via *longjmp()* from a signal handler. Because signals may cause one Ada task to preempt another, directly or indirectly, and may cause asynchronous transfer of control (like *longjmp()*) within a task, operations that are not async-signal safe effectively cannot be used safely within an Ada program.

This standard requires that the interface to all POSIX operations enforce that they be *tasking safe*. The Ada runtime system is required to enforce tasking safety by exercising discipline in the way it uses async-unsafe operations and the way it does task context-switches. However, the application also must refrain from installing asynchronous signal handlers (which cannot be done using this standard, but might be done using interfaces not defined by this standard, including C-language code) that call Ada code that might call an async-unsafe operation. (See also B.2.3.3.)

B.3.10 Signal Type

After consideration of enumeration and private types, an integer type was chosen to represent signals. The primary reason for this decision was to facilitate exchange of signals between applications written in different languages and also to allow easy extension to new signal numbers. (See B.1.)

Declarations of both the long (Ada) names of signals and the conventional short (C) names are provided. The use of the long identifier within Ada programs is encouraged.

B.3.11 Standard Signals

POSIX.1 defines four classes of signals (see 3.3.3):

— Required signals: All implementations must support the required signals.

— Job control signals: Implementations that support the Job Control option must support the job control signals.

— Memory protection signal: Implementations that support the Memory Protection option must support the memory protection signal.

— Realtime signals: Implementations that support the Realtime Signals option must support the realtime signals.

Some of the standard POSIX signals are associated with events (*e.g.*, floating point error, illegal instruction) that in the Ada context require that predefined exceptions be raised. Thus, these signals must be reserved to the Ada language implementation. `Signal_Alarm` is almost certainly going to be used by some implementations to implement **`delay`** operations; so this signal is also reserved. An application is not permitted to interfere with the use of these reserved signals by the Ada language implementation or the POSIX/Ada implementation.

In addition, the signal `Signal_Abort` is reserved for the use of the implementation of Ada task abortion and asynchronous transfer of control.

The other standard signals are traditionally used in UNIX systems for interprocess coordination. One of the most useful aspects of these signals from the point of view of the application is the default actions that are taken when these signals are delivered.

The realtime signals are likely to prove the most useful for applications for several reasons:

— Realtime signals are required to be distinct from the other standard signals and do not have any uses defined by convention; thus, they may be used freely by the application to represent application-specific events (*e.g.*, notification of timer expiration and message arrival).

— Realtime signals may be queued (in FIFO order) so that multiple signal occurrences are not lost.

— Realtime signals may carry an application-defined data value, which may be used to distinguish between multiple occurrences of a signal.

— Notification for realtime signals is required to be in priority order (lowest signal number first) so that some measure of predictable behavior is possible. There is no notification order specified for other standard signals.

Constants are provided for values of the type `System.Address` corresponding to each of the nonreserved nonrealtime standard signals. These constants can be used in an address clause to bind a task entry to a particular signal. The values are given

as constants, rather than functions, because some Ada implementations might require that the address given in an address clause be a compile-time constant. No constants are provided for the reserved signals since an application is not allowed to accept these signals. Similarly, no constants are provided for `Signal_Kill` and `Signal_Stop` since these signals cannot be accepted, caught, or ignored. The function `Signal_Reference` returns an address for each realtime signal that may be used in a similar manner.

In contrast, signal *names* are defined for the reserved signals and for other signals that cannot be accepted, caught, or ignored because an application might need to send these signals to other (non-Ada) processes. The value `Signal_Null` is also provided, even though it cannot be sent or received, because this value has other uses. `Signal_Null` is a safe initial value for variables of type `Signal`, and may be used as a parameter to `Send_Signal` for checking the validity of process and process group identifiers.

B.3.12 Signal Sets

The type `Signal_Set` is a private type with operations to perform common set operations. This type might be implemented as an integer bit mask or as a `Boolean` array. Implicit initialization of objects of type `Signal_Set` is required in 3.3.7. Initialization can be implemented by making this a record type (*e.g.*, containing a vector) or an access type and using the default initialization provided by Ada for these types.

This standard is more restrictive in this matter than the C-language binding in two ways:

(1) It does not allow signals that are not defined in the standard to be members of the empty set.

(2) It does not allow uninitialized signal sets.

B.3.13 Examine and Change Signal Action

The use of *sigaction()* is reserved by this standard for the Ada runtime system. The main reason is safety. Asynchronous signal handlers have several unsafe aspects. First, the runtime system needs to control the actions and masking of reserved signals in order to implement certain runtime checks and tasking operations. Allowing the application to interfere would result in undefined behavior. The runtime system requirements do not rule out the possibility of an interface that allows application handlers for certain signals, *i.e.*, those signals that are not reserved for the runtime system. The main reason for not allowing application handlers for the latter signals is that such an Ada procedure might unknowingly call an runtime system routine that is not async-signal safe, again resulting in undefined behavior. The danger is greater with the advent of POSIX.1c, which introduced several more operations that are not async-signal safe. Short of narrowly specifying a particular task implementation, which would prevent implementors from taking advantage of the most efficient primitives of a particular underlying operating system, there is no way to predict exactly which Ada source constructs will result in calls to async-signal unsafe operations. POSIX.1c also added another strong reason for not allowing asynchronous signal handlers: The interactions with *sigwait()* are undefined, and this standard provides an Ada binding for *sigwait()*.

A less important reason for not allowing signal handler procedures is that Ada 83 does not have procedure pointers. While Ada 95 does introduce procedure pointers, this standard is intended to be compatible with Ada 83 as well as Ada 95, so procedure pointers are not used.

The capability of blocking or ignoring certain signals was almost omitted from POSIX.5 because it is difficult to reconcile with the Ada model of rendezvous. However, if there were no operations to block signals an application would haven no way of limiting the rate of arrival of troublesome signals. The consequences would be especially bad, since the default action for most signals is to terminate the process.

Because an application that is not interested in a signal may want to ignore it or restore the default signal action, but direct access to *sigaction()* is not desirable, the operations `Ignore_Signal` and `Restore_Default_Action` are provided.

For the `Ignore_Signal` operation, this standard requires dequeueing of any pending signals since it preserves an important invariant for programming with signals. Specifically, it should be true that a process can discard all the deliveries of a signal generated prior to a given point in time by ignoring the signal while the signal is not blocked.

The flag bit SA_NOCLDSTOP of the C-language binding provides a way to specify that the `Signal_Child` signal not be generated for a parent process whenever any of its children stops. Since the Ada runtime system hides the equivalent of the *sigaction()* function of the C-language binding, there is no direct way of translating this feature. The ability to interrogate and turn this bit on and off is, therefore, provided by means of separate operations, `Set_Stopped_Child_Signal` and `Stopped_Child_Signal_Enabled`.

B.3.14 Signal Queueing

The basic POSIX signal delivery mechanism is inherently unreliable for signal counting. It only guarantees that, following posting of a signal by some process, that signal will be delivered at least once some time after the posting. This mechanism has well-defined properties that are quite useful as long as the user does not believe all signal occurrences are delivered. Signals are not intended as event-counting mechanisms. Any application that attempts to use them in that way is not portable and is probably unreliable on most POSIX systems.

POSIX.1 does permit implementations to queue signals, but even if queueing is provided, it is not required to faithfully queue every signal. Hence, the portable use for (nonrealtime) signals is to tell a process to “do something at least once after the time when the signal is posted.” In order to permit efficient processing of these semantics by the receiving task, it is desirable to have a function for clearing the queue (*i.e.*, `Ignore_Signal`). By contrast, in light of the unreliable delivery semantics, there does not appear to be any benefit for an operation that would ignore a signal without clearing the queue.

POSIX.5b added a new class of signal, realtime signals, for which signal queueing may be enabled. The *sigaction()* interface is also used to enable and disable signal queueing for a given signal, and the application may need access to this capability,

but we do not want direct access to *sigaction()*. Therefore, the special operations `Enable_Queueing` and `Disable_Queueing` are provided.

B.3.15 Signal Notification Model

POSIX.1 defines two delivery modes for signals. The original standard only provided the signal number. POSIX.1b extended the notification mechanism for certain signals to provide the signal number, the source of the signal, and an application-specified data value queued with the signal by the sender. If the Realtime Signals option is supported, then realtime signals can be queued with data. This POSIX/Ada binding extends the notification mechanism to include signals with data only for the case of *sigwaitinfo()* and *sigtimedwait()*. In particular, the task entry mechanism is intentionally not extended to include notification of signals with data.

Two new types are introduced to support signal notification for signals with data. Type `Signal_Event` represents the information that must be supplied in order to generate signals with data. An object of type `Signal_Event` has the attributes:

— Signal: The signal number to generate.

— Notification: Whether notification is to occur.

— Data: The application-specified data value to enqueue along with the signal number.

Type `Signal_Info` represents the information obtained upon acceptance of a queued signal occurrence. An object of type `Signal_Info` has the following attributes:

— Signal: The signal number of the signal occurrence.

— Source: The source of the queued signal. Five values are defined by this standard, and the implementation may define other sources.

— Data: The application-specified data value enqueued when the signal was generated.

The type of the data value, `Signal_Data`, is intentionally a private type. Unchecked conversion is required with this type to make the user aware that the interface is inherently untyped. Admittedly, there would be no need for the user to do an explicit unchecked conversion if the package defined generic operations with the signal value type as a parameter, but the absence of a visible uncheckd conversion would be misleading. In reality, there would still be an unchecked conversion going on. There is no way in the underlying operating system to force the type of value passed by the sending program to be the same as the type of value expected by the receiving program. To ensure that the unchecked conversion can be done, a requirement is placed on the compiler to support unchecked conversion on certain data types, with predictable results.

In the C-language interface, the *sigev_value* component of a *sigevent* structure is large enough to hold a pointer or integer value. The Ada binding requires type `Signal_Data` to be large enough to hold an integer or address. It is not required to be large enough to hold an Ada access value because Ada access values may be larger than C pointers due to extra space that may be required to represent a constraint.

B.3.16 Examine Pending Signals

The ability to examine pending signals is included to help in deciding when to unblock a signal. Between POSIX.5 and POSIX.5b the semantics were loosened to take into account the possibility, introduced by POSIX.1c of there being two levels of pending signals (at the process level and at the task level).

B.3.17 Send a Signal

The *kill()* function of the C-language binding is unfolded by means of overloading into three distinct procedures. Since `Process_ID` is a private type, the C-language practice of using a negative value to denote that the ID should be interpreted as a process group ID could not be followed. The procedure for sending a signal to a process group thus needs to be split apart from the procedure for sending a signal to a process. Likewise, the C-language convention of using a null ID to indicate that the signal should be sent to all members of the process group of the sending process could not be followed. For this purpose, a version of `Send_Signal` with neither `Process` nor `Process_Group` parameter is provided. Sending a signal to `Null_Process_ID` or `Null_Process_Group_ID` is an error that must be detected by the POSIX/Ada implementation.

B.3.18 Binding for *sigsuspend*

The capability of the *sigsuspend()* function of the C-language binding is mostly provided by the `Await_Signal` operations and by the rendezvous operations if the Signal Entries option is supported. Therefore, there is no *sigsuspend()* function in this standard.

B.3.19 Synchronously Accept a Signal

The *sigwaitinfo()* and *sigtimedwait()* operations are not mapped to Ada tasking operations, such as accept statements, for two reasons:

— They may block the entire process unless POSIX.1 multithreading is supported.

— The interrupt entry mechanism is considered obsolescent for Ada 95; therefore, use of the task entry mechanism to receive signal notification is discouraged.

— The attempt by POSIX.5 to identify POSIX signals with hardware interrupts was misguided in any case, due to semantic differences.

POSIX.1b and POSIX.1c created potential confusion by providing a nonorthogonal set of operations for waiting for signals with overlapping semantics. These operations are *sigwait()*, which depends on the optional support for threads, and *sigwaitinfo()* and *sigtimedwait()*, which depend on the optional support for realtime signals.

The Ada bindings for *sigwaitinfo()* and *sigtimedwait()* have two versions each, one for the case where the *info* parameter is null and the other for the case where signal information is expected to be returned.

The version of `Await_Signal` that does not return information has the same parameter and result type profile as one would expect for an Ada binding of *sigwait()*. Moreover, the semantics seem to be indistinguishable; when *sigwaitinfo()* is called

with a null *info* argument, the effect apparently is equivalent to *sigwait*. Therefore `Await_Signal` has been chosen as the Ada binding for both operations.

After deliberation, it was decided that the form of `Await_Signal` that returns a value of type `Signal` should not depend on whether the underlying system supports POSIX.1 multithreading or on whether it supports the POSIX Realtime Signals option. An Ada equivalent of *sigwait()* is actually simpler to support than the mechanism for binding signals to task entries that was required by POSIX.5.

The form of `Await_Signal_Or_Timeout` that returns a value of type `Signal` also is required to be supported, even if the Realtime Signals option is not supported. One reason for this decision is orthogonality. Another reason is that the functionality is already present in the C-language interface by using a timer to generate an alarm signal, but a timeout cannot be achieved directly in this manner for Ada applications because `Signal_Alarm` is reserved for the Ada implementation. A third supporting argument is that *sigwait()* duplicates the functionality of a simple accept statement on a task entry bound to a signal and POSIX.5 provided timeouts on accept statements using the delay alternative on a selective wait.

Based on the assumption that the implementation of the Ada binding may choose to map these operations to a variety of underlying system services, the error codes for all variants of the `Await_Signal` operation are the union of the set of error codes that might be returned by *sigwait()* and *sigwaitinfo()*.

The signal-awaiting operations with queued information are split into a separate section because they depend on optional functionality.

B.3.20 Queue a Signal to a Process

The binding for *sigqueue()* is a rather direct mapping to the C-language interface. `Signal_Null` may be used to check the validity of the process ID.

B.3.21 Send a Signal to a Thread

No direct binding is provided for *pthread_kill()* for several reasons. First, one goal of this standard is that it be implementable on a variety of platforms, including those that may not support POSIX.1 multithreading, and this operation could not be supported on such systems. Second, it was decided not to support asynchronous signal handlers for reasons discussed in B.3.13. Therefore, a thread wanting to receive notification via a per-thread signal sent by *pthread_kill()* would need to accept it synchronously, using `Await_Signal`. However, the same effect could be achieved more naturally using a protected entry, or rendezvous. Third, the basic capability of asynchronously interrupting a thread is already provided with appropriate protection and cleanup mechanisms via the **`abort`** operation and the asynchronous transfer of control mechanism.

The one gap discovered in the existing Ada mechanisms is where a task is blocked on an interruptible system call and another task wishes to interrupt the call (raising `POSIX_Error` with error code `Interrupted_Operation` in the blocked task). For this purpose, `POSIX_Signals.Interrupt_Task` is provided.

B.3.22 Schedule Alarm

The capability of the *alarm()* function of the POSIX/C binding is provided by the Ada **delay** statement. Moreover, applications cannot be provided safe access to this functionality of the underlying operating system without risk of interfering with the operation of the Ada runtime system. Therefore, no *alarm()* function is in this standard.

B.3.23 Suspend Process Execution

No direct binding is provided for the *sigpause()* function. The capability to wait for a signal is provided by the signal-awaiting operations based on *sigwait()* and *sigwaitinfo()* and (optionally) the task signal-entry mechanism. The capability to wait for termination is provided in Ada when the main subprogram and all tasks not driven by signals complete in the Ada sense. Moreover, since the *pause()* function of the C-language binding never returns normally and this standard limits Ada applications to accepting signals synchronously, the only case in which *pause()* would return would be if a signal is caught by the Ada runtime system. This functionality does not appear to be very useful.

B.3.24 Delay Process Execution

The original rationale for providing no direct binding for the *sleep()* function of the C-language binding was that it is provided by the Ada **delay** statement, and before the multithreading extensions of POSIX.1c applications could not be provided safe access to this functionality of the underlying operating system without risk of interfering with the operation of the Ada runtime system. Even with multithreading the **delay** statement still provides an adequate indirect binding. In lieu of a binding to *nanosleep()*, this standard adds a requirement that the resolution of Ada delays be at least as fine as the resolution of the system-wide realtime clock, `Clock_Realtime`.

The addition of an Ada binding for *pthread_cond_timedwait()* further reduces the need for a binding to *sleep()* or *nanosleep()*.

B.3.25 Task Signal Entries

The use of task entries to accept signals was the only signal notification interface defined by POSIX.5. That interface was modeled after the interrupt entry interface of Ada 83, which has been labeled obsolescent by Ada 95. The signal entry interface is retained by this standard, for continued support of applications developed for POSIX.5, but it is made optional to reflect the fact that it is based on a faulty analogy between signals and interrupts.

A known gap in the signal entry interface definition is that it does not specify any requirement on the parameter profile of a task entry that is bound to a signal. No attempt has been made to specify a parameter profile because signal entries are considered obsolescent. The signal entry interface is not extended to receiving signals with information for the same reasons.

It is becoming generally accepted that POSIX signals are not the same as hardware interrupts and that the POSIX/Ada binding should not encourage such an identification. The Ada 95 model for hardware interrupt handling, based on protected

procedures, does not fit the POSIX.1c model of signal handling, in several respects. First, the scope of an Ada interrupt is considered to be global, rather than per-task, so interrupt handlers do not execute in the context of any specific task. In contrast, in the POSIX.1c model a signal can be targeted to a specific thread and the handler executes within the context of that thread. Likewise, the effect of Ada interrupt masking is global, at least among tasks executing on the processor where the interrupt is masked. In particular, the interrupt must be masked globally whenever a handler for that interrupt is executing. In contrast, in POSIX.1c, execution of a handler only masks the signal for the thread that the handler interrupts.

These differences reflect differences in intent. The Ada language interrupt handlers are intended to be low-level hardware interrupt handlers, unmediated by an operating system. POSIX signals, if they originate from hardware at all, are mediated by the operating system. This intent is supported by the proposal for a new POSIX interface for true hardware interrupts, for which the protected procedure model may be a better fit.

B.3.26 Composability Considerations

In general, integrating subsystems requires some discipline and coordination in development. POSIX signals introduce one point where coordination is required. Signals are a nonsharable resource. In general it is not possible to compose subsystems that use signals, unless it can be determined that they do not use the same signals. The difficulty of composing subsystems that use signals is independent of the language in whicn the subsystems are written. It is especially likely to be a problem where off-the-shelf subsystems (libraries) are linked with an Ada program. If the off-the-shelf subsystem is written in C, the problem is more likely, since there is no check that the C code does not use the signals reserved for the Ada language implementation (*e.g.*, SIGALRM, for timeouts). There is also a potential problem if a C-language subsystem calls an Ada-language subprogram, which in turn is interrupted by a C-language signal handler or calls a C-language subprogram. In this case, if the C-language subprogram calls *longjmp()*, storage may be lost, and Ada runtime system data structures may be left in a chaotic state.

B.4 Process Environment

B.4.1 Organization

The organization of Section 4 of this standard was changed from POSIX.1 to improve modularity. Specifically, the following changes were made:

— 4.1, 4.2, and 4.3 of POSIX.1 were combined into 4.1 of this standard to centralize handling of IDs related to processes.

— 4.4 of POSIX.1 was moved to package `POSIX`, in Section 2 of this standard, to centralize the system inquiry operations.

— 4.6 and 3.1.2.2 of POSIX.1 *argc*, *argv* were combined into 4.3 of this standard since together they define the strings that provide control of a command.

— 4.7.1 of POSIX.1 was moved to Section 7 of this standard to centralize controlling terminal functions.

— 4.7.2 of POSIX.1 was moved to Section 6 of this standard to centralize functions related to pathnames.

— Portions of 2.6 of POSIX.1 were moved into 4.3 of this standard to centralize environment handling.

— 5.2 of POSIX.1 was moved to Section 4 of this standard since the current working directory is another aspect of the environment of the process.

B.4.2 Package POSIX_Process_Identification

B.4.2.1 Process ID and Process Group ID

`Process_ID` and `Process_Group_ID` are separated in this standard into two types although cursory reading of POSIX.1 would suggest a single type was appropriate. If one carefully reads POSIX.1, it becomes clear that although process IDs and process group IDs share the same source of values, they are actually two distinct types with distinct sets of allowed and disallowed values. A valid process group ID may be an invalid process ID and *vice versa*.

Separating these types has the additional benefit of helping clarify the purpose and use of functions and procedures involving these types. For example, with two distinct types it is not possible to invert accidentally the two parameters to `Set_Process_Group_ID`.

`Null_Process_ID` is a constant of the `Process_ID` type, which is guaranteed not to match any process in the system. Eliminating this constant was considered, as it causes some problems when defining the action of operations that take a parameter of type `Process_ID`. However, as currently defined, `Fork` returns `Null_Process_ID` to signal that the process is the parent in the forking process, and `Get_Lock` returns `Null_Process_ID` when a lock is not owned by any process. Several routines that have input parameters of type `Process_ID` require special error checking to disallow operations with the parameter value `Null_Process_ID`.

B.4.2.2 Create Session

The name `Create_Session` is used instead of the alternative name `Set_Session_ID` since the chosen name is far more descriptive. The named procedure actually creates a new session (if successful) and does not allow for a process to set its own session ID without creating a new session.

B.4.2.3 Supplementary Groups

It is unspecified in this standard whether for `Get_Groups` the effective group ID of the calling process is included in the list of supplementary groups, as it is unspecified in posix1. While requiring inclusion or requiring exclusion (arbitrarily) was considered, the resulting complication of Ada implementations was not felt to be worth the minimal added benefit to a user.

The list of supplementary group IDs is not normally accessed by an application because the list serves no useful purpose. The effect of the supplementary group IDs on permitting access to files occurs within the system and outside the application program scope. An application cannot add to or delete from the system supplementary group ID list. A normal application (one without appropriate privileges) cannot change its effective group ID to anything except its real or effective group IDs.

B.4.3 Package Process_Times

B.4.3.1 Tick Count

`Tick_Count` was defined as a visible (integer) type so that all typical integer operations are immediately available. A private type would have required the definition of many functions for this type whose sole purpose would have been to provide integer operations.

The most critical issue for this type is providing sufficient resolution of time values while also providing sufficient range. POSIX.1 provides a simple type *clock_t*, which must be an integer type if it is to be used by an application. The accuracy and range of this type is intentionally left unspecified in POSIX.1, where it is noted that for typical clocks with only 60-100 ticks per second, the range is nearly a year. However, it fails to address systems with higher rates other than to require that the range be at least 24 h.

Since this standard is only a language binding for POSIX.1 it does not require systems to have any greater accuracy or range for POSIX time values than does POSIX.1, even though the limited range and coarse resolution of POSIX time values was a source of considerable concern to the Ada community, where high-resolution systems that run for years are frequently encountered. An implementation-defined integer allows an implementation of the POSIX/Ada binding to provide an accuracy and range that matches what the underlying operating system provides.

Using a record with components defining a fixed expression of time with extremely fine resolution and extremely wide range was considered and rejected. Such a type would suggest an accuracy and range that were not provided by any specific system. Unless one of the record components was a count of ticks, the type would likely not provide the accuracy provided by the system clock. Additionally, it would not permit the use of 64 bit integers, which provide for the required accuracy of counting ticks while still having sufficient range, even for systems with tick intervals under 100 ns.

B.4.3.2 Process Times

`Process_Times` is a private type with functions used to reference structure fields. Since the type is private additional data elements can be declared by an implementation and accessed via another (possibly child) package supplied with the implementation. Although POSIX.1 provides elapsed time as a function return and other times in a structure, the Ada implementation stores all times in a structure for future reference.

`Elapsed_Real_Time` is provided as a separate function because it is often required independently of other process times and because access to just-elapsed time may be significantly more efficient that getting all process times.

`Get_Process_Times` and, by extension, `Elapsed_Real_Time` are explicitly permitted to raise exceptions for implementation-defined conditions. This permission follows POSIX.1 precisely.

B.4.3.3 System Time

The C-language *time_t* type and the POSIX.1 *time()* function have been eliminated in this standard because they duplicate the standard Ada `Calendar.Time` type and the standard Ada `Calendar.Clock` function, respectively. Unfortunately, the Ada time does not fully conform to the functionality provided by POSIX.1 time, which provides monotonicity and provides for expressing and converting times using arbitrary time zones. Therefore, the package `POSIX_Calendar` has been added. (See B.4.5.)

B.4.4 Package `POSIX_Process_Environment`

The C main program parameters *argc* and *argv* were expressed as a `POSIX.POSIX_String_List` returned by `POSIX_Process_Environment.Argument_List`. Expressing these arguments in a string list provides a convenient way for programs to parse the argument string and is essentially identical to the facility defined for C. Although the first item in the argument list is conventionally the name of the command used to start the process, POSIX.1 does not specify or enforce that convention. Therefore, this standard also does not.

NOTE: As of Ada 95there is a standard Ada language-defined interface for argument lists in the package `Ada.Command_Line` defined in A.15 of the Ada RM {1}. This package covers most of the functionality of the argument-list operations of `POSIX_Process_Environment` and also contains a much-restricted form of the exit-status features of `POSIX_Process_Primitives`. The next revision of this standard should consider deprecating use of the POSIX-specific argument-list interface in favor of the standard Ada interface and consider integration of the POSIX exit-status features with those defined by the Ada language.

Functions to provide image and value for the private type `Environment` are not provided. While it is usual to provide these functions for most POSIX private types, the environment is a complex type that does not easily translate into the single strings required by these functions. A user can easily construct a printout of an environment for an application-specific format using an instance of the generic iterators.Similarly, a value function that constructs an arbitrary environment can easily be constructed from standard Ada loop and string input operations.

The character `'='` is forbidden in environment variable names for safety, despite a potential cost in performance; the implementation has to check each character in a variable name. Allowing `'='` in variable names could cause problems when building an Ada binding on top of a POSIX system implemented in C, in which the (name, value) pair is represented as a single string with `'='` marking the start of the value.

This standard leaves the processing of multiple instances of environment variables as undefined. The alternative of specifying behavior for these (erroneous) environments would be of benefit in only rare cases. Where efficiency or other concerns do not dictate otherwise, it is suggested that the handling be as follows:

— Delete operations should ensure that all copies of a name are deleted.

— Set and access operations should reliably access the same (name, value) pair.

— Iteration operations should iterate over all copies of a name, with the first name in a set of identical names being the same name retrieved by access functions.

The semantics of set and delete operations for environment variables were defined to be similar to formal sets, in which adding or subtracting an item ignores the previous state of the set. It was felt that these semantics were appropriate for typical uses of environment variables.

Adding an additional parameter to the operations that would select alternative semantics wherein exceptions are raised for setting an existing name or deleting a nonexistent name was rejected. That approach was viewed as complicating the interface with little benefit to the user. If an application truly requires these alternate semantics, the user can test existence of an environment variable using `Is_Environment_Variable` prior to the delete or set operation.

The effects of one task modifying the current environment while another iterates over a value obtained from `Copy_From_Current_Environment` are undefined in order to allow for efficient implementations.

Representing the type `Environment` presents some awkward choices. Suppose that for compatibility with an underlying C implementation it is desirable to represent an environment as an array of pointers, pointing to strings, where each string represents a (name, value) pair separated by `'='`.

Two ways to allocate space for the strings are

(1) Heap Allocation. The strings might be dynamically allocated in heap space, as environment values are set and unset. Heap allocation imposes no special limit on the size of an environment, but poses problems for storage recovery since the storage of such strings cannot be automatically recovered safely.

(2) Local Allocation. The strings might be allocated within a single contiguous block of storage associated with each object of type `Environment`. Local allocation would impose a limit on the size of an environment (as is permitted in POSIX.1), but would permit automatic storage recovery.

In neither case can assignment between objects of type `Environment` be done via simple copying. For this reason, the type is made limited private. Furthermore, to permit an implementation to handle the current process environment differently than any user-defined environment, this standard provides the current environment as an implicit parameter to each overloaded operation.

In order to permit an implementation to use heap allocation, the user is held responsible for providing an opportunity to recover storage by calling `Clear_Environment` whenever an object of type `Environment` is no longer needed.

Examples of code fragments using operations in `POSIX_Process_Environment` follow:

```
declare
   Env: POSIX_Process_Environment.Environment;
begin
   POSIX_Process_Environment.Copy_From_Current_Environment(Env);
   declare
      FOO: constant POSIX_String:=
        POSIX_Process_Environment.Environment_Value_Of("Foo",Env);
        -- FOO is set null (by default) if it is not in Env
```

```
   begin
      -- Do processing on FOO
      POSIX_Process_Environment.Clear_Environment(Env);
   end;
end;
```

If it is important to distinguish a null **FOO** from **FOO** not being defined in Env and there is no appropriate string to use that is distinct from possible settings of **FOO**, then Is_Environment_Variable can be used as illustrated by the example below. The example eliminates the need for a local copy of the current environment and the overhead of allocating and clearing a potentially large environment.

```
if POSIX_Process_Environment.Is_Environment_Variable("Foo") then
   declare
      foo: constant POSIX_String:=
        POSIX_Process_Environment.Environment_Value_Of("Foo");
   begin
      -- Processing on foo, including a null foo
   end;
else
   -- Processing for a nonexistent foo
end if;
```

B.4.5 Package POSIX_Calendar

This package provides functionality similar to the standard Ada package Calendar. However, dates in this package are interpreted using the time-zone information in the **TZ** environment variable, defined in 2.11.1. Time values obtained from the package Calendar need not have sufficient information to permit their interpretation using the **TZ** environment variable. AI-00195 {B8} requires that the function Calendar.Clock "returns a value that reflects the time of day in the external environment." It has been argued that AI-00195 explicitly does not permit values returned by Calendar.Clock to be modified by the **TZ** environment variable. Furthermore, current implementations of Calendar.Time do not contain sufficient information to permit values of this type to be adjusted using **TZ**. Therefore, this standard defines a new POSIX_Calendar package with the same functionality as the Ada predefined Calendar, but which supports adjustment via the **TZ** environment variable. This package also includes routines to convert between values of type Calendar.Time and POSIX_Calendar.Time. A conversion from Calendar.Time to POSIX_Calendar.Time may require interpretation of the value using the current time zone. In this case, the operation POSIX_Calendar.To_POSIX_Time is equivalent to the following code:

```
-- Given the Calendar.Time value Date, convert it to POSIX˙Calendar.Time:
declare
   Year:   Calendar.Year_Number;
   Month:  Calendar.Month_Number
   Day:    Calendar.Day_Number;
   Secs:   Calendar.Day_Duration;
   Answer: POSIX_Calendar.Time;
begin
   Calendar.Split (Date, Year, Month, Day, Secs);
   Answer := POSIX_Calendar.Time_Of (Year, Month, Day,Secs);
end;
```

B.4.6 Package POSIX_Configurable_System_Limits

This package provides functions to retrieve options and implementation limits. Duplicate names of some of the functions are included, as explained in B.2.3.5. New names are included to conform to the naming rules for options and limits established by POSIX.5b. Old names are retained for compatibility with POSIX.5.

B.5 Files and Directories

These packages are intended to provide the functionality of Section 5 of POSIX.1. Differences in organization from POSIX.1 are noted below.

B.5.1 Organization

In general, Section 5 of this standard contains the types and operations that pertain to either directories or the descriptions (attributes) of files, rather than the contents of files. Section 6 primarily pertains to manipulation of the contents of files. In terms of the data types manipulated, Section 5 deals with files according to pathnames, rather than with `File_Descriptor`s. Functionally, Section 5 operations deal with files that are not open to the program, while Section 6 deals with open files.

The transformation from a pathname to an open file descriptor is made by `Open` and `Open_Or_Create`. These operations were placed in Section 6 since they are expected to be associated with `Close`, `Read`, `Write`, and `Seek`. The only exceptions to this pattern are the functions that extract attributes of the `Status` value of a file and the various functions in the package `POSIX_Configurable_File_Limits` that are overloaded to accept either a pathname of a `File_Descriptor`. Both of these functions belong in Section 5 because they are queries on the description of a file, rather than on its contents.

`Change_Working_Directory` and `Get_Working_Directory`, corresponding to *chdir()* and *getcwd()* in the C binding, were moved from their original location in POSIX.1 to 4.3 of this standard, as the current working directory is an aspect of the environment of a process.

B.5.2 Package POSIX_Permissions

B.5.2.1 Permission Sets

`Permission_Set` is defined as a `Boolean` array in order to allow the use of the logical operators: **`and`**, **`or`**, and **`not`**, in the construction of permission sets. Constants of type `Permission_Set` are defined corresponding to owner, group, and others access. A constant is also provided for the union of owner, group, and others access. Finally, constants are provided for setting the group ID and the user ID.

The use of a visible array type for `Permission` deviates from the convention followed nearly everywhere else in this standard; it is a relic dating back to the earliest work on POSIX.5. Extensions to this standard are encouraged not to emulate this case. Instead, they should follow the predominant convention, which is to use a private type in this sort of situation. For example, see the type `Option_Set`, which is used in many similar situations.

The constant approach allows implementation extension packages to add elements to the sets that are not necessarily contiguous. Thus, conforming applications can be written that port to extended implementations with reasonably predictable results. For example, consider an extended package, `My_Permissions`, that, in addition to the user, group, and others permissions, adds in `My_Special_User_Permissions`, `My_Special_Group_Permissions`, and `My_Special_Other_Permissions`. A program to remove all others permissions could use the expression:

```
all_but_other := arbitrary_set & (not Others_Permissions_Set);
```

B.5.2.2 Process Permission Set

The *umask()* function in the C binding establishes the file mode creation mask of the process to the value of its parameter and returns the prior value of the mask as its result value. In Ada it is considered poor style for a function to have side effects like this. Thus, the corresponding operation `Set_Allowed_Process_Permissions` is not a function, but a pair of overloaded procedures where one returns the old value and the other does not. `Set_Allowed_Process_Permissions` is overloaded with a version that returns the old permissions as an **out** parameter. Like other operations, this operation is atomic in the face of tasking, but the user is still responsible for coordinating the use of this operation by multiple tasks.

Future extensions to this standard should attempt to abide by this stylistic rule against using functions for side effects, even though it is apparently violated a few places in this standard. In each of those cases there was some specific overriding rationale. For example, the `Open` operation on files is a function. It might be argued that such an operation is not called primarily for its side effect, even though it will have a side effect of creating a new open file description. The primary effect may be viewed as to return a handle that can be used for I/O on the specified file. An analogy can be drawn to the Ada **new** operation, which has the side effect of allocating and initializing storage, but which is still syntactically an expression. This argument was not universally supported at the time it was originally made, and it does not address the contrary precedent set by the use of a procedure for the open operation in the standard Ada I/O packages. In the drafting of POSIX.5b, there was opposition to continuing this precedent, for the open operations on new kinds of objects, such message queues. This opposition was overcome by the argument that it would unnecessarily surprise users of the standard if operations that are otherwise so similar were to sometimes have the form of a procedure and sometimes have the form of a function. Therefore, all such operations are functions, for consistency with the look and feel established by `Open` in POSIX.1.

Also, in the C binding, *umask()* is the only way to determine the current setting of the file mode creation mask. Being a destructive operation that requires a second call to restore the value, this operation is not safe in the face of Ada tasking. The function `Allowed_Process_Permissions`, which interrogates the current setting, has been added to avoid these problems and to simplify the application program.

B.5.3 Package `POSIX_Files`

Most of the file operations are defined as a straightforward mapping from the C binding into Ada. The primary difference is that instead of being functions who

return codes that must be interpreted, most of the operations are procedures that can raise exceptions.

The directory iterator generic procedure is included in the package POSIX_Files because the other operations specific to directories (which are a subtype of files) are in this package. The structure of a directory is private and is only revealed through use of the iterator, which provides each directory entry sequentially. An example of iterating through a directory and printing each file name and its size follows:

```
package Example1 is
   procedure Print_Sizes_Of_PWD;
end Example1;
with POSIX,
     POSIX_IO,
     POSIX_Files,
     POSIX_File_Status,
     POSIX_Process_Environment;
package body Example1 is
   procedure Print_Dir_Entry
      (D_E:  in POSIX_Files.Directory_Entry;
       Quit: in out Boolean) is
       Size: POSIX.IO_Count;
   begin
      Size:= POSIX_File_Status.Size_Of
       (POSIX_File_Status.Get_File_Status
          (POSIX_Files.Filename_Of (D_E)));
     -- Print pathname, size
   end Print_Dir_Entry;
   procedure Print_Sizes is new
      POSIX_Files.For_Every_Directory_Entry (Print_Dir_Entry);
   procedure Print_Sizes_Of_PWD is
   begin
      Print_Sizes (POSIX_Process_Environment.Get_Working_Directory);
   end Print_Sizes_Of_PWD;
endExample1;
```

B.5.4 Package POSIX_File_Status

The operations in this package are a straightforward mapping from the C operations to retrieve attributes of a file object based on its pathname or file descriptor. First, an object of the private type Status is retrieved. Then each attribute can be retrieved from the Status object. New functions are provided for the file-like objects added by POSIX.5b: shared memory objects, message queues, and named semaphores.

B.5.5 Package POSIX_Configurable_File_Limits

The POSIX.5 package POSIX_Configurable_File_Limits has been made dependent on POSIX_IO in order to support the POSIX.1 *fpathconf()* capability, which requires an argument of type POSIX_IO.File_Descriptor.

For each of the different limits or options associated with a file or directory, two overloaded pairs of functions are provided. One pair of functions takes an open file descriptor as a parameter. The other pair of functions takes a pathname as a parameter. The first function of each pair determines whether a specific limit exists, and

the second acquires the value of the limit if a limit exists. Using a separate function name for each limit or option (rather than a single operation like *pathconf()*) provides the advantage of allowing static type checking on the return type based on the limit or option being acquired.

An alternative design approach considered was to have a single pair of subprograms that return a variant record whose discriminant is the specific limit or option acquired. This method is used in POSIX.1, except that POSIX.1 does not have an explicit discriminant. Using a single record type does allow arbitrary access to limits and options with the same subprogram signature, but eliminates static type checking on the correspondence between the limit or option and the returned value. If there is no explicit discriminant, then there is no type checking available at all. Moreover, implementation extensions might require modification to the record type definition, in the standard package specification, or they would have to provide duplicate operations on a different variant record type.

An alternative approach was also considered for determining whether the parameter exists, specifically, the approach taken by POSIX.1. It uses a single return parameter with a reserved value that is used to signal that there is no corresponding limit. Such values may be either negative one (`-1`), as in C, or a very large value.

The chosen method is a combination of a `Boolean` return value and the use of a reserved value. If a particular application needs to distinguish the difference between a very large value and the absolute lack of a limit, two calls may be required. However, it is expected that the distinction is typically unnecessary, and only a call to acquire the limit will be used.

The special character originally in this package, `Disable_Terminal_Special_Characters`, was moved to Section 7 of this standard because it is applicable only to terminal devices.

Again, there are duplicate names of some of the functions in package `POSIX_Configurable_File_Limits`. New names are added to conform to the naming rules established by POSIX.5b and old names are retained to be compatible with POSIX.5.

B.5.6 Directory Operations

The first possibility considered as an Ada binding for these functions was a separate package, `POSIX_Directories`. This package contained the operations `Open`, `Close`, `For_Every_Directory_Entry`, `Make_Directory`, `Remove_Directory`, and `Rename`. Since directories are only a special case of files, the operations `Make_Directory`, `Remove_Directory`, and `Rename` were moved to `POSIX_Files`. The `Rename` operation was the same as the `POSIX_Files.Rename` operation.

Because of the use of a generic procedure to implement the directory iteration operation (as opposed to other techniques described in the global rationale), the operations `Open` and `Close` became unnecessary (their function would be included in the actions taken by `For_Every_Directory_Entry`). Elimination of the `Open` and `Close` on directories left the package `POSIX_Directories` with the single operation `For_Every_Directory_Entry`, so the remaining operation was moved into `POSIX_Files`.

B.6 Input and Output Primitives

The operations in Section 6 of this standard apply in general to the contents of files as opposed to file attributes. The synchronous I/O interface is provided in package `POSIX_IO` and is modeled on the Ada I/O operations. The AIO interface added by POSIX.5b is provided in package `POSIX_Asynchronous_IO`. The interface to provide advisory reader and writer locks on files is provided in package `POSIX_File_Locking`.

One significant change wrought by POSIX.5b is that the Ada 95 type `Stream_Element_Array` is used for the data transferred to and retrieved from files (and other file-like objects).

B.6.1 Package `POSIX_IO`

The model of a file is that it is a stream of bytes, typically on external storage. Bytes to be transferred to and from a file are specified in terms of an offset and a count of bytes. The offset is specified by the procedure `Seek`. The number of bytes to transfer is specified as a parameter to the operation. Normally, the number of bytes actually transferred is available as a return parameter from the operation. It is not assumed that a byte is necessarily eight bits or that the transfer of one byte of data means that only eight bits are transferred. Additional parity bits or other nondata bits may be transferred.

In POSIX.5, the type `POSIX_Character` was used as a measurement unit for data in the description of low-level operations on files. Some people found this use of a character type for these operations on files confusing, interpreting it as a requirement or permission for these operations to perform some sort of character set conversion. This was never intended, so this standard now uses the term *byte* wherever it needs to specify the length of a file, an offset in a file, or an amount of data transferred. Likewise, uses of the type `POSIX_String` for buffers of low-level IO operations in POSIX.5 are being phased out in favor of the type `Ada_Streams.Stream_Element_Array`, which is defined in Ada 95. For compatibility with POSIX.5, the procedures `Read` and `Write` using buffers of type `POSIX_String` are retained, while overloaded `Read` and `Write` procedures using buffers of type `Stream_Element_Array` are added for use by new applications. The I/O operations on `POSIX_String`s are obsolescent and should be avoided in new applications.

The use of `Stream_Element_Array` as the buffer type of these operations allows applications to define Ada stream objects using POSIX I/O operations without the need for time consuming item-by-item unchecked conversion. Together with the marshalling and umarshalling operations (`'Read` and `'Write`) defined in [Ada 95 13.13], stream objects provide a reliable way of doing I/O on complex data objects, such as records, arrays, and even tagged objects.

A file may be opened with operations such as `Open`, `Open_Or_Create`, or `Create_Pipe`. After being opened, there is an association between an external file, a device, or an object and a `File_Descriptor`. All subsequent operations use the file descriptor.

A file descriptor is logically a handle for an open file description. (See B.6.1.1.) Whether `File_Descriptor` should be a visible integer type or a private type was

discussed. The primary reason for making it a visible integer type was to allow an application to have an array indexed by `File_Descriptor`. The main inconveniences are that it is poor Ada style and it allows meaningless operations (*e.g.*, + and *) on the file descriptors.

Each open file description for a disk file has associated with it a current read-write pointer, which denotes the position in the file at which the next read or write operation will occur. The current position is a number in the range of subtype `IO_Count` and is the number of bytes from the beginning of the file. The bytes are numbered from `0..length-1`, following the standard POSIX practice.

A POSIX convention is that processes normally start with

— File descriptor 0 open to read standard input `POSIX_IO.Standard_Input` from a terminal or other data source.

— File descriptor 1 open to write standard output `POSIX_IO.Standard_Output` to a terminal or other data destination.

— File descriptor 2 open to write standard error `POSIX_IO.Standard_Error` to a terminal or other data destination.

By convention, many programs get their input from `POSIX_IO.Standard_Input`, put their results on `POSIX_IO.Standard_Output` and write unusual error messages on `POSIX_IO.Standard_Error`.

B.6.1.1 File Descriptor, Open File Description, and External File

When a file is opened with `Open` or one of the other subprograms, three different objects become related to one another:

— A `File_Descriptor` is a value that is returned by `Open` and used as a handle by `Read`, `Write`, `Close`, and other subprograms in the Ada binding.

— An open file description is an object (not directly visible to the application) that is traditionally managed inside operating system kernel memory, although it might be implemented differently.

— An external file is the actual file and its contents on a storage medium such as a disk.

It is important to understand the difference between these three objects. In the simplest case, after a successful `Open`, there is one file descriptor, one file description, and one external file. Writing a byte using the file descriptor as handle sends a byte to the system, advances the read/write pointer by 1 in the file description, and eventually writes the byte to disk. In slightly more complicated examples, the same open file description may be referenced by more than one `File_Descriptor`, and the same external file may be referenced by more than one open file description.

The case of more than one `File_Descriptor` referencing the same file description occurs after `Duplicate`, or `Exec` when the file is not closed upon exec. When two `File_Descriptors` reference the same file description, they share the same read/write pointer, mode (permissions), and flags (options and access modes). For example, `Read` or `Seek` changes the read/write pointer for both file descriptors. Reading

a byte at position 5 on descriptor 1, then seeking to position 10 on descriptor 2, then reading another byte on descriptor 1 result in reading the tenth byte.

This standard also allows two open file descriptions to reference the same external file. The case of more than one open file description referencing the same external file occurs when two separate `Open` calls are made, whether these two calls are made by the same process or by different processes. If two open file descriptions reference the same external file and then one is removed (*i.e.*, deleted) with `POSIX_Files.Unlink`, the other still remains open, and the file continues to exist until the last file description is closed. At that time the external file really is removed from the system. Concurrent I/O operations on a single file made via different open file descriptions will be interleaved by the system. Interleaving of I/O operations one the same file is desirable if both open file descriptions are being used to read the file or both were opened with the `Append` option and are being use to write to the file. Interleaving may also be desirable if read and write locks are used. Otherwise, interleaved I/O on a single file is likely to be dangerous, but this standard does nothing to prevent it.

File access modes (*i.e.*, `Read_Only`, `Read_Write`, `Write_Only`) and file options (*e.g.*, `Append`, `Non_Blocking`, `Truncate`, `Exclusive`) are properties of the open file description. They apply to all file descriptors referring to the open file description. Some of these options (*e.g.*, `Non_Blocking`, `Append`) can be read and/or set by the `..._File_Control` operations. When multiple file descriptors refer to the same open file description and an option (*e.g.*, `Append`) is changed using one file descriptor, the change applies to I/O operations on all file descriptors referring to that open file description.

Conversely, the same process can open a file twice and obtain two different file descriptors referring to two distinct open file descriptions. The file can be open with the `Non_Blocking` option specified for one file description and not specified for the other. Then blocking operations that could not complete immediately would return error code `Resource_Temporarily_Unavailable` on one file descriptor, but block the caller on the other file descriptor.

Unlike the access modes and file options, the flag that specifies that a file should be closed on exec is a property of the file descriptor, and not the open file description. Thus two file descriptors can refer to the same open file description, with one file descriptor having the close-on-exec property. When an exec operation occurs, one file descriptor will be closed, but the other will not and the open file description will remain intact.

B.6.1.2 Open and Close

Some combinations of options to `Open` and `Open_Or_Create` probably have no logical sense; for example, `Read_Only` in combination with `Append` or `Truncate`. The interface does not check these combinations; the behavior is implementation defined. (POSIX.1 states that the behavior is implementation defined if *open()* is called with the equivalent combination of flags.)

The file modes `Read_Only`, `Write_Only`, and `Read_Write` are provided as a specific parameter to `Open` and `Open_Or_Create` because a file mode must be specified when a file is open. The other file options are passed as a value of type `Open_Option_Set`, which has a default value indicating no options. The `Open_Option_Set` type is

derived from `POSIX.Option_Set` which is a private type, and can be implemented as a bit mask. Some file open options are defined by this standard. Implementations may define still more options. Using these options should be straightforward, using expressions such as the following:

```
Options => Append + No_Controlling_Terminal + Implementation_Specific_Option
```

The C-language operation *open()* is separated into two functions: `Open` and `Open_Or_Create`, instead of using a flag (*i.e.*, *O_CREAT*) to specify file creation. On the other hand, the related flag *O_EXCL* is retained as the option `Exclusive`.

`Close` corresponds to the C-language operation *close()*. One design issue was whether the `File` parameter of `Close` should be an Ada **`in`** or **`in out`** parameter. The argument for using **`in out`** is that a file descriptor can be set to a conventional value (*e.g.*, `-1`) upon return from `Close`. The inconvenience is that `Close(Standard_Output)` become invalid, since `Standard_Output` is a constant. The mode **`in`** was chosen because it was considered too inconvenient to require a programmer to use a temporary variable for `Close(Standard_Output)`. (Incidentally, this precedent set by POSIX.5 is *not* followed for some close operations added by POSIX.5b, such as closing a named semaphore or closing a message queue; but these descriptors are a private type distinct from type `File_Descriptor`.)

If a file is opened with `Open`, `Open_Or_Create`, or `Duplicate` and then is deleted with `POSIX_Files.Unlink`, the contents of the file still exist until the file is closed by all processes that have it opened. I/O operations and file locks work normally as long as an open file description exists for the file.

The `Duplicate` and `Duplicate_And_Close` functions correspond to the C-language operations *dup()*, *dup2()*, and *fcntl()* with the *F_DUPFD()* flag. `Duplicate` with `Bound` equal to zero is the same as *dup()*, and with `Bound` greater than zero it is the same as *dup2()*. `Duplicate_And_Close` is the same as *fcntl()* F_DUPFD).

`Create_Pipe` is identical to the C operation *pipe()*, except the parameters are declared in a fashion more appropriate to Ada.

B.6.1.3 Read and Write

`Read` and `Write` use the actual length of the `Buffer` parameter as the I/O request length and use an **`out`** parameter `Last` (rather than an **`out`** parameter `Length`) to return the count of bytes actually transferred. This parameter profile mirrors that of `Text_IO.Get_Line`. Generic procedures `Generic_Read` and `Generic_Write` are provided as a user convenience. An instantiation on an unconstrained type may be rejected by the implementation, as per the Ada rules on generics. The model is that bytes are written, and when the file is read back, bytes equal to the size of `Item` are read back and put into `Item` via `Unchecked_Conversion`. No type checking or reconstruction of unconstrained types occurs, and no array descriptors or record structure information is written into the external file.

The error behavior of `Generic_Read` and `Generic_Write` is, therefore, closer to POSIX.1 *fread()* and *fwrite()*, respectively, than to POSIX.1 *read()* and *write()*.

The fact that `Read` can raise `End_Error` is different from POSIX.1 semantics. In POSIX.1 *read()* will just return zero and *errno* will not be set. This standard reflects

the behavior of the language-defined Ada I/O packages in the face of an end-of-file condition.

B.6.1.4 Seek

In early versions of the UNIX system, the C operation *lseek()* was called *seek()*. As file sizes grew beyond 64 K bytes, the parameter changed from the C-language type *int* to type *long*, and *seek()* was renamed *lseek()*. Here it is renamed back to `Seek`, which accurately describes the operation. The C-language operation *lseek()* is split into three subprograms for convenience: `Seek`, `File_Position`, and `File_Size`. `File_Size` might be implemented with several calls to *lseek()* or one call to *fstat()*.

B.6.1.5 File Control

The C *fcntl()* operations for getting and setting modes and flags become more rational when separated into several functions. In other words, the Ada caller does not have to worry about separating bits in the result that is returned from *fcntl()*. The file record locking operations from C *fcntl()* are implemented by the separate file locking facilities in package `POSIX_File_Locking`. The F_DUPFD operations are supported by `Duplicate_And_Close`. The current design of `Open_Option_Set` will support use of the file control operations to set and query file options defined by future POSIX standards. The operation `Get_File_Control` is a procedure with two **`out`** parameters to keep the two types `File_Mode` and `Open_Option_Set` separate.

B.6.1.6 Update File Status Information

Although *fchmod()* is in Section 5 of POSIX.1 this binding puts the operation into `POSIX_IO` (see 6.1) to preserve the invariant that only the operations that operate on files by name are placed in `POSIX_Files` and the operations that operate on files via file descriptors are in Section 6. To do otherwise would also have created a dependence of `POSIX_Files` on `POSIX_IO`. These factors were felt to outweigh the functional affinity with *chmod()*.

B.6.1.7 Truncate a File to A Specified Length

Although *ftruncate()* is in Section 5 of POSIX.1, this binding puts the operation into `POSIX_IO` (see 6.1)d, to preserve the invariant that only the operations that operate on files by name are placed in `POSIX_Files` and the operations that operate on files via file descriptors are in Section 6. It would also have created a dependence of `POSIX_Files` on `POSIX_IO`.

B.6.1.8 File Synchronization

The interfaces for file synchronization are a very direct mapping to the corresponding interfaces defined in POSIX.1.

B.6.2 Package `POSIX_File_Locking`

B.6.2.1 File Locking

POSIX file record locking is advisory. A lock held by one process keeps another process from setting a conflicting lock. But a lock does not prevent other operations,

such as open, read, or write.

Locks may be read locks or write locks. They may cover a whole file or a portion of a file. A lock may start or extend beyond the current end of a file, but may not extend before the beginning of a file.

The operations of package `POSIX_File_Locking` in the Ada binding split out the behavior of the C function *fcntl()* having to do with file locking. The use of three separate subprograms is believed to be clearer than the corresponding multipurpose C function.

This interface does not follow the convention of using private types to hide C structured types, which is followed elsewhere in this standard, as it makes use of a visible discriminated record type `File_Lock`. This visible record has a discriminant that is used to define two different variants, corresponding to whether the lock covers the entire file or only a portion of the file. The rationale for this deviation from the general practice of this standard (which is to avoid visible record types) dates back to POSIX.5, was not documented, and has been forgotten.

In C, a file lock of length zero is used to mean the whole file. In this standard, a variant record is used instead. The C-language record has the process ID as a field, but since this field is only used in `Get_Lock`, it has been made into an **`out`** parameter.

Implementors of `Get_Lock` should make a local copy of the `Lock` parameter before using it to guard against aliasing problems if the caller passes the same actual parameter for the `Lock` and `Result` parameters.

B.6.3 Package `POSIX_Asynchronous_IO`

B.6.3.1 IO Control Blocks

The use of control blocks and buffers is inherently unsafe since there is a persistent reference to the control block for the duration of the IO operation. IN other words, if the control block is deallocated and later reallocated for another use the behavior of the operation is undefined. The same problem occurs with the data buffers that are used for AIO. If the I/O operation is a write, a dangling reference may result in writing out garbage. If the I/O operation is a read, the dangling reference may result in overwriting arbitrary memory, and that generally leads to chaotic failure of the program.

In order to minimize these dangers this binding uses a private type `AIO_Descriptor` that is a handle or reference to an AIO control block, which is not visible. Creation of the reference and allocation of the control block are performed using `Create_AIO_Control_Block`. Destruction of the reference and deallocation of the control block are performed using `Destroy_AIO_Control_Block`. The implementation of `Destroy_AIO_Control_Block` may check, and refuse to deallocate the control block if it not safe. Whether to perform this check is up to the implementation. If the implementation detects that the reference is to a control block of an I/O operation in progress or does not correspond to any control block, `Constraint_Error` is raised.

`AIO_Descriptor` was made a private type instead of a limited private type to allow values of type `AIO_Descriptor` to be moved into and out of arrays of type `AIO_`-

`Descriptor_List`. The choice of a private type is also consistent with other similar descriptor types defined in this standard.

The solution to the dangling reference problem for buffers is to require that the buffer pointer be of a specific type declared in a library level package so that the storage should persist for the life of the process. If a programmer attempts to use unchecked deallocation on such a buffer or to use unchecked conversion to obtain a value of the `IO_Array_Pointer` type, it becomes the responsibility of the programmer to ensure against dangling references. The idea was considered of providing a deallocation operation that would check that there are no outstanding I/O operations on a buffer that is deallocated, but there seemed to be no simple way to do this check without kernel involvement.

B.6.3.2 Return Status and Error Status

For AIO, the *aio_error()* and *aio_return()* functions provide completion and error status information that is provided for other POSIX operations via the `POSIX_Error` exception and the `Get_Error_Code` function (parameterless). `Get_AIO_Status` binds to *aio_error()*, but only returns `Completed_Successfully`, `In_Progress`, or `Canceled`. All other *errno* values raise `POSIX_Error`. The error code can then be obtained by calling `POSIX.Get_Error_Code` from the exception handler for `POSIX_Error`. An alternative to `Get_AIO_Status` is the function `Get_AIO_Error_Code`, which is a more direct binding to *aio_error()*. It was added in response to ballot objections that in a time-constrained situation one should not be forced to use an exception handler just to determine whether an outstanding I/O operation is in progress. It is primarily to support use of this function that the error codes `Operation_In_Progress`, `Operation_Completed`, and `Operation_Canceled` were added to package `POSIX`.

The `Get_Bytes_Transferred` function binds directly to *aio_return()* and has the limitations documented in the C-language interface: the return value is undefined if the I/O operation is still in progress and `Get_Bytes_Transferred` may be called exactly once for a given asynchronous operation. Therefore, `Get_AIO_Status` or `Get_AIO_Error_Code` should be invoked to ensure that a given asynchronous operation is no longer in progress before invoking `Get_Bytes_Transferred`.

B.6.3.3 File Status Flags

In POSIX.1, O_DSYNC or O_SYNC is passed to *aio_fsync()* to specify the kind of synchronization desired. Constants corresponding to *O_DSYNC* and O_SYNC are declared in `POSIX_IO` where they are used as option sets (*i.e.*, flags). However, these constants are not used as flags when they are passed as parameters to the *aio_fsync()* operation. Inconsistent type usage has been avoided by splitting *aio_fsync()* into two Ada operations.

B.6.3.4 Asynchronous I/O Priority

The `Integer` type is used for the parameter of `Set_Priority_Reduction` and the return value of `Get_Priority_Reduction`. This value is subtracted from the process priority to obtain the priority of the I/O operation. The type is, therefore, required to be the same as for process priorities, which is `Integer`. Since the value

is required to be nonnegative, the subtype `Natural` is specified. No upper bound is specified because the range of allowed priority reductions may not be determinable until run time.

B.6.3.5 Buffers

Buffers for AIO operations are specified as explicit Ada access types to the type `Stream_Element_Array`, which is an array type. Where the C-language interface specifies a pointer to an array and a length as separate **`in`** parameters, they are replaced here by a pointer to an Ada unconstrained array; the `Length` attribute of the array actual parameter specifies the length. The actual number of bytes transferred is obtainable via the function `Get_Bytes_Transferred`.

B.6.3.6 AIO Descriptor List

`AIO_Descriptor_List` is defined to be an array of `AIO_Descriptors`. `AIO_Descriptor_List` is the type corresponding to the C *aiocb* pointer array. It was necessary to split the function *aio_suspend()* into two forms to handle the case of a single control block pointer. The C-language interface only specifies a form for an array of pointers, assuming that a single pointer can be treated as if it were an array of length one. In the Ada binding, two distinct types are involved.

Also, it is not specified what happens if the *nent* parameter of *aio_suspend()* is zero or negative. This issue is avoided in the Ada interface by specifying the index range of `AIO_Descriptor_List` to be positive.

B.6.3.6.1 Constraint Error

Because AIO operations may require an unpredictable length of time to complete, it is particularly important to initiate AIO operations only on values of type `AIO_Descriptor` that have been created by `Create_AIO_Control_Block` and not yet destroyed by `Destroy_AIO_Control_Block`. Otherwise storage may be compromised in an unpredictable manner. An implementor can add a measure of safety by marking or invalidating the value of type `AIO_Descriptor` when the corresponding AIO control block is successfully destroyed, though this standard does not require it to do so.

`POSIX_Error` with an error code of `Invalid_Argument` was selected to correspond to the detection of an attempt to initiate an AIO operation on an invalid value of type `AIO_Descriptor`. However, implementations may substitute `Constraint_Error` since that may be raised by the runtime system anyway if the value is invalid for the type.

Implementors are not required to detect this condition, and there may be situations where detection would be difficult. Particularly troublesome is the case in which the application makes copies of a valid value of type `AIO_Descriptor` through assignment and then passes one of the copies to `Destroy_AIO_Control_Block`. Depending on the implementation, an AIO operation on another copy has an unspecified result.

B.7 Device- and Class-Specific Functions

Section 7 of this standard follows very closely the same-numbered section of POSIX.1. A large part of the text is a direct copy from that section. No functionality changes are intended to have occurred in the translation. The model of behavior is that a sequence of bits that constitute a unit of data transmission, called a *byte* in this standard, is read in via the serial line and then converted to a `POSIX_Character` before being given to the application.

One additional function, *ctermid()*, from 4.7.1 of POSIX.1, has also been included in Section 7 as function `Get_Controlling_Terminal_Name`, since this Section 7 contains a discussion of the controlling terminal concept.

The only major change from POSIX.1 is to treat the terminal characteristics as a private type and supply the appropriate access functions and modification procedures. Initially, one might feel use of these access and modificaton operations will cause much lower performance, but an inline implementation will have the same performance as a C-language structure reference. An unsuccessful attempt was made to develop structure modification operators (`+` and `-`) for manipulating the `Terminal_Characteristics` type. This attempt was abandoned because not all of the component data types were `Boolean` types and thus use of the same operators could cause confusion.

Error checking of the `Terminal_Characteristics` modification routines is required in this standard where it is not in the C-language binding to POSIX.1. If the system can support all the functionality of the interface, all of the checking will not be required.

For historical reasons, the preferred values for `Null_POSIX_Character` and `Flag_POSIX_Character` are `POSIX.POSIX_Character'VAL(0)` and `POSIX.POSIX_Character'VAL(16#FF#)`, respectively, but this standard does not require that assignment of values.

B.8 Language-Specific Services for Ada

B.8.1 General Rationale

A compiler or library or runtime system that conforms to both the Ada RM {1} and this standard would provide two I/O systems to the application writer:

(1) The I/O operations that are made available to the application by Section 5, Section 6, and Section 7 of this standard, generally referred to as *POSIX I/O* in this document.

(2) The Ada I/O operations that are made available to the applications by Annex A of the Ada RM {1}, generally referred to as *Ada I/O*' in this document.

The two I/O systems do not have identical objectives. The POSIX I/O system has the objective of making the I/O model of POSIX.1 available to the Ada programmer. The Ada I/O system has the objective of providing a more convenient and more portable set of I/O capabilities, that are integrated with the Ada language and do not depend on having a POSIX compliant operating system.

Section 8 has the objective of rationalizing the Ada I/O model within POSIX I/O system; it does not have the objective of incorporating all of the functionality of the POSIX I/O model. Instead, it interprets relevant portions of the Ada RM {1} and constrains and details some of the implementation dependencies permitted by the the Ada RM {1}, so that Ada I/O is more completely defined in a POSIX environment. Applications needing the full functionality of the POSIX I/O model should use the POSIX I/O system directly.

Because both sets of I/O operations are available, it is possible that a given collection of application programs will use both sets of operations. For this reason, it is desirable to permit the interchange of external files so that they can be read and updated by the use of either set of I/O operations after being created and written by the other set of I/O operations.

Interchange of data is deemed to be useful along any one or combination of the following dimensions:

— Ada I/O operations and POSIX I/O operations

— Programs compiled by Ada compilers and by C compilers

— Programs produced by different Ada compilers

A complete mapping between the POSIX and Ada I/O operations is quite difficult primarily because of a lack of underlying standardization concerning external representations of data. It is, however, practical to provide a reconciliation for a useful subset of I/O operations.

The approach is intended to satisfy the following objectives:

— Compatibility with currently available UNIX system Ada compilers.

— Interoperability (ability to interchange external data) between POSIX I/O and a subset of Ada I/O so that
  — Ada input operations can read data written by Ada output operations.
  — POSIX input operations can read data written by Ada output operations.
  — Ada input operations can read data written by POSIX output operations
  — Access to POSIX standard input, output, and error files can be achieved via Ada I/O operations.

In order to circumvent some of the shortcomings of `Text_IO`, the idea of providing a `POSIX_Text_IO` package, which would serve as a functional replacement for `Text_IO` and which would incorporate additional or modified functionality, was considered. This concept was rejected because of the expense it would place upon implementors. They would still be required to implement standard Ada `Text_IO` as well as `POSIX_Text_IO` and the POSIX I/O facilities described in Section 5 and Section 6. Requiring the implementation of three I/O systems seemed excessive in a language binding. Moreover, reliance on yet another set of I/O facilities would reduce portability of Ada code to other (non-POSIX) systems.

It remains possible to perform formatted I/O on POSIX strings by exploiting the character set mapping defined in Section 2 of tihs standard. The functions `POSIX.To_String` and `POSIX.To_POSIX_String` will convert between POSIX strings and the corresponding Ada strings. The Ada strings can be processed using normal `Text_IO`.

The implementation is required to close Ada files (files opened using `Text_IO` or any instantiation of `Direct_IO` or `Sequential_IO`) upon termination of the Ada program so that the external files corresponding to the Ada file objects are in a predictable state. POSIX.1 already required that POSIX files be closed when a POSIX process terminates. Because an Ada active partition corresponds to a POSIX process, this additional requirement probably amounts only to flushing Ada I/O buffers upon partition termination.

It is intended that open Ada file objects must be closed when a nonerroneous Ada application terminates by reaching the end of the main subprogram or by terminating due to an unhandled exception. It is desirable, but not required, that the implementation also close the file objects when an erroneous Ada application terminates or when the corresponding POSIX process dies because it does not catch a POSIX signal.

This standard requires that the flushing of output buffers and the closing of open file objects must occur after the termination of the main subprogram of the partition and after the termination of the library units, but prior to the termination of the POSIX process. The reason for this additional standardization is to ensure that the external files corresponding to Ada file objects are left in a predictable state whenever possible.

In addition to the flushing required at application termination, the package `POSIX_Supplement_to_Ada_IO` provides a mechanism for the application explicitly to flush output files. This mechanism was provided with some reluctance because of the burden that it places upon implementors to deal with private types outside of the packages in which the private types were defined. Nevertheless, it is necessary because the execution of `POSIX_Process_Primitives.Start_Process` may cause some ambiguity regarding the status of data that are contained within Ada output buffers, but that are not yet written to the underlying operating system. It is suggested that an application should flush all output files before starting a new process.

B.8.2 Limitations on Interoperability

As mentioned in the B.8.1 the extent of interoperability is necessarily limited. The intended limits of interoperability are described in this subclause.

It is intended that Ada Text I/O be interoperable with POSIX I/O to the extent permitted by character set limitations. The character set limitations are inescapable because the Ada RM {1} precisely defines the character set to be processed by Ada programs. Ada character literals and strings are defined to consist of only these characters. The Ada Text I/O operations are defined to operate using this character set. POSIX.1 permits greater latitude in the character sets, and so it, in general, cannot be assumed to correspond with the Ada character set. This problem is treated by requiring that a core set of Ada characters and POSIX characters be placed in an invertible mapping as described in Section 2.

In addition to characters and strings, Ada Text I/O provides operations for I/O upon integer, fixed point, floating point, and enumeration types. Each of these types is converted to or from a character representation in accordance with the rules of Ada. By utilizing these rules, a program using POSIX I/O can exchange data with a program using Ada Text I/O.

Ada Text I/O incorporates a concept of line terminators and page terminators that is not native to POSIX. The specifications in the Ada RM {1} allow some degree of freedom in the implementation of these terminators. This standard defines a form of Text I/O that restricts that freedom by defining those terminators in terms of the Ada character set in a manner that permits Ada Text I/O to read and write text files in the format typically expected by POSIX.

Ada also provides facilities for performing Sequential I/O and Direct I/O upon arbitrary types. Traditionally, these operations are implemented by using an external representation that is identical or closely related to the internal representation. In general, the internal representations are highly dependent upon the hardware representation of integers, as floats, *etc*. The authors of the Ada RM {1} recognized this tradition by permitting the external representation to be implementation dependent. Fixing this problem would require a great deal of additional standardization of external formats. Furthermore, Sequential I/O and Direct I/O can be used to perform I/O upon data aggregates such as arrays and records. Their representation is also highly implementation dependent, especially in the case of unconstrained arrays and variant records. Fixing this problem would also require substantial additional standardization. For these reasons, the interchange of data between Ada Sequential and Direct I/O and POSIX I/O is left as implementation dependent.

Even within a given machine, defining interoperable data formats is difficult. Defining interoperable formats between Ada and POSIX is more difficult because Ada `Sequential_IO` and `Direct_IO` are oriented to records while POSIX I/O is oriented to bytes. Attempts to read Ada values from an arbitrary POSIX file could be imagined as reading the number of bytes corresponding to the internal Ada representation of the object, but what should be done with leftover bytes at the end of a file that is not of exactly the right length? Presumably, the leftover bytes constitute a type mismatch that is discovered long after the first type mismatch (the first record of the file) occurred. The only Ada model that corresponds neatly with the POSIX I/O model is one of instantiating `Sequential_IO` or `Direct_IO` to deal with individual bytes. Using an instantiation of either of these packages would require an execution of an Ada `Read` or `Write` operation for every individual byte of the file. Using the POSIX I/O system may be more satisfactory in this case.

Some other languages, of course, superimpose record structuring upon the POSIX I/O model. Achieving interoperability with programs written in other languages is a potentially fruitful standardization activity that falls outside the scope of an Ada binding and, hence, is left for future standardization efforts.

Because Ada I/O operations may look ahead for the end-of- file marker, it is possible that vendors may wish to implement Ada Text I/O operations with a layer of buffering hidden from both the application program and from the underlying operating system. So operating upon a single external file with interleaved Ada I/O data transfers and POSIX I/O data transfers has undefined results.

B.8.3 Rationale for Form Parameter

Section 8 of this standard relies heavily upon the `Form` parameter to provide additional interpretation to the `Open` and `Create` procedures. This usage is specifically

sanctioned by A.10.1 of the Ada RM {1} to describe system-dependent characteristics of these operations.

A typical example of the Form parameter follows:

```
New_File : File_Type; ...
Create
  (File => New_File,
   Name => "NewFile",
   Mode => Out_File,
   Form => "Owner => Read_Write," &
           "Group => Read," &
           "Other => None," &
           "Blocking => Tasks");
```

The Form_String and Form_Value functions provide a convenient capability for constructing and parsing Form parameters. Implementors are encouraged to supply a package containing their own analogous functions to deal with implementation- defined fields within the Form. For purposes of example, consider that these functions are called Vendor.Form_String and Vendor.Form_Value. An application could parse a Form_String applying both Form_Value and Vendor.Form_Value to the string. Form_String ignores fields that are syntactically well formed, but that have nonstandard field names; therefore, no exception is raised for the implementation-defined fields. If the application wishes to construct a Form string, it can concatenate the results of Form_String and Vendor.Form_String to include both standard and implementation-defined fields.

The record types Form_Values_for_Open and Form_Values_for_Create have default values provided for each of their components to reduce the likelihood that instances of the records will have uninitialized components. The type Possible_-File_Descriptor, which is used as a component of Form_Values_for_Open, is a variant record because a POSIX.File_Descriptor value may or may not actually be present in the field. The alternative of using an access type was considered; in this case, the null value would be distinguished to mean that no POSIX.File_-Descriptor is present. This alternative was rejected because the use of access types in a binding may permit implementations that leak storage.

Despite the similarity in the names, the Blocking field of the Form parameter is not equivalent to the O_NONBLOCK flag of POSIX.1. Nonblocking I/O in the sense of C is not consistent with the Ada model of I/O; all Ada I/O is blocking in the sense that C uses the term. The Blocking field provides a choice regarding how much of the program should be blocked by an I/O operation—the entire Ada program or only the calling task. Because of wide variations in implementations (for example, distributed processors), it is not practical to require that all implementations support both behaviors. The range of functionality implemented for Text_IO may be broader than that implemented for POSIX I/O because the Ada runtime system is an additional resource that the implementor may exploit to synthesize additional behavior. The type POSIX.Blocking_Behavior enumerates the behaviors defined by this standard. The constant POSIX.IO_Blocking_Behavior gives the behavior selected by the implementation. The subtype POSIX.Text_IO_Blocking_Behavior enumerates a possibly broader set of behaviors supported for Text_IO. Standard input and

standard output use the behavior selected by `POSIX.IO_Blocking_Behavior`. The enumeration literal `Tasks` was selected in preference to `Task` because the latter is a word reserved by Ada.

The `Form` parameter does not provide functionality equivalent to that provided by the O_EXCL, O_NOCTTY, and O_TRUNC flags of POSIX.1 I/O. To gain that functionality, an application program should use the POSIX I/O functions described in Section 6. The reasons for not providing the functionality to the application are described as follows:

— O_EXCL: At the time POSIX.5 was drafted, some members of the drafting committee believed that there was no need for an interface to specify the O_EXCL flag. Their interpretation of the Ada language `Create` procedure was that it must raise `Use_Error` if there already exists a file with the specified pathname (similar to POSIX open with O_EXCL and O_CREAT). Apparently, this view is not universally accepted.

NOTE: This issue may deserve further attention in the next revision to this standard.

Table B.1 indicates a rough correspondences between the two models. The two left-hand columns indicate the settings of the O_EXCL and O_CREAT flags when a POSIX.1 *open()* operation is executed. The right-hand column identifies the Ada procedure with the same intent.

Table B.1 – Correspondence of File Creation Flags

O_EXCL	O_CREAT	Ada Procedure
On	Off	Not defined by POSIX.1
Off	Off	`Open`
On	On	`Create`
Off	On	No Ada equivalent

A.8.2 (3,7) of the Ada RM {1} has the effect of prohibiting the final case listed in Table B.1. Paragraph 3 says that `Create` must create a new file. Paragraph 7 says that `Open` raises `Name_Error` if no external file with the given name exists.

— O_NOCTTY: According to 7.1.1.3 of POSIX.1, the initial identification and subsequent replacement of the controlling terminal is implementation defined. The only use of the O_NOCTTY flag is to prevent a terminal device file from becoming the controlling terminal. The intent of this standard is that a file object opened or created by Ada I/O would never become the controlling terminal; i.e, the `Open` and `Create` procedures are interpreted as if the O_NOCTTY flag were set. The normative text does not totally forbid access to the controlling terminal from the Ada I/O model. If a POSIX file descriptor is already associated with the controlling terminal according to the rules of the implementation, then it is possible to achieve access to the controlling terminal by associating an Ada file object with the POSIX file descriptor. For example, one could use `Standard_Input` and/or `Standard_Output` to access standard input and/or standard output if they were associated with the controlling terminal. If an arbitrary file descriptor were associated with the controlling terminal, then one could use `Open` with the `File_Descriptor` option to achieve access to the controlling terminal.

— O_TRUNC: In the Ada model of I/O, functionality similar to that provided by the flag O_TRUNC is obtained by executing the `Text_IO.Reset` procedure.

Other syntaxes were considered for the file permission fields of the `Form` parameter. One alternative was similar to the conventional syntax of *chmod()*. Another utilized a subset of Ada aggregate syntax. These two alternatives were rejected because they hinted at functionality that could not be completely provided. The selected syntax was chosen primarily because it can be efficiently parsed at run time.

Alternatives to the `File_Descriptor` field of the `Form` parameter were considered. In particular, the use of a stylized file name to reference a file descriptor number was considered. This solution offered some advantages but was rejected because it had the effect of forbidding certain file names to the user. Furthermore, the inclusion of both actual file names and POSIX file descriptors within the same type seemed to be a subversion of the normally strong typing practices used in Ada programming. In POSIX, it is quite clear that file names and file descriptors are distinct entities. For example, it is possible for multiple file descriptors to refer to a single file and for the same file descriptor to refer to different external files, all within the lifetime of a single execution.

The use of `Append` on a terminal device is not regarded as an error because the physical characteristics of the device cause implicit appending of the output. Use of `Append` on a device that cannot support appending does not cause an exception because the underlying POSIX abstraction does not provide an error code in such circumstances. The interpretation of `Create` with `Append => True` was carefully considered. Three alternatives were as follows:

(1) If the file exists, then append to it; if the file does not exist, then create it.

(2) Ignore the `Append` indication.

(3) Raise an exception.

The first option is attractive because it roughly corresponds to functionality available from POSIX I/O. Unfortunately, it violates a requirement of A.8.2 (3) of the Ada RM {1} that `Create` "establishes a new external file". The second option was rejected because it misleadingly hints at the rejected functionality of the first option. Therefore, the third option was adopted.

A.14 of the Ada RM {1}adarm states that it is not specified by the language whether the same external file can be associated with more than one file object and specifies certain requirements if file sharing is supported by the implementation. Implementing file sharing is difficult because of the internal buffering implicit in the Ada I/O model. Vendors who choose to implement concurrent access, totally or in part, may wish to give particular consideration to the case where the external file is accessed by one or more Ada file objects that have the file opened for append and sharing that file with the Ada file object that created the file. In this case, an output operation upon any of the file objects would ideally result in appending the data to the end of the file. Vendors may also wish to give special consideration to the case where external files are shared among separate processes.

The `Form` field `Page_Terminators => False` is provided to permit reading and writing the typical format of text files in POSIX. Users who wish to exchange text files with a program written in another language should use this field.

The treatment of text file terminators is a troublesome subject. The text of the Ada RM {1} describes the processing of Text I/O subprograms in terms of line ter-

minators, page terminators, and file terminators. For example, it specifies that the end of a text file is specified by the combination of the line terminator followed by the page terminator and then the file terminator. A.10 (8) of the Ada RM {1} states that the actual nature of terminators is not defined by the language. It is desirable for an Ada binding to POSIX to select a representation so that text files may be interchanged between programs written in different languages.

The most obvious way to map the Ada terminators to a representation in the external file is to map the line terminator to line feed, the page terminator to form feed, and the file terminator to physical end of file. This mapping, however, has several unpleasant side effects. For example, since all files written with such a mapping will end in line feed/form feed/end of file, directly writing such a text file to a CRT would end by immediately clearing the screen in some implementations. Furthermore, performing text input of many typical POSIX text files would result in failure if each form feed were not preceded by a line feed.

In attempting to define a mapping that would avoid such problems, it was found that various implementors have already solved the problems, but in different ways. It became apparent that any standardization of terminator representation would break some existing implementations.

AI-00172 {B7} provided the solution to the problem. It says that an implementation may assume that certain external files do not contain page terminators and may enforce this assumption by refusing to recognize any input sequence as a page terminator. Furthermore, it invites implementations to use a field within the `Form` parameter to identify such files and suggests that `Text_IO.Standard_Input` may always be assumed to be such a file. The `Form` field `Page_Terminators => False` is provided to allow the application to indicate that it will take advantage of the interpretation provided by the AI.

AI-00172 {B7} applies only to the treatment of files when they are input to an Ada program. It remains to define how such files may be output (see A.10 (7) of the Ada RM {1}) requires that the end of a file is marked by the sequence line terminator, page terminator, file terminator (see A.10 (7) of the Ada RM {1}) says that the actual physical representation of the terminators is implementation defined. In the case of a file with `Page_Terminators => False`, this standard chooses the following representations: line terminator is the character mapped to `Ada.Characters.Latin_1.LF`; file terminator is the physical end of file; and the page terminator is not physically represented at all. In other words, an output file with `Page_Terminators => False` is a single page that has a single (conceptual, but not physical) page terminator just before the end of file. The physical representation is indistinguishable from a file with no page terminators whatsoever, but is still well formed from the Ada point of view.

Additional details of files read and written with `Page_Terminators => False` are specified to ensure the ability to interchange them between programs written in different languages. On output, the Ada page terminator is not represented in the file, so a text file consists of sequences of characters separated by line feeds and terminated with a line feed and physical end of file. This treatment of page terminators corresponds to the structure of a typical POSIX text file. Upon input, the form feed is not recognized as a page terminator. Any occurrence of `ASCII.FF` is simply input to

the program as a character. (Presumably, it appears in the file because it was placed there by the writing of a `ASCII.FF` character.)

On files without page terminators, `Text_IO.New_Page` is specified to raise `Use_Error`. The alternative, of course, is simply to omit writing the page terminator to the file; but that would mean ignoring an obvious error and would seem to contradict directly the description of `New_Page` in A.10.5 (16) of the Ada RM {1} . A.13 (10) of the Ada RM {1} sanctions the use of `Use_Error` when an operation is attempted that is not possible for reasons that depend on characteristics of the external file.

For similar reasons, `Use_Error` is raised when `Text_IO.Set_Line` attempts to start a new page.

The presumption that a file contains no page terminators has interesting effects upon other operations of `Text_IO`. (These effects are described here in rationale rather than in the normative text because they are not part of POSIX.1. Instead, they are an interpretation of the ramifications of the Ada RM {1} in this situation.)

— `Close` or `Reset` of an output file without page terminators does not raise `Use_Error`, notwithstanding the text in the Ada RM {1}. A.10.2 (3-4) of the Ada RM {1} does not say that `New_Page` is actually called, it says that `Close` and `Reset` have the effect of calling `New_Page`. To assume that `Use_Error` would be raised would be to assume that one could never `Close` or `Reset` a file that is intended subsequently to be read in a manner conforming to AI-00172 {B7}. This matter was submitted to the Ada Rapporteur Group for clarification and has resulted in AI-00886 {B9}, which approves the position.

— `End_of_Page` operates identically with `End_of_File`.

— Execution of `Text_IO.Set_Page_Length` with any length other than zero (unbounded) raises `Use_Error`. The function `Text_IO.Page_Length` returns the value of zero, meaning unbounded.

— `Skip_Page` on such a file would read and discard all of the characters remaining in the file up to the file terminator. A subsequent execution of `Skip_Page` would raise `End_Error`.

The selection of the default value caused some controversy. `Text_IO.Standard_Input` and `Text_IO.Standard_Output` act as if they were opened with `Page_Terminators => False` because that corresponds to the format that users expect in the POSIX standard input and output files. It was preferable to have the default be `False` for all files, but it was thought that, notwithstanding AI-00172 {B7}, a compiler implementing this interpretation would fail the Ada Compiler Validation Capability tests that serve to certify the conformance of the compiler to the the Ada RM {1}.

B.8.4 Unaffected Implementation Dependencies

Below is a list of some of the additional implementation dependencies that are allowed by the the Ada RM {1} and that remain unaffected by this standard. They are listed here for the convenience of the reader.

A.7 (6) of the Ada RM {1}:

> "The effect of input-output for access types is implementation defined."

A.8.2 (2) of the Ada RM {1}:

> "For direct access, the size of the created file is implementation dependent."

A.10 (8) of the Ada RM {1}:

> "The effect of the input or output of control characters (other than horizontal tabulation) is not defined by the language."

A.10.1 of the Ada RM {1}:

> "The ranges of type `Count` and subtype `Field` of package `Text_IO` are implementation defined."

B.8.5 Notes on Specific Topics

B.8.5.1 Alternative National Character Sets

In the development of POSIX.5 there were suggestions that provision should be made for alternative national character sets. No such provision was made for the following reasons:

(1) The objective of Section 8 of this standard is to interpret the requirements of Annex A of the Ada RM {1} in terms of POSIX. It is beyond the scope of this effort (indeed, of the entire POSIX effort) to attempt to change the standard Ada language. The text of the Ada RM {1} defines Text I/O operations upon characters as acting upon the Ada type `Standard.Character`. It remains possible for the user to make interpretations upon existing data types that would have the effect of representing alternative character sets; these possibilities remain permissible under this standard.

(2) POSIX.1 and Ada 83, which were the base standards at that time, did not have provisions for national character sets. Provisions made in the POSIX/Ada binding would be quite vulnerable to obsolescence when alternative national character sets are incorporated in the Ada and POSIX standards.

Ordinary and wide character types are defined based on the ISO 8859-1 and 10646 BMP international character sets in A.1 of the Ada RM {1}, and permission is given to add child packages of `Ada.Characters` for symbols of the local character set in A.3.3 (27) of the Ada RM {1}. However, the standard I/O operations defined by the Ada language are only defined for the type `Standard.Character`.

B.8.5.2 Interoperability of `File_Type` and `File_Descriptor`

It has been suggested that this standard should provide a mechanism for interconverting between `Text_IO.File_Type` and `POSIX_IO.File_Descriptor`. These suggestions are not incorporated into this standard because they seem useful only for intermixing Ada I/O and POSIX I/O data transfer operations on a single open file. It is not a goal of the current standard to permit such practice. Such a goal was considered and rejected on the grounds that it would be exceedingly difficult to write this standard in a way that did not present onerous problems to implementors. However, this standard does provide a mechanism, using the `Form` parameter, to open an Ada file object that is to be associated with an open POSIX file identified by a `File_Descriptor`.

B.8.5.3 Access to standard error

It is virtually a requirement of good POSIX programming practice to use the standard error file for the output of error messages. Although A.10.1 of the Ada RM {1} now provides the function `Standard_Error` to return a value of type `File_Type` for the standard input and output files, this function was not available in Ada 83. The example below illustrates how an application program may open a file type for the standard error file, using Ada 83and this standard.

```
-- Example of access to standard error
with POSIX_IO;
use POSIX_IO;
with Text_IO;
use Text_IO;
 ...
   Standard_Error : File_Type;
 ...
   Open
     (File => Standard_Error,
      Mode => Out_File,
      Name => "Stderr",
      Form => " Append           => True,   " &
              " Page_Terminators => False,  " &
              " File_Descriptor  =>         " &
             File_Descriptor'Image (POSIX_IO.Standard_Error));
```

NOTE: It may be advisable for the next revision of this standard to make explicit the relationship of `Standard_Error` in `POSIX_IO` and `Ada.Text_IO`.

B.8.5.4 `Data_Error` on `Get` Procedure

It has been suggested that it is not cost-effective to check the bytes that are input by Text I/O to assure that they satisfy the Ada type constraint on `Character`. In fact, it has further been suggested that the Ada RM {1} does not require `Text_IO.Get` to check input characters to ensure that they fall within the bounds of type `Character`. A.13 (13) of the Ada RM {1} includes the following: "The exception `Data_Error` can be propagated by the procedure `Read` if the element read cannot be interpreted as a value of the required type. This exception is also propagated by a procedure `Get` (defined in the package `Text_IO`) if the input character sequence fails to satisfy the required syntax, or if the value input does not belong to the range of the required subtype." In the the Ada RM {1}, the exception seems optional in the case of `Read`. In the case of `Get`, it appears to be required (the Ada RM {1} uses *is* in the sense of *shall* throughout the document). Nevertheless, this standard does not make any presumption in this regard. It requires only that any error checking required of `Get` by A.8 (13) of the Ada RM {1} be applied to the input characters after they have been mapped in accordance with the requirements of Section 2 of this standard. Implementors should note that the rules of Section 2 regarding character set mapping permit the definition of a mapping that has the effect of avoiding the occurrence of `Data_Error` on `Get`.

In particular, if the mapping is specified so that all non-Ada characters of the underlying character set are mapped to one or more Ada characters, then it would never

be necessary to raise `IO_Exceptions.Data_Error`. It would still be necessary, of course, to perform the conversions implied by the mapping.

B.8.5.5 Stream I/O

Stream-oriented I/O operations have been introduced in Ada 95. However, there appears to be a close match between the *read()* and *write()* operations of POSIX.1 and the `Read` and `Write` operations of `Ada.Streams.Stream_IO` defined in A.12.1 of the Ada RM {1}. At the time this standard was last revised, there was not yet much experience with this new Ada feature; therefore, no binding between the Ada stream I/O and the POSIX I/O operations is defined. During the ballot process for POSIX.5b, a partial connection was established by the introduction of I/O operations on the type `Stream_Element_Array`.

A more thorough look at the appropriate relationship between Ada streams and POSIX I/O would be an appropriate subject for a future revision to this standard.

B.9 System Databases

B.9.1 Requirements from POSIX.1

The group database contains the following fields:

— Group name.

— Numerical group ID.

— List of names/numbers of all users allowed in the group.

The user database contains the following fields:

— Login name.

— Numerical user ID.

— Numerical group ID.

— Initial working directory.

— Initial user program.

Both databases may contain implementation-defined fields.

POSIX.1 uses the ambiguous phrase “names/numbers” to desribe the content of the *gr_mem* field of the group database structure. The type of *gr_mem* is a *char***, implying that the entries are character strings. They are strings, but they may either contain names or consist solely of a string of digits that may be treated by the user as numbers.

POSIX.1 does not guarantee that a value obtained from `POSIX_Process_Identification.Image` (given a valid value of type `User_ID`) can be used as the user name provided to `Get_User_Database_Item`.

The initial user program is usually a shell program. This program is executed at the end of the login sequence. If the initial user program field in the user database contains a null string, the system default initial user program shall be used. This

standard (like POSIX.1) does not provide any additional definition of how the system default is established or how the name of the system default initial program can be obtained.

The operations on the group database are *getgrgid()* (group ID) and *getgrnam()* (name). The operations on the user database are *getpwuid()* (user ID) and *getpwnam()* (name).

B.9.2 Rationale for the Current Design

B.9.2.1 Portability Versus Extensibility

One of the most fundamental agreements is that the POSIX packages in this standard should represent a completely portable interface wherever possible. POSIX.1 permits the system database type declarations to be extensible by the implementor. The developers of this standard did not see how to provide a portable interface for accessing implementation extensions. Therefore, they decided to define packages containing the guaranteed portable POSIX services and permit implementors to define other packages that define their extensions.

A second agreement, closely related to the first, is that the representation of a database entry would be private, with operations to return information from the structure. Hiding the representation of database entries is good Ada style and permits substantial freedom in implementing the binding. In particular, there is no specific requirement that the implementor construct a record containing all possible data elements when the user only requires one specific element. Not having to store values for unused data elements is very useful when the system database information is not contained within a single file.

It is strongly recommended that any implementation-defined package use the same names and types as the POSIX standard package.

B.9.2.2 Iterators

The group member list is implemented using a private type and the iteration scheme, as described in B.1.

There is no iterator over the entire user or group database. This omission is deliberate in POSIX.1 and is reflected in this standard. The rationale for POSIX.1 notes that “The [UNIX linear search operations] provide a linear database search capability that is not generally useful ... and because in certain distributed systems, especially those with different authentication domains, it may not be possible or desirable to provide an application with the ability to browse the system databases indiscriminately.”

B.9.2.3 Pathnames, Filenames, and User and Group Names

POSIX pathnames are values of `POSIX_String`. Therefore, the functions that fetch pathnames return values of type `POSIX.POSIX_String`. Likewise, user and group names are specified as POSIX strings. Using the same type for all these strings makes the interoperability between user names and other POSIX strings much simpler because no type conversion is required.

At one time, consideration was given to providing distinct subtypes `User_Name_String` and `Group_Name_String` as unconstrained subtypes of `POSIX.POSIX_String`. This would have provided a bit of extra type naming documentation, but only as a convention since in this case there would be no compile-time checking nor even any runtime constraint checking.

B.10 Data Interchange Format

There is no Ada binding to Section 10 of POSIX.1. This section of POSIX.1 actually does not fit well in the system API since it defines a data interface primarily of interest to implementations of the `tar` file archive utility. There has been no interest in developing an Ada-specific binding for this section of POSIX.1. However, the section number is reserved, primarily to preserve the correspondence of section numbers between this standard and POSIX.1.

B.11 Synchronization

Semaphores, mutexes, and condition variables are important tools for synchronization. Semaphores are intended for use primarily between *processes*. Mutexes and condition variables overlap in functionality with the rendezvous and with protected operations. However, semaphores and (optionally) mutexes and condition variables are useful between tasks in different processes, and are intenteded to facilitate interoperability between Ada and C code.

B.11.1 Package `POSIX_Semaphores`

B.11.1.1 Semaphore Types

In POSIX.1 both the semaphore object and a pointer to the semaphore object are used in the interface. The main challenge to the Ada binding was to provide equivalent functionality, but at the same time to provide safe usage and allow implementations the freedom to allocate the semaphore either at the point of declaration, or out of line in user or system space.

Explicit declaration of semaphore objects was required in order to allow for shared anonymous semaphores (which were considered an essential capability). Explicit declaration permits the user to allocated unnamed semaphores in shared memory. The implementation is still allowed to allocate data structures associated with the semaphore out of line, but is responsibile for supporting access to the semaphore object from different processes.

However, unless a handle model is adopted, complete safety is not possible due to the loophole in Ada that allows copies of private types values (even limited) to be created (as a result of function results and passing **`in out`** parameters). It is an implementation choice for an Ada compiler whether **`in out`** parameters are passed by copy or by reference. Usually, this decision is based on the size of the parameter. For this reason, a correct implementation of this standard must arrange to pass the `Sem` parameter to `Initialize` and `Finalize` by reference. Implementations (of the binding) can arrange for this by adjusting the internal characteristics of the type based on the Ada compiler in use.

The final model adopted is that all the semaphore operations operate on a handle object, a `Semaphore_Descriptor`. The `Semaphore` type is still visible to allow for in-line allocation of such objects (for unnamed semaphores only). But the only way to operate on these objects is via the handle returned either by `Descriptor_Of` (for unnamed semaphores) or by `Open` or `Open_Or_Create` (for named semaphores). This approach is not completely safe since references to finalized/closed semaphores are still possible. However, in practice such errors can be avoided by appropriate programming conventions. Since the problem cannot be completely fixed in any case and the other capabilities are important, the decision was to accept this trade-off.

The `Semaphore_Descriptor` is not a visible access type for two reasons:

(1) Requiring it to be an access type would unnecessarily limit implementations.

(2) The interesting operations on access types (*e.g.*, allocators, dereferencing) are not meaningful for semaphore descriptors and are, therefore, excluded.

While creating copies of semaphore objects (`Semaphore`) has undefined effect, copies of semaphore handles (`Semaphore_Descriptor`) are allowed since they always point to the same semaphore object.

Procedures `Finalize` and `Close` take `Sem` as an **`in out`** parameter in order to allow the implementation to set the value of the parameter to some illegal value after the operation completes, which facilitates detection of erroneous attempts to use the finalized or closed semaphore. Since other copies of the same `Semaphore_Descriptor` value may exist this technique will not protect against the erroneous use of such copies. However, it is believed that some protection is better than none and that programming conventions can address the problem of use of copies.

B.11.1.2 Semaphore Initialization and Opening

In POSIX.1 *sem_init()* (for unnamed semaphores) and *sem_open()* (for named semaphores) differ in whether the semaphore is assumed to be shared. Named semaphores opened via *sem_open()* can always be shared among processes. The *sem_init()* function has an argument *pshared* to tell whether the (unnamed) semaphore can be shared. However in a single-threaded process, the meaning of *pshared* when it equals zero is unspecified. Of course, in a multithreaded environment a nonshared semaphore can be used by different threads in the same process. Since Ada requires support for multitasking, both values of `Is_Shared` are always meaningful and are, therefore, allowed.

The *sem_open()* function has been split into two operations, according to the value of the *oflag* argument, *O_CREAT*; this splitting is for consistency with the treatment in POSIX.5 of `Open` for files. For the same reason, the original **`out`** parameter that returns a semaphore descriptor is changed to a function return value.

As the initial state of a semaphore is considered important to code comprehensibility, no default is specified for `Value`. Thus, the initial state of a semaphore must be explicitly provided when it is initialized or created.

B.11.1.3 Waiting on a Semaphore

The Ada operation corresponding to the C function *sem_trywait* does not raise an exception corresponding to the error code *EAGAIN*. `Try_Wait` returns a `Boolean`

value to indicate success or failure of the waiting operation. Other sources of error (*e.g.*, an invalid semaphore descriptor) will cause an exception.

`Try_Wait` is a function, rather than a procedure; therefore, it can be used conveniently in **`while`** loops. Also, this operation has no `Masked_Signals` parameter since (unlike `Wait`) it is inherently nonblocking.

B.11.1.4 Unblocking from a Wait on a Semaphore

The effect of thread priority on the selection of which task to unblock when a semaphore is posted is intentionally less specified in the Ada binding than in POSIX.1. One reason is that the Ada and C-language bindings were developed and balloted concurrently by different groups of people. Successive drafts of the C-language threads standard made subtle changes between uses of the words *thread* and *process* that left uncertainty about whether the highest priority thread is required to be unblocked, or just the highest priority process. Rather than risk having to back out inconsistent specifications later, the Ada binding balloting group chose to say nothing on this subject. A second reason is to permit this standard to be implemented over a system that does not provide kernel support for multithreaded processes. Without kernel support for threads it may be impractical to guarantee priority order wakeup at the thread level from blocking kernel calls. Indeed, it may be that when a task waits on semaphore it blocks all the tasks in the process (see B.2.4.2). c

B.11.1.5 Semaphore Values

Intentionally, the POSIX.1 document only partially specifies the value that is returned by *sem_get_value()* when the semaphore is locked. The value returned may be positive (*i.e.*, the actual value of the semaphore), or it may be a negative number whose absolute value represents the number of tasks waiting for the semaphore at some unspecified time during the call. For this reason, the value returned by `Get_Value` is of type `Integer`. On the other hand, the initial value of the semaphore is always nonnegative (since a negative value would imply a nonempty queue of blocked tasks already waiting on the semaphore). Therefore, `Initialize` and `Open_Or_Create` do not allow the initial value of a semaphore to be set to a negative value, and consequently, the `Value` parameter of these operations is of subtype `Natural`.

B.11.2 Mutexes and Condition Variables

The interface defined in packages `POSIX_Mutexes` and `POSIX_Condition_Variables` follows quite closely the corresponding interface in POSIX.1.

Since this standard always requires support for tasks, it makes no sense to provide a direct Ada mapping for the _POSIX_THREADS option (on which C-language support for threads, as well as mutexes and condition variables, depends). Therefore, this binding substitutes the Mutexes option, which controls the availability of mutex and condition variable operations.

B.11.2.1 Intended Use of Mutexes and Condition Variables

Mutexes and condition variables do not provide any special functionality beyond the built in task synchronization features introduced by Ada 95, which include protected c

objects (see B.11.2.5) and the package `Ada.Synchronous_Task_Control`. However, at the time POSIX.5b was balloted the Ada 95 standard was not yet approved and none of the commercial Ada compilers supported the new task synchronization primitives; the only standard task synchronization mechanism available was rendezvous.

An interface to the POSIX.1 thread synchronization primitives is included in this standard based on the assumption that (for many applications on many implementations) such primitives will be more efficient than the Ada rendezvous. This will almost certainly be true if the Ada runtime system is layered over POSIX.1 threads and is, therefore, forced to use mutexes and condition variables to implement rendezvous. With such an implementation it would be more efficient for an application to use mutexes and condition variables directly for situations (like simple mutual exclusion) that do not require the full power of rendezvous.

If Ada tasking is not implemented over a system that supports POSIX.1 threads the Ada runtime system will probably not use mutexes and condition variables internally, since there are more efficient ways of implementing Ada multitasking. With such an implementation the mutexes and condition variables required by this standard would be implemented using the internal task synchronization mechanisms of the particular Ada runtime system. In this case one should not assume that using mutexes and condition variables will be any more efficient than rendezvous.

If the compiler and Ada runtime system support Ada 95 protected objects mutexes and condition variables will probably not provide any performance advantage. If the Ada implementation is layered over a system that supports the POSIX.1 thread synchronization primitives there is no need for mutexes and condition variables to be more efficient than Ada protected objects, and if the Ada tasking implementation is not layered over POSIX.1 threads mutexes and condition variables are likely to be less efficient.

Similarly, a good implementation of the package `Synchronous_Task_Control` can be more efficient than mutexes and condition variables, since the semantics of the operations are inherently simpler.

In any case, protected objects are much safer than mutexes and condition variables, since the interactions with other Ada language features (*e.g.*, exception propagation, task abort, asynchronous transfer of control, and finalization) have been worked out carefully. For example, protected operations are safe for use with asynchronous task abort, but if a task is aborted while holding a mutex the application will deadlock. (See B.11.2.5.)

The mutexes and condition variables defined by this standard might be useful for synchronization between C and Ada components of a mixed-language application, but not portably. This standard does not address all the issues that would need to achieve at level of interoperability. For example, the memory representation of an Ada `Mutex` object need not be the same as that of a C-language pthread_mutex_t object (the Ada `Mutex` object may be a reference to the actual mutex or it may contain additional data components), in which case calling `Lock` directly on an imported C mutex object would not work.

Perhaps the best argument that can be made for the utility of POSIX mutexes and condition variables in an Ada application is for portability of software designs. Hav-

ing semantically consistent types and primitives available to both C and Ada applications allows a component originally coded in C to be recoded in Ada, and *vice versa*, without altering the design. The same virtue applies, of course, to the rest of the interfaces defined in this standard. c

B.11.2.2 Use of Descriptors

The motivation for having the `Mutex_Descriptor` and `Condition_Descriptor` types (and their treatment) is similar to that for the `Semaphore_Descriptor` type. Mutexes and condition variables may need to be allocated in shared memory, just like unnamed semaphores, to be useful between processes.

Type `Attributes` may denote the attributes object itself or a reference (index or access) to the attributes object.

B.11.2.3 Shared Mutexes and Condition Variables

POSIX.1 specifies the Process Shared option. If this option is supported mutexes and condition variables can be shared between processes. Otherwise, these synchronization mechanisms may be used only among tasks in the same process. In order to be sharable, the Process Shared option must be supported, and the mutex or condition variable must be initialized with `Is_Shared` set to `True`.

B.11.2.4 Ceiling Priorities and Unblocking Behavior

The ceiling priorities associated with mutexes are intended to be related to Ada priorities, so that support for this feature does not rely on support for Section 13 of this standard. However, when the Priority Task Scheduling option is supported, ceiling priorities also must lie within the range of priorities for the POSIX scheduling policy `FIFO_Within_Priorities` for compatibility with POSIX.1.

The selection of the task to be unblocked when a mutex is unlocked and the order of unblocking of tasks when a condition is either signaled or broadcast are specified by POSIX.1 when Priority Task Scheduling option is supported. Otherwise, the order of unblocking of tasks is unspecified.

POSIX.1 says that if a signal is delivered to a thread that is blocked on a mutex, then upon return from the signal-handler, the thread is to be blocked again as if it was not interrupted. This requirement does not appear in the Ada binding since the application has no visibility of signals being delivered asynchronously. The same is true of the wait operation for condition variables.

Incidentally, *cond_timedwait()* is the only function for which the C interface specifies error code *ETIMEDOUT* instead of *EAGAIN*. The Ada binding preserves this distinction, specifying error code `Timed_Out` for a condition timed wait that expires.

B.11.2.5 Comparison to Protected Types

Ada 95 contains the *protected objects*, monitor-like constructs designed to provide, among other things, data synchronization. Protected operations serve much the same requirements as mutexes and condition variables for Ada applications, but are

Protected procedures provide similar functionality to mutexes. A protected procedure can only be executed by one task at a time; other tasks attempting to call protected operations of the same protected object cannot proceed until the executing task completes its protected action. Protected objects can be used to provide mutual exclusion for critical sections, much as a mutex does. The only functionality of mutexes not supported is the ability of a task to unlock mutexes in an arbitrary order since protected procedure calls must be properly nested. However, this is viewed as an advantage of protected objects since unlocking mutexes in other than last-in-first-out order is considered by many to be an unwise programming practice.

An important difference is that the locking mechanism used to implement protected objects is not considered potentially blocking, in the sense that protected actions (other than entry calls) are permitted within other protected actions but mutex locking operations are not.

Protected entries provide similar functionality to that of condition variables. In the case of entries, the actual conditions that should be checked on returning from a condition wait are included in the entry declaration. For the following condition wait

```
pthread_mutex_lock(mutex);
while (! condition) pthread_cond_wait(&condition,&mutex);
```

has a similar effect to calling the protected entry `Wait` of the following protected object:

```
protected type Condition_Wait is
  entry Wait;
  procedure Signal;
private
  Condition: Boolean:= False;
end Condition_Wait;
protected body Condition_Wait is
  entry Wait when Condition is
  begin
    Condition:= False;
  end Wait;
  procedure Signal is
  begin
    Condition:= True;
  end Signal;
end Condition_Wait;
```

A task making a call on an entry whose barrier is `False` is blocked until the barrier becomes `True`. When exiting a protected procedure, a check is made to see whether any tasks are waiting on entry queues of the same protected object. If there are, the corresponding barrier expressions are reevaluated; if any become true, one waiting task is made eligible to execute. The operation `Signal` in this case makes the barrier true and wakes up the task in a single operation, as opposed to POSIX.1, which separates the signal action from the state of the condition being awaited for. The protected type interface has no equivalent to a condition variable wait without an associated `Boolean` condition and loop, but that is a virtue since condition variables are prone to spurious wakeups (which make usage without a `Boolean` condition and loop unsafe).

While the function of these two code fragments is similar, there are significant dif-

ferences in semantics. For example, if a task is waiting for the `Wait` entry barrier to open and another task attempts to exit the `Signal` procedure the task waiting on the `Wait` entry will be the next task to execute in the protected object, regardless of whether or not higher priority tasks are waiting on other entry queues. By contrast, when the POSIX.1 condition variable is signaled, it must contend for the mutex again. If higher or equal priority tasks are attempting to lock the mutex, they may get it and execute before the task that was waiting on the condition variable. This difference make the behavior of protected objects more predictable and less vulnerable to race conditions.

A significant difference between the use of mutexes and the use of protected objects is that with bare mutexes there is no abort deferral. Thus, if a task is aborted or there is an asynchronous transfer of control while the task is holding a mutex the mutex may not be released, leading to deadlock or undefined behavior (if the same task later tries to lock the mutex). For this reason, it is recommended that mutexes not be used in tasks that are subject to abortion or asynchronous transfer of control. A similar caution applies to the use of mutexes with code that might raise an exception.

B.11.2.6 Threads Considerations

This section defines operations that may block an entire process. As with all such operations, POSIX.1 specifies that they need only block the calling thread (but they may still block the whole process). As with all blocking system calls, there is a potential change of semantics when moving between a POSIX implementation that does not support threads and one that does. In particular, using semaphores to synchronize between Ada tasks within a single process may be workable if POSIX threads are supported and Ada tasks map to POSIX threads, but otherwise it is likely to cause deadlock within the process.

B.12 Memory Management

B.12.1 Process Memory Locking

The packages `POSIX_Memory_Locking` and `POSIX_Memory_Range_Locking` are very direct bindings to the C interface. In the case of the operations on the whole address space, such a direct binding seems safe and appropriate for use by an Ada application programmer. In the case of the range locking operations, it does not seem so appropriate.

The memory range locking operations seem awkward to use in a portable fashion. For locking, an Ada application programmer will have difficulty determining the exact correspondences between memory ranges and Ada entities. It is well known that the `Address` and `Size` attributes are not portable, especially in the presence of so-called *dope* (implicitly allocated record and array components used by the Ada implementation for address computations and constraint checking). It is also the case that the values of certain objects may reside only in registers some or all of the time; thus, it may not be meaningful to speak of the address of some objects.

For unlocking, there also seems to be a problem determining whether two objects occupy the same page of memory, *i.e.*, whether the space of one can be unlocked with-

out accidentally also unlocking the space of the other. (These problems are similar to those with memory mapping, which is discussed in B.12.2.3.)

Nevertheless, the memory range locking interface is provided since, if it is needed, nothing less may suffice. A separate package is used partly because the range locking interface is controlled by a separate option, but mainly to isolate it from the safer memory locking interface on the entire process address space.

The implementation is allowed to impose the restriction that the starting address and the length in storage units (not necessarily the same as bytes) of the memory range to be locked/unlocked must be page-aligned. `POSIX_Page_Alignment` is provided in Section 2 to ease the calculation of such page-aligned quantities.

B.12.2 Package `POSIX_Memory_Mapping`

This interface may be used to map portions of a process address space either to portions of a regular file or to shared memory objects. One subtlety to keep in mind is that the lengths of memory regions are specified in storage units (to be compatible with Ada address arithmetic), but the lengths of file regions are specified in bytes. Depending upon the value of `System.Storage_Unit`, these two may or may not be the same.

B.12.2.1 Options

Some of the options are derived from `POSIX.Option_Set`, including `Protection_Options` and `Synchronize_Memory_Options`. Other options are defined as constants of an implementation-defined type, including `Mapping_Options` and `Location_Options`. The decision about which approach to use was based ultimately on whether a reasonable default value could be specified or whether the user should be forced to specify one of the possible choices.

The `Protection` parameter to `Map_Memory` and the `Options` parameter to `Synchronize_Memory` have the obvious default value of `Empty_Set`. The possibility of providing default values for the `Mapping` and `Location` parameters of `Map_Memory` was considered, but was rejected on the grounds that no obvious reasonable default values exist.

B.12.2.2 Map Memory

The C function *mmap()* has been split into two overloaded versions of the functions `Map_Memory`. One version allows the user to specify a suggested starting address and requires a location option to be specified. The other version omits the suggested starting address and location option and allows the implementation complete freedom in locating the address mapping. Splitting into two distinct operations is believed to be clearer than the C interface convention of interpreting a suggested starting address of zero as allowing the implementation to locate the memory mapping.

B.12.2.3 Required Representation Support and Shared Variable Control

In response to a ballot objection, this standard imposes the requirement to support the pragmas `Atomic` and `Volatile` of Ada 95 in order to suppress unwanted compiler optimizations.

Without some of the new features of Ada 95 memory mapping would be more unsafe for Ada users. Such problems seem to be inherent in the semantics.

A big problem with the use of shared memory is the danger of undesired compiler optimizations, namely the possibility that the compiler may assume that the value of a variable can only be modified by code generated by that compiler and within the same program. Based on this assumption and data flow analysis, the compiler may omit loads and stores to memory, keeping the value of the variable in a register. It may also delete entirely any code that computes values that appear to be dead, *e.g.*where the value computed is assigned to a variable that is overwritten before being read again. This kind of optimization is incorrect if the variable is in shared memory.

To prevent such undesired (and harmful) optimizations on objects that are in shared memory, some way is needed to inform the Ada compiler that these objects may be read and updated concurrently by other processes. The pragmas `Volatile` and `Atomic`, provide this capability. In the case of Ada 83, where these pragmas are not available, several work-arounds are known. One that may work on some implementations is to use access values to denote the objects in shared memory, in order to reduce the likelihood of data flow analysis detecting candidates for such undesired optimizations. The other alternative is to use implementation-defined pragmas or compilation flags to suppress optimization.

Another problem is uncertainty about what the Ada runtime system may be already doing with shared memory; for example, if the runtime system is using multiple POSIX processes to implement a single Ada active partition (as it may do to reduce the blocking effect of certain system calls or to take advantage of a multiprocessor) there is danger that the application may interfere with the Ada runtime system. Also, there is uncertainty about the exact memory layout used by the Ada language implementation; this problem also arises with memory range locking (see B.12.1). Ada compilers are not uniform in the way they support the type `System.Address` and address clauses. In particular, some compilers do not allow nonstatic expressions in address clauses. Even the recommended level of representation support specified in C.2 of the Ada RM {1} does not include support for nonstatic expressions.

In summary, to make good use of the memory-mapping and memory-region locking capabilities, an application needs to be able to

(1) Do arithmetic on addresses. (See 2.9.)

(2) Convert addresses to usable access-type values.

(3) Specify the address of a data object via an address clause with a nonstatic expression.

(4) Insert system calls to set open files and shared memory objects and establish mappings before the elaboration of the Ada objects that are to occupy the mapped address ranges, including objects declared in library packages.

(5) Suppress optimizations that are not correct in the presence of concurrent access to data from another process.

The operations provided by this standard, combined with those provided by the Ada language, are believed to address all these requirements.

B.12.3 Package POSIX_Shared_Memory_Objects

The present interface for opening shared memory objects is based on the POSIX.5 treatment of Open and Open_Or_Create for files. The functions here have "_Shared_Memory" appended in order to distinguish them from the corresponding operations on files in package POSIX_IO since they have identical parameter-result profiles. The only difference between the function specifications, for overloading, is that the shared memory functions define the File parameter to be of type POSIX.POSIX_String, while the POSIX_IO functions define File as type POSIX.Pathname, which is defined as a subtype of type POSIX.POSIX_String.

Incidentally, a similar overloading problem holds for the various unlink operations. Thus, for all unlink operations added by POSIX.5b the operation name specifies what is being unlinked (*e.g.*, Unlink_Shared_Memory, Unlink_Semaphore, *etc.*).

The option set scheme used is consistent with POSIX.5. In particular, the Exclusive option is used on Open_Or_Create_Shared_Memory to specify that the operation should fail if the named object already exists. This issue also arises for named semaphores and messages queues and is resolved in the same way.

B.12.4 Package POSIX_Generic_Shared_Memory

The generic package POSIX_Generic_Shared_Memory is intended to provide a more abstract interface to shared memory objects than POSIX_Shared_Memory_Objects. The operations of this package represent only a subset of the capabilities of the lower level interfaces, but it is far easier to use. This interface appears to be adequate for most Ada applications where shared memory is used as a fast way of sharing data between processes.

B.12.4.1 Intended Use

The package POSIX_Generic_Shared_Memory uses a simpler parameter set than the other packages defined in Section 12 of this standard. For example, the operations Open_And_Map_Shared_Memory and Open_Or_Create_And_Map_Shared_Memory have no Mode parameter as the operations Open_Shared_Memory and Open_Or_Create_Shared_Memory have to specify Read_Only or Read_Write. Instead, the corresponding value is set for the generic interface according to the value of the Protection parameter. Applications needing to specify missing parameters will need to use the lower level operations defined in this section.

B.12.4.2 Initialization of Shared Memory

An application instantiates POSIX_Generic_Shared_Memory supplying a user-defined type for shared memory objects, then calls the procedure Open_Or_Create_And_Map_Shared_Memory to obtain a file descriptor to a new shared memory object of the user-defined type. This new shared memory object is not required to be initialized, even if the actual type corresponding to the generic parameter Object_Type has default initialization. Though this is inconsistent with the Ada language definition, requiring correct default initialization of shared memory objects was considered too burdensome for implementers. There are several implementation problems.

One problem is the need for synchronization and mutual exclusion. If one process has created a shared memory object and is initializing it, another process should not be able to open and access the object between the time it is created and the time initialization is complete. Likewise, an object should not be initialized more than once. Initialization requires some form of synchronization and mutual exclusion, but this is just the first instance of a need that continues for every subsequent access to the shared memory object. As mentioned above, the user is expected to use other features of this standard to achieve such synchronization and mutual exclusion.

Another problem is that the code for default initialization is generated implicitly by the Ada compiler. If default initialization is required, the implementer of the body of this generic package will need to rely on the compiler to generate the initialization code. The known solution involves the use of an address clause for a local object of type `Object_Type`. However, some Ada compilers suppress initialization for objects with address clauses. Therefore, it is not specified whether default initialization takes place when a shared memory object is created via this generic interface.

Some Ada language implementations may require initialization of certain types of objects, such as those that require dope. In such cases, the implementation of this standard must either reject the generic instantiation or provide the necessary initialization.

With Ada 95, there is an alternative way of arranging for the compiler to initialize objects in shared memory, that should be more reliable. Shared memory objects may be allocated and initialized via the standard **`new`** operator, using a user-defined storage pool as defined in 13.11 of the Ada RM {1}. All that is needed is for the user-defined allocator to return an address that is mapped to the shared memory object.

B.12.4.3 Accessing Generic Shared Memory

`Open_And_Map_Shared_Memory` and `Open_Or_Create_And_Map_Shared_Memory` return a file descriptor to the application for use in later operations (just as for the nongeneric interface). One reason is that the supply of file descriptors available to a process is generally limited. If an abstract handle type were introduced here, that would hide the fact that each open shared memory object consumes a file descriptor, which is not available for opening other files. Another reason for making the file descriptor visible is that all the memory management operations that take file descriptors as parameters can be used directly on these objects. This eliminates the need for a special operation to unlink a (generic) shared memory object and also provides sophisticated users with the means of achieving functionality beyond the intended scope of the package (perhaps with some loss of portability). A third reason for using a visible file descriptor is to permit sharing of the file descriptor with subroutines written in other languages, such as C. Some disadvantages, however, are also apparent. POSIX defines many operations on the `File_Descriptor` type, including some whose effects are not defined for shared memory objects. Moreover, the use of file descriptors obtained via this generic interface as parameters in direct calls to the lower-level memory management operations from the `POSIX_Memory_Locking`, `POSIX_Memory_Range_Locking`, `POSIX_Memory_Mapping`, and `POSIX_Shared_Memory_Objects` packages is undefined.

A generic shared memory object cannot be mapped to the region of the process address space associated with an Ada object declaration. Instead, the interface provides the function `Access_Shared_Memory`, which, given a file descriptor returned by one of the open functions, returns a access value of type `Shared_Access` that may be dereferenced to obtain access to the shared memory object.

NOTE: In a future revision to this standard it may be advisable to provide a way to map a shared passive partition to a file or shared memory object.

B.12.4.4 Locking and Unlocking Shared Memory

The interface does not provide for locking in the sense of protecting shared memory objects against concurrent access by more than one process. A user can provide this capability, at whatever granularity is desired, using other features of this standard. In particular, it is possible for `Object_Type` to be a composite data type that includes one or more unnamed semaphores as components.

The lock and unlock operations provided serve to suppress and enable virtual memory paging of shared memory objects, relieving the user of the need to be concerned with the address computation needed to apply `POSIX_Memory_Range_Locking`.

B.13 Execution Scheduling

While compatible and interoperable implementations of POSIX threads and Ada tasks are possible, the scheduling models of POSIX.1 and Annex D of the Ada RM {1} each allow implementations that would not be compatible with each other. That both standards are loosely defined and yet different may be viewed as a sign of agreement in the POSIX and Ada communities that the time is not yet ripe for tight standardization of real-time scheduling behavior. Both POSIX and Ada try to leave considerable freedom to the implementation. Unfortunately, the specific freedoms allowed by each standard do not match up exactly, nor are the conceptual models on which the standards are based exactly the same.

In the long term, the need to support both C threads and Ada tasks on the same platform may lead to consensus on the relationship between C threads and Ada tasks and on the appropriate requirements to achieve language interoperability and application portability. However, at the time POSIX.5b was developed there was very little experience with implementations of POSIX.1 multithreading or Annex D of the Ada RM {1}.

Therefore, the goals for Section 13 of this standard are as follows:

(1) To provide Ada applications access to task scheduling services analogous to those provided for C threads by POSIX.1.

(2) To allow the Ada implementor to choose the best way of implementing Ada task scheduling semantics using the services of the particular underlying operating system (which may not support POSIX.1 multithreading) for an expected class of applications.

To accomplish the former, one must define interfaces. To accomplish the latter, one must avoid overspecifying the interactions of these interfaces with Ada tasking.

For these reasons, Section 13 is a thinner and more direct binding to POSIX.1 than the rest of this standard. The developers of POSIX.5b believed the resulting stylistic incongruity would be less objectionable than a self-contained binding that is inconsistent with the base standards or is incompatible with the future evolution of realtime scheduling practices. The standard in its present form requires the reader to refer to POSIX.1 for the behavior of the interface.

B.13.1 Scheduling Concepts and Terminology

The definitions of many scheduling terms are taken verbatim from the Ada RM {1} or from POSIX.1 (which are reasonably compatible with one another). One exception is the concept of Scheduling Contention Scope defined in POSIX.1. This standard merely defines constants corresponding to the C constants and refers readers to POSIX.1 for their meanings. The reason is that POSIX.1 says very little that a portable application can count on.

There are minor differences in terminology and behavior between the Ada RM {1} and POSIX.1 which can cause the unwary some difficulty. For an example of a difference in terminology, see the note on the difference between the Ada term “ready” and the POSIX term “runnable,” in 2.2.2.147. For an example of a difference in behavior, compare the definition of the `Sched_FIFO` scheduling policy to the definition of the Ada `FIFO_Within_Priorities` task dispatching policy. For the latter, a delay statement will cause the calling task to move to the end of its scheduling queue even if the delay is too short to block the task. The POSIX `Sched_FIFO` policy does not require this unconditional round-robin effect for the POSIX operations with timeouts.

B.13.2 Package `POSIX_Process_Scheduling`

This package is intended to be as direct a mapping as possible of the facilities of POSIX.1 for process scheduling.

B.13.2.1 Scheduling Parameters

The only scheduling parameter defined by POSIX.1 is priority. However, the approach taken by this standard easily supports future extensions that may add more scheduling parameters.

B.13.2.2 Scheduling Policies

Constants are defined corresponding to the three scheduling policies defined by POSIX.1. Readers are referred to that standard for their meanings.

B.13.2.3 Modify Process Scheduling Policy and Parameters

These operations are a direct mapping to corresponding operations in POSIX.1.

B.13.2.4 Process Yield CPU

This operation is a direct mapping to the corresponding operation in POSIX.1b, whose effect was redefined by POSIX.1c. This standard must allow for, but cannot require, that the underlying system support the POSIX.1 multithreading option; therefore, the meaning of this operation is deliberately vague.

B.13.2.5 Get Scheduling Limits

These operations are a direct mapping to corresponding operations in POSIX.1.

Priorities here are intentionally of type `Integer`, for consistency with the C interface and for consistency with the precedent established by Ada, for task priorities. The various ranges of priorities supported by the various scheduling policies is not known statically; therefore, defining bounds on this range is not likely to be helpful.

The anticipated practice is for a user to call the functions `Get_Maximum_Priority` and `Get_Minimum_Priority` to obtain upper and lower bounds that can be used to define a subtype, which is then used within the application.

B.13.3 Task Scheduling

13.3 is an attempt to harmonize the thread scheduling facilities of POSIX.1 with the task scheduling facilities of Annex D of the Ada RM {1}.

B.13.3.1 Approach

The present approach is a compromise between two extreme positions expressed among both the technical reviewers and the balloters. One extreme position is that all thread scheduling facilities of POSIX.1 are specific to the C language and should not appear in an Ada binding. The arguments against this extreme are as follows:

— It is then difficult to rationalize the scheduling effects of the mutex and condition variable synchronization facilities, particularly in mixed-language applications.

— The task scheduling model of Annex D of the Ada RM {1} is more specific than the POSIX.1 thread scheduling model in two respects. First, Annex D of the Ada RM {1} requires that all tasks within an active partition be scheduled with the same dispatching policy, whereas POSIX.1 allows each thread in a process to be scheduled with its own policy. Second, Annex D of the Ada RM {1} defines only a single task dispatching policy, roughly corresponding to *Sched_FIFO*, whereas POSIX.1 also provides the policy *Sched_RR*.

The other extreme position is that the Ada binding should include a direct binding to POSIX.1 thread scheduling facilities as an alternative to the Annex D of the Ada RM {1}. The main argument against this extreme is that it presents a POSIX view of Ada tasking in conflict with the Ada RM {1} and leaves unanswered questions of interaction with other Ada language constructs.

Thus, the present binding is a compromise. It provides a binding to POSIX.1 thread scheduling, but it is more abstract and more limited than the C language facilities. The advantage is that it allows for the possibility that the implementor can harmonize the scheduling semantics of POSIX.1 and the Ada RM {1} and achieve efficient implementation of the POSIX task scheduling interface.

There is no binding to the multithreading option of POSIX.1 since this standard requires all implementations to support Ada tasking, regardless of whether the underlying operating system supports POSIX.1 multithreading. The Priority Task Scheduling option corresponds to the priority thread scheduling option of POSIX.1.

B.13.3.2 Dynamic Priorities

The thread scheduling parameters interface of POSIX.1 is mapped by this binding to the dynamic priority facilities of Annex D of the Ada RM {1}, since priority is the only thread scheduling parameter defined by POSIX.1. Should a future extension add more thread scheduling parameters, this issue will have to be revisited.

B.13.3.3 Task Dispatching Policy Pragma

The thread scheduling policy interface of POSIX.1 is mapped by this binding to extensions to Annex D of the Ada RM {1}. The mapping is not a simple matter of using the `Task_Dispatching_Policy` pragma to define the new POSIX scheduling policies, since this pragma is a configuration pragma and applies the same task dispatching policy to all tasks of the active partition. It was necessary to define a two-step interface.

First, a new identifier `POSIX_Task_Dispatching` is defined for the `Task_Dispatching_Policy` pragma. Use of this value enables the second step. Second, a new `Task_Creation_Attributes` pragma is added. This pragma applies to a task type, and may be used to specify the scheduling policy (*e.g.*, `Sched_FIFO`, `Sched_RR`, `Sched_Other`) and the scheduling contention scope (*e.g.*, `System_Wide`, `Within_Process`) for all task objects of the task type, so different tasks within the process can have different scheduling policies and different scheduling contention scopes.

POSIX.5b intentionally left fuzzy the exact behavior of the POSIX scheduling policies because of the aforementioned differences in detail between POSIX scheduling rules and Ada scheduling rules. It is not the intent of this binding to require different tasking support in the runtime system than is required for Ada validation. Since there was no one clearly best way to rationalize in detail the POSIX thread scheduling model with the Ada task scheduling model, without imposing unreasonable limitations on the Ada language implementation, it is left to the implementor.

B.13.4 Synchronization Scheduling

B.13.4.1 Semaphores

POSIX.1 defines semaphores and specifies the effects on process scheduling of this synchronization mechanism. The main provision is that when multiple threads are blocked waiting on a semaphore, they must be unblocked in priority order for the policies *Sched_FIFO* and *Sched_RR*, but this standard must allow for implementations that do not support Ada multitasking but do not support POSIX.1 multithreading. It was difficult to know how this should be translated into the unblocking of tasks, without overspecifying the interface, so the wording was left imprecise (*i.e.*, “the task to be unblocked shall be selected in a manner appropriate to the scheduling policies and parameters in effect for the blocked tasks and their processes”).

B.13.4.2 Mutexes and Condition Variables

The mapping of synchronization effects for mutexes and condition variables is much more direct, replacing the term “thread” by the term “task” and using the Ada names for attributes and policies. Basically, the behavior of POSIX.1 is required:

(1) The inheritance effects for mutexes are the same as provided in POSIX.1.

(2) The unblocking of multiple tasks waiting on a condition variable or trying to lock a mutex is the same as provided in POSIX.1: for the policies Sched_FIFO and Sched_RR, tasks are unblocked in priority order.

The assumption here is that if this option is supported, either there is an underlying operating system that supports POSIX.1 multithreading, or mutexes and condition variables are implemented directly by the Ada runtime system.

B.13.4.3 Open Questions

The following is a partial list of the scheduling issues left open by this standard:

— Are POSIX threads supported by the underlying system? And if so, are Ada tasks implemented using the threads facilities?

— What is the effect of process scheduling on task scheduling?

— What is the default Scheduling Policy and Priority of a process?

— Which values of the Scheduling Contention Scope attribute are recognized by the system and what is their effect on task scheduling across an Ada application consisting of multiple active partitions?

— Is Annex D of the Ada RM {1} supported? If so, then what are all the interactions with the interfaces defined by this standard? For example,

  — What is the mapping from Ada priorities (for tasks) to POSIX priorities (for processes and for mutex ceiling priorities)?

  — Do the priority inheritance effects of locking mutexes through the interface defined by this standard affect the order of entry queue service as defined in D.4 (11) of the Ada RM {1}? If so, how?

  — For the POSIX task scheduling policies Sched_FIFO and Sched_RR, which scheduling rules apply? Those of POSIX.1 or those of the Ada RM {1}(suitably extended)?

B.13.5 Thread Scheduling Pragmas vs. Environment Variables

POSIX.1 supports thread creation with the function *pthread_create()*, which takes an attribute structure describing the type of thread to be created. Task creation in Ada is part of the language and takes place at the elaboration of the declaration or evaluation of the allocator creating the task. The kind of task created is determined by the declaration of the task, including pragmas in the task specification.

In POSIX.1, the result of scheduling two threads with different scheduling policies together is generally implementation defined. Therefore, if an application wants to use a special scheduling policy it needs to ensure that all threads in the system are created with this policy, *e.g.*, by modifying the initial thread using *pthread_setschedattr()* to change the initial thread to this policy and then specifying that policy in the attributes structure used to create all further threads.

Ensuring that all the tasks in the system are created with the same scheduling policy is more difficult to arrange for Ada tasks. By the time that the environment task can call a procedure to affect its own scheduling attributes, other tasks may

already have been created by elaboration of task object declarations with whatever default attributes are defined by the Ada runtime system. Thereafter, any attempt to change the scheduling policy will result in a system of tasks with various policies with implementation defined effects.

This standard does not directly address the problem, although two possible solutions were considered:

(1) Provide pragmas to specify the scheduling attributes desired at task creation.

(2) Provide environment variables to specify the scheduling attributes desired for the environment task, and make all other tasks inherit the attributes of the task creating them.

Presumably, the `Task_Creation_Attributes` pragma could be extended to specify that task attributes are to be inherited from the creator of the task being created. However, it was felt that the alternative of explicitly specifying task creation attributes for each task type was not onerous.

B.14 Clocks and Timers

Section 14 of this standard provides facilities that complement the standard timing facilities already provided in Ada, such as the `Duration` type, the `Calendar` package, and the `Ada.Real_Time` package.

One central issue for Section 14 is how to reconcile these extended clock and timer features with the standard features of Ada. In particular, it might seem nice to require that the standard Ada time operations be implemented using the same time source as the POSIX realtime operations, but there are differences in requirements that make this unwise.

The main functional differences are as follows:

— Unlike the Ada real-time clock, the POSIX CLOCK_REALTIME clock is not required to be monotonic.

— The POSIX timer interface has several capabilities not available with Ada **`delay`** statements. (See B.14.4.)

— The POSIX clock and timer interfaces are interoperable with C-language code, but the Ada time interfaces are not required to be so.

A second central issue is that the timer delivery mechanism may use the `Signal_Alarm` signal. POSIX.5 does not allow an application to provide a handler for this signal since it is likely to be in use by the Ada runtime system to implement the standard **`delay`** and `Calendar.Clock` operations. Any attempt to create a timer that delivers `Signal_Alarm` is treated as an error. A user of timers must specify some other signal.

The possibility of mapping the POSIX.1 clock and timer interfaces to the `Ada.Real_Time` package was considered. This mapping makes some sense, since there is considerable overlap in functionality. For applications where the `Ada.Real_Time` is adequate, it is recommended that that interface be used.

B.14.1 Types and Constants

The `Timespec` type is defined in the `POSIX` package. See the rationale in B.2.4.7.

The C-language type *itimerspec* is mapped to the Ada private type `Timer_State` in order to allow for possible future extensions to `Arm_Timer`. As with other private types in this standard, operations are provided to set and get the defined components of the private type, namely Initial and Interval.

The expected implementation of clock and timer identifiers is as handles to the actual clocks and timers, but it is deemed unnecessary to mention this expectation in the normative part of this standard, since: the name ending in `_ID` implies that handles are intended; all private types in this standard are intended to allow implementation as handles; whether `Clock_ID` and `Timer_ID` actually are handles makes no difference to the interfaces defined in this standard.

Operations in this section are defined to apply to other clocks, in addition to `Clock_Realtime`. Normally, implementors are not allowed to add visible declarations to the package specifications contained in this standard, but here, the addition of constants of type `Clock_ID` is explicitly permitted. To reduce the possibility of unintended name clashes, these names are required to begin with "`Clock_`". The permission to add clock identification constants to this package specification is inconsistent with the stated intent of the standard that it always be possible to determine when an application is using extensions by examining **`with`** clauses.

NOTE: This inconsistency apparently did not cause any objections from the balloting group for POSIX.5b. However, it would probably be unwise for an implementor to take advantage of this permission. There is a possibility that if the issue is raised in the next revision of this standard, the permission might be removed, to reestablish ***`with`****-clause portability.*

B.14.2 Clock Operations

The `Get_Time` operation is more general than the Ada `Calendar.Clock` function since it supports multiple clocks.

There have been complaints from many realtime application developers that Ada provides no standard way to reset `Calendar.Clock`. The `Calendar.Clock` function may be implemented via `Get_Time` for `Clock_Realtime`. This would then permit applications that read `Calendar.Clock` to reset that clock via `Set_Time`.

B.14.3 Timer Creation

This standard provides no binding to the POSIX.1 function *timer_create()* having no signal event parameter. In POSIX.1 if no signal event parameter is specified, a default signal is sent upon timer expiration. Unfortunately, the default signal for `Clock_Realtime` is `Signal_Alarm` which is reserved by this standard to the Ada runtime system. Thus, an application must specify some other signal for `Clock_Realtime`. Furthermore, no other clocks are specified (yet) by this standard. Thus, a portable application can make no use of a `Create_Timer` operation having no signal event parameter. The rather limited functionality offered by such an interface does not seem to compensate for the confusion it would cause.

B.14.4 Timer Operations

The C-language operation *timer_settime()* has a *flags* parameter with one defined value TIMER_ABSTIME. The type of the parameter is mapped to type `Timer_Options`, which is derived from `POSIX.Option_Set`, and the defined value is mapped to the constant `Absolute_Timer`. An overloaded procedure `Arm_Timer` is provided to allow the prior timer state to be returned to the caller.

An early draft of POSIX.5b suppressed the C constant TIMER_ABSTIME and provided two separate versions of the timer-arming operation. As a result of a ballot objection, this standard now provides a closer mapping to the original C-language interface. The change was made to allow more easily for future extensions that might add additional flags and/or additional parameters to *timer_settime()*.

A separate operation `Disarm_Timer` has been added that corresponds to calling *timer_settime()* with a zero value of *value.it_value*. Again, it seems desirable to define a separate operation for a logically distinct function. However, there is the consequence that `Arm_Timer` is now required to detect a zero value for the Initial attribute of `New_State` and to return an error. Treating a zero as a detected error was considered preferable to possible alternatives, such as to allow the behavior to be unspecified, to detect and ignore the condition, or to allow disarming the timer (which is the behavior specified in the C interface).

This standard requires that a timer expiration signal be caught by a task calling `Await_Signal` or executing an **`accept`** statement for a task entry that is attached to the signal of the timer. With these restrictions, timers might appear to have little advantage over the **`delay`** statement. They *are* useful, however, for the following reasons:

— Periodic timers are possible.

— A count of timer overruns is provided.

— Timers may be based on other clocks, such as CPU-time clocks, which may be defined by future POSIX revisions.

— Timers can be used to send/queue signals.

B.14.5 High Resolution Sleep

POSIX.5 chose not to provide a binding for the *sleep()* function since it might suspend the entire process and since the Ada **`delay`** statement is available. Similar reasoning led finally to the conclusion not to provide a binding for *nanosleep()*, since it appears to duplicate the Ada **`delay`** statement as well.

The decision not to provide a binding to *nanosleep()* was taken for the following reasons:

(1) The overlap between *nanosleep()* and **`delay`** is deemed too great to justify a binding.

(2) The function *nanosleep()* does not reliably block only the calling task.

(3) No absolute version of *nanosleep()* is provided by POSIX.1, similar to the Ada **`delay until`** statement (although a further extension of POSIX.1 is under development that may provide an absolute *nanosleep()*).

B.15 Message Passing

B.15.1 Message Queue Attributes

The attributes defined for message queues are a mixture of attributes that apply to the message queue itself (*e.g.*, Max Messages, Message Length, and Message Count) and attributes that apply to an open message queue description (*e.g.*, Options). All applications that have message queue descriptors that refer to the same message queue object should retrieve the same values for the attributes that apply to the message queue itself, but each user may have its own Options attribute. The only option defined by this standard is `Non_Blocking`.

Interestingly, the Options attribute of the `Options` parameter is ignored on a call to `Open_Or_Create`. The blocking behavior of the open message queue description created is governed by the `Options` parameter of type `POSIX_IO.Open_Option_Set`. The only purpose of the Options attribute is to allow changing the blocking behavior via a call to `Set_Attributes` after the message queue is opened. This rather curious interface is a faithful mapping of POSIX.1.

B.15.2 Priorities

It is assumed that the range of message priorities supported by the implementation is not necessarily determinable at compile time. Therefore, the type `Message_Priority` may include more priority values than are actually supported by the implementation.

Message priorities are independent of process and task priorities.

B.15.3 Generic Message Passing

The nongeneric procedures `Send` and `Receive` are direct mappings to Ada of the corresponding C interface to send and receive messages as untyped byte streams.

The generic package `Generic_Message_Queues` may be instantiated to provide operations to send and receive messages of any (constrained) user-defined type. The package is essentially a wrapper for an instantiation of `Unchecked_Conversion`, but also provides an error buffer that might be useful for debugging.

Since instantiations of `Send` and `Receive` may be called with any valid message queue descriptor for the parameter `MQ`, it is not possible for the implementation of this generic package to have any knowledge of the attributes of the specified message queue. In particular, it is not possible to check at the time the package is instantiated that the size of the actual type supplied for `Message_Type` is compatible with the attribute Message Length of any message queue.

Also, it is not possible in general for this interface to enforce reliable type checking on messages. Messages may arrive from programs that are compiled separately and written in different languages. The generic receive procedure may receive a message of the wrong type, but of a size and format that is indistinguishable from a message of the anticipated type.

In situations where messages of several types are expected to be received on the same message queue, the form of `Receive` with a buffer of type `Stream_Element_Array`

should be used. It is the responsibility of the application to parse the message to infer its type. Unchecked conversion can then be used to convert all or portions of the buffer into data of the desired Ada type(s).

If the implementation is able to determine that the received data do not match the anticipated type and raises `Constraint_Error`, the application may (immediately) call `Get_Error_Buffer` to retrieve (a portion of) the raw message data. The raw message data returned by `Get_Error_Buffer` can then be parsed by the application to perhaps determine the nature of the problem.

B.15.4 Notification

Only one process at a time may request notification of arrival of a message on any particular message queue, so there is no question of selection of which process to notify. Also, a pending receiver will get a new message before notification is made for `Request_Notify`.

The request for notification for message arrival is limited to the next message delivery. If notification is then desired for subsequent deliveries to the message queue, the application must re-request notification (as part of the processing in response to the preceding message arrival).

It appears that `Remove_Notify` is most likely to be useful mainly as a recovery or timeout mechanism in case the expected notification is not received.

B.16 Task Identification

This package is included in this standard because a way is needed to identify tasks, notably for `POSIX_Signals.Interrupt_Task`.

The package is as nearly exact a copy as possible, for Ada 83compilers, of the `Ada.Task_Identification` package of the Ada RM {1}. The semantics are by reference.

B.17 Thread-Specific Data

No Ada interface is provided corresponding to the thread-specific data defined in POSIX.1. The thread-specific data interface is considered to be language-specific and, therefore, does not require a binding in this standard.

The package `Ada.Task_Attributes` of C.7.2 of the Ada RM {1} provides similar capability. However, the Ada task attributes are more powerful than the thread-specific data interface defined in POSIX.1 since it allows one task to set and query data specific to another task using the ID of the task as key.

B.18 Detailed Network Interface - XTI

B.18.1 Error Handling

The developers of this standard were faced with a problem concerning the use of *errno* for system errors and *t_errno* for XTI errors. In addition, the conventions of C

this standard allow for only one error handler to capture all errors. The developers were considering the use of two error handlers to resolve this problem of the two error assignments. However, to complicate matters, earlier drafts of P1003.1g had forbidden the use of assignments to *t_errno*. To resolve the P1003.1g issue, the working group submitted a Coordination Ballot to P1003.1g that stated an objection to the forbidding of assignments to *t_errno*. P1003.1g resolved to accept this objection and to modify P1003.1g to remove the prohibition on the assignment to *t_errno*. The developers of this standard agreed that *errno* and *t_errno* can be unified in the Ada binding. Therefore, in this document, the XTI error functions and system error functions are invisible, and there is one exception (`POSIX_Error`) and one function (`Get_Error_Code`) to return codes.

The subtype `XTI_Error_Code` is defined to provide discrete `Error_Code` ranges for the XTI Detailed Network Interface optional services. The subtype `Addrinfo_Error_Code` is also defined to provide discrete `Error_Code` ranges for the Sockets Detailed Network Interface with the Network Management optional services (*i.e.*, to support the `Get_Socket_Address_Info` procedure, which has a similar error handling difficulty). The implementation has the responsibility to ensure that these error code ranges do not overlap or collide with existing system error codes. Figure B.1 depicts the error handling process.

B.18.2 XTI Addresses

The C-language data structures in P1003.1g use the *netbuf* type for numerous other structure members. Where these members denote network addresses, they are mapped to the tagged type `XTI_Address` in the Ada binding. A similar approach is used for `Socket_Address` in Section 18 of this standard. The protocol-specific instances of network addresses are detailed in D.2.

In Draft 4 of POSIX.5c, due to several general objections to dependence on Ada 95 without an Ada 83 fallback approach and several additional specific objections to the application of tagged types the tagged type representation of network addresses was replaced with private pointer types. A general XTI address type is provided in the base package for use with the base package operations, and protocol-specific address types are provide in the child packages (along with functions to convert the pointers between the general and protocol-specific network address types).

B.18.3 Package `POSIX.XTI`

In P1003.1g, the C structure *netbuf* has multiple uses. Because of the strong data types of Ada, this standard maps to the C binding using several private types. Parsing of these data objects is done by providing separate functions and procedures.

In Draft 4 of POSIX.5c, all remaining cases of objects or attributes with the type `Network_Buffer` were removed and replaced with subprogram parameters or explicit pointer types.

The following approach is used for data buffers and references to various other objects. This rationale also applies to data buffers and private types in package `POSIX.Sockets`.

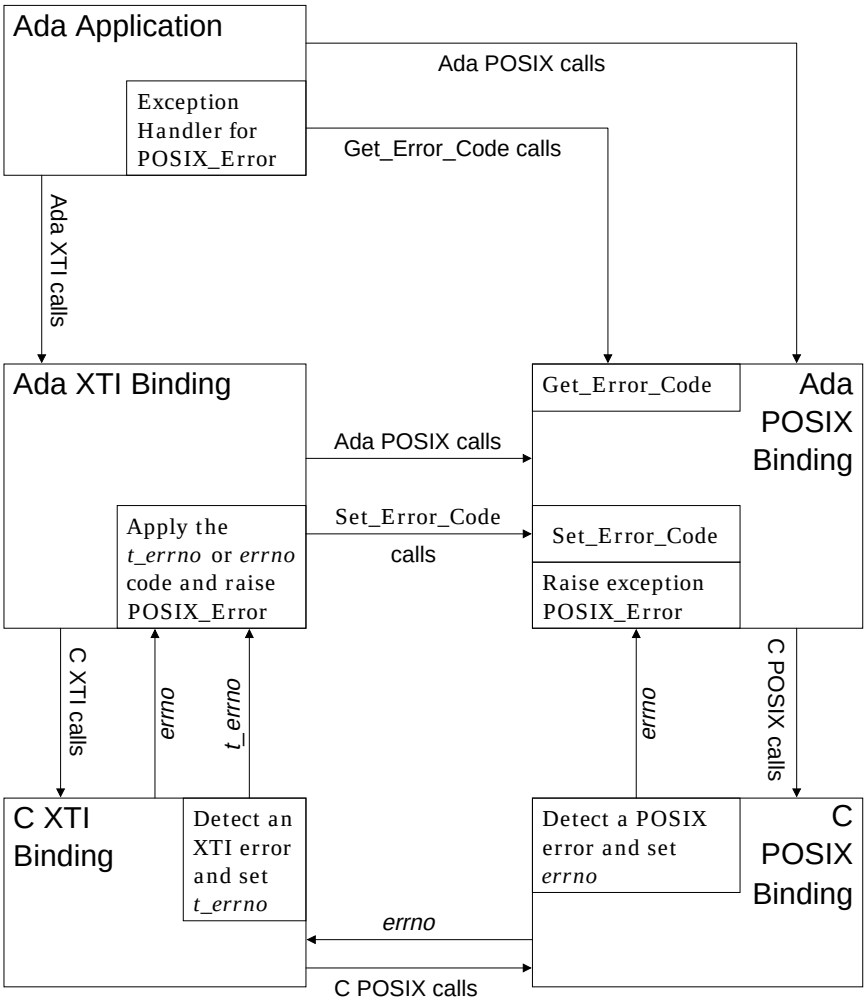

Figure B.1 – *t_errno* **and** *errno*

— Data buffers are implemented with a raw address of type `System.Address` and a length of type `POSIX.IO_Count`. This approach may risk storage corruption. However, to provide a more flexible and useful interface this compromise has been made. Due to Ada 95 language restrictions, parameters with unconstrained types cannot be passed via access parameters. The alternative, of using private types specific to this standard for this kind of buffer, was deemed too inflexible and inefficient for the calling application. If required, applications can build a safe interface to the binding to perform such bounds checks.

— Pointer types are defined for certain objects using Ada 95 general access types. General access types are used whenever a persistent reference to a private type must be deposited into another object.

— Subprogram parameters that have an **`out`** role and are private types are passed as **`in out`** to convey the responsibility of the application to allocate storage and prepare certain attibutes of the object prior to the call.

B.19 Detailed Network Interface - Socket

B.19.1 Socket Addresses

The C-language data structures for socket addresses in P1003.1g are protocol-specific address types that are cast to a socket address type for use with the sockets function calls. In this Ada binding, socket addresses are mapped to the tagged type `Socket_Address`. A similar approach is used for `XTI_Address`. The protocol-specific instances of socket addresses are detailed in D.1.

In Draft 4 of POSIX.5c, due to several general objections to dependence on Ada 95 without an Ada 83 fallback approach and several additional objections to the specific applications of tagged types the tagged type representations of network addresses were replaced with private pointer types. A general socket address type is provided in the base package for use with the base package operations, and protocol-specific address types are provide in the child packages (along with functions to convert the pointers between the general and protocol-specific specific network address types).

B.19.2 Endpoints and Sockets

Sockets and XTI endpoints have the type `POSIX_IO.File_Descriptor`. An alternate approach would be to have the sockets and XTI routines use subtypes of `File_Descriptor`. For example:

```
subtype Socket_Descriptor is POSIX_IO.File_Descriptor;
subtype XTI_Endpoint      is POSIX_IO.File_Descriptor;
```

This change was rejected as having too much impact to other packages in this standard that operate on file descriptors.

B.19.3 Socket Options

Two methods were proposed for getting and setting socket options. The current binding uses separate "`Get_`" and "`Set_`" subprograms for each option. An alternate approach was proposed, using a single pair of overloaded subprograms for getting and setting all the options and a record type containing option values. This second approach was rejected for the following reasons:

— The package implementation requires more code with this method to allocate data structures, copy data, and so on. (The application would require similar extra overhead.)

— An implementation may not add to a record type, since it may break code written for the original record.

— POSIX.5b has no visible record types.

— Also, in this standard it was deemed desirable to be able to verify that only standard packages are used in an application by checking **`with`** clauses. This could not be verified (easily) using an extensible record type.

B.19.4 Package POSIX.Sockets

B.19.4.1 Create a Pair of Connected Sockets

`Create_Pair` is typically used to create a pair of connected sockets before a process creation operation (similar to `Pipe`). A POSIX.5c ballot comment was received voicing concerns about portability issues related to calling the unsafe process primitive `Fork`. The ballot resolution group felt that the discussion of process creation using either the `Process_Template` approach or `Fork` and `Exec` was complete. No new portability issues are introduced by POSIX.5c. In particular, the safe way to create a new process connected to its parent by a socket pair is for the parent to create the socket pair and then call `Start_Process` to create the child process. The "`Set_File_Action_`" operations on the `Process_Template` permit the parent process to control the correspondence of the sockets with file descriptors of the child process.

B.19.4.2 Get and Set Options on Sockets

The *level* parameter in the C-language binding for the *getsockopt()* and *setsockopt()* functions is not a qualifier to the option. Rather, the *level*/*optname* pair itself specifies the option. This is necessary in C because *optname* is an integer; therefore, some means of partitioning the *optname* integer space is required. In the Ada binding, where each option has its own "`Get_`" and "`Set_`" subprograms, no parameter is needed for the level. Therefore all the "`Get_`" and "`Set_`" options calls in package `POSIX.Sockets` occur at the socket level. Lower level protocol options are specified in D.1.

B.19.4.3 Get Socket Type

Although it is part of the C-language API for *getsockopt()*, the Ada function `Get_Socket_Type` has been moved from the discussion of socket options to a separate subclause. Putting the description of this function at the same level as `Get_Socket_Name` and `Get_Peer_Name` is more consistent with the functional approach of this standard.

B.19.4.4 Receive Data From a Socket

The C function *recvfrom()* is named for historical reasons and also because function names cannot be overloaded in C. "Receive From" is a misnomer; it suggests "receive from this peer," but it really means "receive, and tell me the identity of the peer." Therefore, the Ada binding simply overloads the `Receive` procedure with an optional **`out`** parameter called `From`.

B.19.4.5 Send Data Over a Socket

Similarly, the C function *sendto()* is bound to the overloaded `Send` procedure with an optional **`in`** parameter called `To`.

B.20 Network Support Functions

The network support section found in P1003.1g has been removed from this standard. The functions that it provided have been allocated to other caluses of this standard as follows:

— 7.2.1 - In P1003.1g this subclause contains Internet-specific information. It was decided that all Internet specific information will be moved to D.1 of this standard.

— 7.2.3 - The function *getaddrinfo()* of P1003.1g maps to `Get_Socket_Address_Info`, which has been moved to a subclause in Section 18 since it relates to sockets only. This new subclause is preceded with text stating that this subclause must have the Network Management option support.

— 7.3 - These functions in Section 7 of P1003.1g are defined as obsolescent. Therefore, they were removed from the Ada binding.

— 7.4 and 7.5 - These subclauses contains Internet-specific information. It was agreed that all Internet specific information will be moved to D.1 of this standard.

— 7.6 and 7.7 - These functions in Section 7 of P1003.1g are defined as obsolescent. Therefore, they were removed from the Ada binding.

— 7.8 - These functions are traditionally used to convert Internet addresses and port numbers between host and network byte order (*htonl()*, *htons()*, *ntohl()*, *ntohs()*). It was decided to move the responsibility for any such conversion to the implementation of the binding. Text is added specifying that all Internet address and port number objects are in host byte order and that any required conversions shall be done automatically by the implementation of the binding.

Since the issue of big-endian *versus* little-endian storage of integer types may still be a concern for network data buffers, general purpose types and functions were added to package `POSIX` to support these conversions.

— 7.9 - This subclause contains Internet specific information. It was agreed that all Internet specific information will be moved to D.1 of this standard.

In the C interface, the network support functions that return the `Network_Info` and `Protocol_Info` objects are considered unsafe due to storage allocation performed by these services. A `Storage` parameter is provided on these functions to permit the calling application to allocate this storage. However, due to limitations in the operating system services underlying this binding, no guarantees about the behavior of these objects related to reentrancy or multitasking safety are made. Informative notes have been added to these sections with warnings about premature deallocation of this storage and limits to multitasking safety.

NOTE: Although multitasking-safe versions of these services are becoming available on some commercial operating systems, no standard approach currently exists. It is expected that a future revision of this standard may address this issue.

B.21 Protocol Mappings Annex

The protocol-specific items were originally documented in the body of the standard as a subclause under each function. However, to remain consistent with P1003.1g, the protocol mappings were removed from the body of the document and placed in a normative annex. Also for consistency with P1003.1g, IBM's System Network Architecture (SNA) was not included in the protocol mappings.

However, Annex D is not organized using the same structure as P1003.1g. The latter is organized by protocol with interface-specific information listed as subsections. The

developers of this standard deemed that this organization does not lend itself to easily adding protocols because the added protocol sections would be intermixed within existing sections. Instead, the Annex D of this standard is organized by interface (Sockets and XTI) with protocol specific information contained within the interface sections. This organization allows for the addition of a protocol without changing the original structure of the document. c

Annex C
(informative)
Ada/C Cross-References

This annex describes the relationship between the Ada-language names defined in this standard and the C-language names defined in the base standards.

C.1 Ada-to-C Cross-Reference

This clause lists the C-language name or names that correspond most closely to each Ada-language name defined by this standard. It is divided into subclauses according to the packages in which the Ada names are declared. Ada names for which there are no corresponding C names are not listed.

Package Ada_Task_Identifiation (16.1)

This package is specific to the Ada language.

Package Ada_Streams (2.7)

This package is specific to the Ada language.

Package POSIX (2.4)

Ada Name	C Name	
Address_In_Use	TADDRBUSY	
Address_In_Use	EADDRINUSE	
Address_Not_Available	EADDRNOTAVAIL	
Already_Awaiting_Connection	EALREADY	c
Argument_List_Maxima'Last	ARG_MAX	
Argument_List_Too_Long	E2BIG	
Bad_Address	EFAULT	
Bad_File_Descriptor	EBADF	
Bad_Message	EBADMSG	
Broken_Pipe	EPIPE	
Buffer_Not_Large_Enough	TBUFOVFLW	c
Change_Owner_Restriction	_POSIX_CHOWN_RESTRICTED	
Child_Processes_Maxima'Last	CHILD_MAX	
Communications_Provider_Mismatch	TPROVMISMATCH	
Connection_Aborted	ECONNABORTED	
Connection_Refused	ECONNREFUSED	
Connection_Reset	ECONNRESET	
Could_Not_Allocate_Address	TNOADDR	c
Directory_Not_Empty	ENOTEMPTY	
Domain_Error	EDOM	
Endpoint_Queue_Full	TQFULL	
Endpoint_Queue_Length_Is_Zero	TBADQLEN	
Event_Requires_Attention	TLOOK	c

Exec_Format_Error . . . . . ENOEXEC
File_Exists . . . . . EEXIST
File_Too_Large . . . . . EFBIG
Filename_Limit_Maxima'Last . . . . . NAME_MAX
Filename_Too_Long . . . . . ENAMETOOLONG
Filename_Truncation . . . . . _POSIX_NO_TRUNC
Flow_Control_Error . . . . . TFLOW | c
Get_Error_Code . . . . . *errno*
Groups_Maxima'First . . . . . NGROUPS_MAX
Host_Down . . . . . EHOSTDOWN
Host_To_Network_Byte_Order . . . . . *htonl()*
Host_To_Network_Byte_Order . . . . . *htons()*
Host_Unreachable . . . . . EHOSTUNREACH
Illegal_Data_Range . . . . . TBADDATA | c
Image . . . . . *perror()*
Improper_Link . . . . . EXDEV
Inappropriate_IO_Control_Operation . . . . . ENOTTY
Incorrect_Address_Type . . . . . EAFNOSUPPORT
Incorrect_Address_Format . . . . . TBADADDR
Incorrect_Or_Illegal_Option . . . . . TBADOPT
Incorrect_Surrogate_Queue_Length . . . . . TRESQLEN | c
Input_Line_Limit_Maxima'Last . . . . . MAX_CANON
Input_Output_Error . . . . . EIO
Input_Queue_Limit_Maxima'Last . . . . . MAX_INPUT
Insufficient_Permission . . . . . TACCES | c
Interrupted_Operation . . . . . EINTR
Invalid_Argument . . . . . EINVAL
Invalid_Communications_Provider . . . . . TBADNAME
Invalid_File_Descriptor . . . . . TBADF
Invalid_Flag . . . . . TBADFLAG
Invalid_Flags . . . . . EAI_BADFLAGS | c
Invalid_Seek . . . . . ESPIPE
Invalid_Sequence_Number . . . . . TBADSEQ | c
Is_A_Directory . . . . . EISDIR
Is_Already_Connected . . . . . EISCONN | c
Job_Control_Support . . . . . _POSIX_JOB_CONTROL
Link_Limit_Maxima'Last . . . . . LINK_MAX
Machine . . . . . *uname(), machine*
Memory_Allocation_Failed . . . . . EAI_MEMORY
Name_Failed . . . . . EAI_FAIL
Name_Not_Known . . . . . EAI_NONAME
Network_Down . . . . . ENETDOWN
Network_Reset . . . . . ENETRESET
Network_To_Host_Byte_Order . . . . . *ntohl()*
Network_To_Host_Byte_Order . . . . . *ntohs()*
Network_Unreachable . . . . . ENETUNREACH | c

No_Address_For_Name	EAI_NODATA	
No_Buffer_Space	ENOBUFS	c
Message_Too_Long	EMSGSIZE	
No_Child_Process	ECHILD	
No_Data_Available	TNODATA	
No_Disconnect_Indication_On_Endpoint	TNODIS	c
No_Locks_Available	ENOLCK	
No_Orderly_Release_Indication_On_Endpoint	TNOREL	c
No_Space_Left_On_Device	ENOSPC	
No_Such_Device_Or_Address	ENXIO	
No_Such_File_Or_Directory	ENOENT	
No_Such_Operation_On_Device	ENODEV	
No_Such_Process	ESRCH	
No_Unit_Data_Error_On_Endpoint	TNOUDERR	c
Node_Name	*uname(), nodename*	
Not_A_Directory	ENOTDIR	
Not_A_Socket	ENOTSOCK	
Not_Connected	ENOTCONN	c
Not_Enough_Space	ENOMEM	
Open_Files_Maxima'Last	OPEN_MAX	
Operation_Canceled	ECANCELED	
Operation_In_Progress	EINPROGRESS	
Operation_Not_Implemented	ENOSYS	
Operation_Not_Permitted	EPERM	
Operation_Not_Supported	ENOTSUP	
Operation_Not_Valid_For_State	TOUTSTATE	
Option_Not_Supported	EOPNOTSUPP	
Outstanding_Connection_Indications	TINDOUT	c
Pathname_Limit_Maxima'Last	PATH_MAX	
Permission_Denied	EACCES	
Pipe_Limit_Maxima'Last	PIPE_BUF	
Portable_Argument_List_Maximum	_POSIX_ARG_MAX	
Portable_Child_Processes_Maximum	_POSIX_CHILD_MAX	
Portable_Filename_Limit_Maximum	_POSIX_NAME_MAX	
Portable_Groups_Maximum	_POSIX_NGROUPS_MAX	
Portable_Input_Line_Limit_Maximum	_POSIX_MAX_CANON	
Portable_Input_Queue_Limit_Maximum	_POSIX_MAX_INPUT	
Portable_Link_Limit_Maximum	_POSIX_LINK_MAX	
Portable_Open_Files_Maximum	_POSIX_OPEN_MAX	
Portable_Pathname_Limit_Maximum	_POSIX_PATH_MAX	
Portable_Pipe_Limit_Maximum	_POSIX_PIPE_BUF	
Portable_Stream_Maximum	_POSIX_STREAM_MAX	
Portable_Time_Zone_String_Maximum	_POSIX_TZNAME_MAX	
POSIX_Version	_POSIX_VERSION	
Protocol_Error	TPROTO	
Protocol_Not_Supported	EPROTONOSUPPORT	c

Package POSIX_Asynchronous_IO (6.3)

Ada Name	C Name

Synchronize_Data . . . . . *aio_fsync()*, O_DSYNC
Synchronize_File . . . . . *aio_fsync()*, O_SYNC
Write . . . . . *aio_write()*, LIO_WRITE

Package POSIX_Calendar (4.4)

The package POSIX_Calendar is Ada-specific.

Package POSIX_Condition_Variables (11.3)

Ada Name	C Name
Attributes	*pthread_condattr_t*
Broadcast	*pthread_cond_broadcast()*
Condition_Descriptor	*pthread_cond_t*
Finalize	*pthread_cond_destroy()*
Finalize	*pthread_condattr_destroy()*
Get_Process_Shared	*pthread_condattr_getpshared()*
Initialize	*pthread_cond_init()*
Initialize	*pthread_condattr_init()*
Set_Process_Shared	*pthread_condattr_setpshared()*
Signal	*pthread_cond_signal()*
Timed_Wait	*pthread_cond_timedwait()*
Wait	*pthread_cond_wait()*

Package POSIX_Configurable_File_Limits (5.4)

Ada Name	C Name
Asynchronous_IO_Is_Supported	*fpathconf()*, _PC_ASYNC_IO
Asynchronous_IO_Is_Supported	*pathconf()*, _PC_ASYNC_IO
Change_Owner_Is_Restricted	*fpathconf()*, _PC_CHOWN_RESTRICTED
Change_Owner_Is_Restricted	*pathconf()*, _PC_CHOWN_RESTRICTED
Filename_Is_Limited	*fpathconf()*, _PC_NAME_MAX
Filename_Is_Limited	*pathconf()*, _PC_NAME_MAX
Filename_Is_Truncated	*fpathconf()*, _PC_NO_TRUNC
Filename_Is_Truncated	*pathconf()*, _PC_NO_TRUNC
Filename_Limit	*fpathconf()*, _PC_NAME_MAX
Filename_Limit	*pathconf()*, _PC_NAME_MAX
Input_Line_Is_Limited	*fpathconf()*, _PC_MAX_CANON
Input_Line_Is_Limited	*pathconf()*, _PC_MAX_CANON
Input_Line_Limit	*fpathconf()*, _PC_MAX_CANON
Input_Line_Limit	*pathconf()*, _PC_MAX_CANON
Input_Queue_Is_Limited	*fpathconf()*, _PC_MAX_INPUT
Input_Queue_Is_Limited	*pathconf()*, _PC_MAX_INPUT
Input_Queue_Limit	*fpathconf()*, _PC_MAX_INPUT
Input_Queue_Limit	*pathconf()*, _PC_MAX_INPUT
Links_Are_Limited	*fpathconf()*, _PC_LINK_MAX
Links_Are_Limited	*pathconf()*, _PC_LINK_MAX
Link_Is_Limited	*fpathconf()*, _PC_LINK_MAX

Package POSIX_Configurable_System_Limits (4.5)

Ada Name **C Name**

OSI_Connectionless_Is_Supported . . . . . *sysconf(),* _SC_PII_OSI_CLTS
OSI_Connection_Is_Supported . . . . . *sysconf(),* _SC_PII_OSI_COTS
OSI_Minimal_Is_Supported . . . . . *sysconf(),* _SC_PII_OSI_M
Page_Size . . . . . *sysconf(),* _SC_PAGESIZE
Poll_Is_Supported . . . . . *sysconf(),* _SC_POLL
Prioritized_IO_Is_Supported . . . . . *sysconf(),* _SC_PRIORITIZED_IO
Mutex_Priority_Inheritance_Is_Supported . . . . *sysconf(),* _SC_THREAD_PRIO_INHERIT
Mutex_Priority_Ceiling_Is_Supported . . . . . *sysconf(),* _SC_THREAD_PRIO_PROTECT
Priority_Process_Scheduling_Is_Supported . . . *sysconf(),* _SC_PRIORITY_SCHEDULING
Queued_Signals_Maximum . . . . . *sysconf(),* _SC_SIGQUEUE_MAX
Realtime_Signals_Are_Supported . . . . . *sysconf(),* _SC_REALTIME_SIGNALS
Realtime_Signals_Maximum . . . . . *sysconf(),* _SC_RTSIG_MAX
Saved_IDs_Supported . . . . . *sysconf(),* _SC_SAVED_IDS
Saved_IDs_Are_Supported . . . . . *sysconf(),* _SC_SAVED_IDS
Select_Is_Supported . . . . . *sysconf(),* _SC_SELECT
Semaphores_Are_Supported . . . . . *sysconf(),* _SC_SEMAPHORES
Semaphores_Maximum . . . . . *sysconf(),* _SC_SEM_NSEMS_MAX
Semaphores_Value_Maximum . . . . . *sysconf(),* _SC_SEM_VALUE_MAX
Shared_Memory_Objects_Are_Supported . . . . *sysconf(),* _SC_SHARED_MEMORY_OBJECTS
Socket_IO_Vector_Maximum . . . . . *sysconf(),* _SC_UIO_MAXIOV
Sockets_DNI_Is_Supported . . . . . *sysconf(),* _SC_PII_SOCKET
Streams_Maximum . . . . . *sysconf(),* _SC_STREAM_MAX
Synchronized_IO_Is_Supported . . . . . *sysconf(),* _SC_SYNCHRONIZED_IO
System_POSIX_Version . . . . . *sysconf(),* _SC_VERSION
Process_Shared_Is_Supported . . . . . *sysconf(),* _SC_THREAD_PROCESS_SHARED
Time_Zone_String_Maximum . . . . . *sysconf(),* _SC_TZNAME_MAX
Timer_Overruns_Maximum . . . . . *sysconf(),* _SC_DELAYTIMER_MAX
Timers_Are_Supported . . . . . *sysconf(),* _SC_TIMERS
Timers_Maximum . . . . . *sysconf(),* _SC_TIMER_MAX
XTI_DNI_Is_Supported . . . . . *sysconf(),* _SC_PII_XTI
XTI_IO_Vector_Maximum . . . . . *sysconf(),* _SC_T_IOV_MAX

Package POSIX_Event_Management (19.1)

Ada Name	C Name
Add	FD_SET
File_Descriptor_Set	*fdset*
File_Not_Open	POLLNVAL
In_Set	FD_ISSET
Initialize_File_Descriptor_Set	FD_ZERO
Poll	*poll()*
Poll_Error	POLLERR
Poll_FD	*pollfd*
Read_High	POLLPRI
Read_Normal	POLLRDNORM
Read_Not_High	POLLIN

Read_Priority	POLLRDBAND
Remove	FD_CLR
Select_File	*select()*
Write_Normal	POLLOUT
Write_Normal	POLLWRNORM
Write_Priority	POLLWRBAND

c

Package POSIX_File_Locking (6.2)

Ada Name	C Name
File_Lock	flock
Get_Lock	*fcntl()*, F_GETLK
Read_Lock	F_RDLCK
Set_Lock	*fcntl()*, F_SETLK
Unlock	F_UNLCK
Wait_To_Set_Lock	*fcntl()*, F_SETLKW
Write_Lock	F_WRLCK

Package POSIX_File_Status (5.3)

Ada Name	C Name
Device_ID_Of	*st_dev*
Device_ID	*dev_t*
File_ID_Of	*st_ino*
File_ID	*ino_t*
Get_File_Status	*fstat()*
Get_File_Status	*stat()*
Group_Of	*st_gid*
Is_Block_Special_File	S_ISBLK
Is_Character_Special_File	S_ISCHR
Is_Directory	S_ISDIR
Is_FIFO	S_ISFIFO
Is_Regular_File	S_ISREG
Is_Message_Queue	S_TYPEISMQ
Is_Semaphore	S_TYPEISSEM
Is_Shared_Memory	S_TYPEISSHM
Is_Socket	S_ISSOCK
Last_Access_Time_Of	*st_atime*
Last_Modification_Time_Of	*st_mtime*
Last_Status_Change_Time_Of	*st_ctime*
Link_Count_Of	*st_nlink*
Links	*nlink_t*
Owner_Of	*st_uid*
Permission_Set_Of	*st_mode*
Size_Of	*st_size*
Status	*stat*

c

Package POSIX_Files (5.2)

Ada Name	C Name
Accessibility	*access()*
Change_Owner_And_Group	*chown()*
Change_Permissions	*chmod()*
Create_Directory	*mkdir()*
Create_FIFO	*mkfifo()*
Directory_Entry	*dirent*
Execute_Ok	X_OK
Existence	*access()*, F_OK
Filename_Of	*d_name()*
For_Every_Directory_Entry	*closedir()*
For_Every_Directory_Entry	*opendir()*
For_Every_Directory_Entry	*readdir()*
For_Every_Directory_Entry	*rewinddir()*
Is_Accessible	*access()*
Is_Block_Special_File	*stat()*, S_ISBLK
Is_Character_Special_File	*stat()*, S_ISCHR
Is_Directory	*stat()*, S_ISDIR
Is_FIFO	*stat()*, S_ISFIFO
Is_File_Present	*access()*
Is_File	*stat()*, S_ISREG
Is_Message_Queue	*stat()*, S_TYPEISMQ
Is_Semaphore	*stat()*, S_TYPEISSEM
Is_Shared_Memory	*stat()*, S_TYPEISSHM
Is_Socket	*stat()*, S_ISSOCK
Link	*link()*
Read_Ok	R_OK
Remove_Directory	*rmdir()*
Rename	*rename()*
Set_File_Times	*utime()*
Unlink	*unlink()*
Write_Ok	W_OK

Package POSIX_Generic_Shared_Memory (12.5)

Ada Name	C Name
Open_And_Map_Shared_Memory	*ftruncate()*, *shm_open()*
Open_And_Map_Shared_Memory	*mmap()*, *shm_open()*
Open_Or_Create_And_Map_Shared_Memory	*ftruncate()*, *shm_open()*
Open_Or_Create_And_Map_Shared_Memory	*mmap()*, *shm_open()*
Unmap_and_Close_Shared_Memory	*close()*, *munmap()*
Lock_Shared_Memory	*mlock()*
Unlock_Shared_Memory	*munlock()*

Package POSIX_Group_Database (9.2)

Ada Name	C Name
Get_Group_Database_Item	*getgrgid()*
Get_Group_Database_Item	*getgrnam()*
Group_Database_Item	*group*
Group_ID_List_Of	*gr_mem*
Group_ID_Of	*gr_gid*
Group_Name_Of	*gr_name*

Package POSIX_IO (6.1)

Ada Name	C Name
Append	O_APPEND
Change_Permissions	*fchmod()*
Close	*close()*
Create_Pipe	*pipe()*
Data_Synchronized	O_DSYNC
Duplicate_and_Close	*dup2()*
Duplicate	*dup()*
Duplicate	*fcntl()*, F_DUPFD
Exclusive	O_EXCL
File_Mode	O_RDONLY
File_Mode	O_RDWR
File_Mode	O_WRONLY
File_Position	*lseek()*
File_Size	*lseek()*
File_Synchronized	O_SYNC
From_Beginning	SEEK_SET
From_Current_Position	SEEK_CUR
From_End_Of_File	SEEK_END
Generic_Read	read
Generic_Write	write
Get_Close_On_Exec	*fcntl()*, F_GETFD, FD_CLOEXEC
Get_File_Control	*fcntl()*, F_GETFL
Get_Owner	*fcntl()*, F_SETOWN
Get_Terminal_Name	*ttyname()*
IO_Offset	*off_t*
IO_Offset	*size_t*
IO_Offset	*ssize_t*
Is_A_Terminal	*isatty()*
Non_Blocking	O_NONBLOCK
Not_Controlling_Terminal	O_NOCTTY
Open_Or_Create	*open()*, O_CREAT
Open_Or_Create	creat
Open	*open()*
Position	SEEK_CUR
Position	SEEK_END

c

Position	SEEK_SET
Read_Only	O_RDONLY
Read_Synchronized	O_DSYNC
Read_Write	O_RDWR
Read	*read()*
Seek	*lseek()*
Set_Close_On_Exec	*fcntl()*, F_SETFD, FD_CLOEXEC
Set_File_Control	*fcntl()*, F_SETFL
Set_Socket_Group_Owner	*fcntl()*, F_SETOWN
Set_Socket_Process_Owner	*fcntl()*, F_SETOWN
Signal_When_Socket_Ready	O_ASYNC
Synchronize_Data	*fdatasync()*
Synchronize_File	*fsync()*
Truncate	O_TRUNC
Truncate_File	*ftruncate()*
Write_Only	O_WRONLY
Write	*write()*

C

Package POSIX_Limits (2.6)

Ada Name	C Name
Portable_Argument_List_Maximum	_POSIX_ARG_MAX
Portable_Asynchronous_IO_Maximum	_POSIX_AIO_MAX
Portable_Child_Processes_Maximum	_POSIX_CHILD_MAX
Portable_Clock_Resolution_Minimum	_POSIX_CLOCKRES_MIN
Portable_FD_Set_Maximum	_POSIX_FD_SETSIZE
Portable_Filename_Maximum	_POSIX_NAME_MAX
Portable_Groups_Maximum	_POSIX_NGROUPS_MAX
Portable_Input_Line_Maximum	_POSIX_MAX_CANON
Portable_Input_Queue_Maximum	_POSIX_MAX_INPUT
Portable_Links_Maximum	_POSIX_LINK_MAX
Portable_List_IO_Maximum	_POSIX_AIO_LISTIO_MAX
Portable_Message_Priority_Maximum	_POSIX_MQ_PRIO_MAX
Portable_Open_Files_Maximum	_POSIX_OPEN_MAX
Portable_Open_Message_Queues_Maximum	_POSIX_MQ_OPEN_MAX
Portable_Pathname_Maximum	_POSIX_PATH_MAX
Portable_Pipe_Length_Maximum	_POSIX_PIPE_BUF
Portable_Queued_Signals_Maximum	_POSIX_SIGQUEUE_MAX
Portable_Realtime_Signals_Maximum	_POSIX_RTSIG_MAX
Portable_Semaphores_Maximum	_POSIX_SEM_NSEMS_MAX
Portable_Semaphores_Value_Maximum	_POSIX_SEM_VALUE_MAX
Portable_Socket_Buffer_Maximum	_POSIX_HIWAT
Portable_Socket_Connection_Maximum	_POSIX_QLIMIT
Portable_Socket_IO_Vector_Maximum	_POSIX_UIO_MAXIOV
Portable_Streams_Maximum	_POSIX_STREAM_MAX
Portable_Timer_Overruns_Maximum	_POSIX_DELAYTIMER_MAX

C

Portable_Timers_Maximum	_POSIX_TIMER_MAX
Portable_Time_Zone_String_Maximum	_POSIX_TZNAME_MAX
Argument_List_Maxima'Last	ARG_MAX
Asynchronous_IO_Maxima'Last	AIO_MAX
Asynchronous_IO_Priority_Delta_Maxima'Last	AIO_PRIO_DELTA_MAX
Child_Processes_Maxima'Last	CHILD_MAX
FD_Set_Maxima'Last	FD_SETSIZE c
Filename_Maxima'Last	NAME_MAX
Groups_Maxima'First	NGROUPS_MAX
Input_Line_Maxima'Last	MAX_CANON
Input_Queue_Maxima'Last	MAX_INPUT
Links_Maxima'Last	LINK_MAX
List_IO_Maxima'Last	AIO_LISTIO_MAX
Message_Priority_Maxima'Last	MQ_PRIO_MAX
Open_Files_Maxima'Last	OPEN_MAX
Open_Message_Queues_Maxima'Last	MQ_OPEN_MAX
Page_Size_Range'Last	PAGESIZE
Pathname_Maxima'Last	PATH_MAX
Pipe_Length_Maxima'Last	PIPE_BUF
Queued_Signals_Maxima'Last	SIGQUEUE_MAX
Realtime_Signals_Maxima'Last	RTSIG_MAX
Semaphores_Maxima'Last	SEM_NSEMS_MAX
Semaphores_Value_Maxima'Last	SEM_VALUE_MAX
Socket_Buffer_Maxima'Last	SOCK_MAXBUF
Socket_IO_Vector_Maxima'Last	UIO_MAXIOV c
Streams_Maxima'Last	STREAM_MAX
Timer_Overruns_Maxima'Last	DELAYTIMER_MAX
Timers_Maxima'Last	TIMER_MAX
Time_Zone_String_Maxima'Last	TZNAME_MAX
XTI_IO_Vector_Maxima'Last	T_IOV_MAX c

Package POSIX_Memory_Locking (12.1)

Ada Name	C Name
Current_Pages	MCL_CURRENT
Future_Pages	MCL_FUTURE
Lock_All	*mlockall()*
Unlock_All	*munlockall()*

Package POSIX_Memory_Mapping (12.3)

Ada Name	C Name
Allow_Execute	PROT_EXEC
Allow_Read	PROT_READ
Allow_Write	PROT_WRITE
Change_Protection	*mprotect()*
Empty_Set	PROT_NONE

Exact_Address . . . . . MAP_FIXED
Map_Memory . . . . . *mmap()*
Map_Private . . . . . MAP_PRIVATE
Map_Shared . . . . . MAP_SHARED
Invalidate_Cached_Data . . . . . MS_INVALIDATE
Wait_For_Completion . . . . . MS_SYNC
Synchronize_Memory . . . . . *msync()*
Unmap_Memory . . . . . *munmap()*

Package POSIX_Memory_Range_Locking (12.2)

Ada Name	C Name
Lock_Range	*mlock()*
Unlock_Range	*munlock()*

Package POSIX_Message_Queues (15.1)

Ada Name	C Name
Close	*mq_close()*
Generic_Message_Queues.Receive	*mq_receive()*
Generic_Message_Queues.Send	*mq_send()*
Get_Attributes	*mq_getattr()*
Attributes	*mq_attr*
Message_Queue_Descriptor	*mqd_t*
Non_Blocking	O_NONBLOCK
Open_Or_Create	*mq_open()*, O_CREAT
Open	*mq_open()*
Receive	*mq_receive()*
Remove_Notify	*mq_notify()*
Request_Notify	*mq_notify()*
Send	*mq_send()*
Set_Attributes	*mq_setattr()*
Unlink_Message_Queue	*mq_unlink()*

Package POSIX_Mutexes (11.2)

Ada Name	C Name
Attributes	*pthread_mutexattr_t*
Finalize	*pthread_mutex_destroy()*
Finalize	*pthread_mutexattr_destroy()*
Get_Ceiling_Priority	*pthread_mutex_getprioceiling()*
Get_Ceiling_Priority	*pthread_mutexattr_getprioceiling()*
Get_Process_Shared	PTHREAD_PROCESS_PRIVATE
Get_Process_Shared	PTHREAD_PROCESS_SHARED
Get_Process_Shared	*pthread_mutexattr_getpshared()*
Get_Protocol	*pthread_mutexattr_getprotocol()*
Highest_Blocked_Task	PTHREAD_PRIO_INHERIT

Package POSIX_Options (2.5)

Ada Name	C Name	
Asynchronous_IO_Support	_POSIX_ASYNCHRONOUS_IO	
Change_Owner_Restriction	_POSIX_CHOWN_RESTRICTED	
Filename_Truncation	_POSIX_NO_TRUNC	
File_Synchronization_Support	_POSIX_FSYNC	
Internet_Datagram_Support	_POSIX_PII_INTERNET_DGRAM	c
Internet_Protocol_Support	_POSIX_PII_INTERNET	
Internet_Stream_Support	_POSIX_PII_INTERNET_STREAM	
ISO_OSI_Protocol_Support	_POSIX_PII_OSI	c
Job_Control_Support	_POSIX_JOB_CONTROL	
Memory_Mapped_Files_Support	_POSIX_MAPPED_FILES	
Memory_Range_Locking_Support	_POSIX_MEMLOCK_RANGE	
Memory_Locking_Support	_POSIX_MEMLOCK	
Memory_Protection_Support	_POSIX_MEMORY_PROTECTION	
Message_Queues_Support	_POSIX_MESSAGE_PASSING	
Mutex_Priority_Ceiling_Support	_POSIX_THREAD_PRIO_PROTECT	
Mutex_Priority_Inheritance_Support	_POSIX_THREAD_PRIO_INHERIT	
Network_Management_Support	_POSIX_PII_NET_SUPPORT	c
OSI_Connection_Support	_POSIX_PII_OSI_COTS	
OSI_Connectionless_Support	_POSIX_PII_OSI_CLTS	
OSI_Minimal_Support	_POSIX_PII_OSI_M	
Poll_Support	_POSIX_POLL	c
Prioritized_IO_Support	_POSIX_PRIORITIZED_IO	
Priority_Process_Scheduling_Support	_POSIX_PRIORITY_SCHEDULING	
Realtime_Signals_Support	_POSIX_REALTIME_SIGNALS	
Saved_IDs_Support	_POSIX_SAVED_IDS	
Select_Support	_POSIX_SELECT	c
Semaphores_Support	_POSIX_SEMAPHORES	
Shared_Memory_Objects_Support	_POSIX_SHARED_MEMORY_OBJECTS	
Sockets_DNI_Support	_POSIX_PII_SOCKET	c
Synchronized_IO_Support	_POSIX_SYNCHRONIZED_IO	

Process_Shared_Support	_POSIX_THREAD_PROCESS_SHARED
Timers_Support	_POSIX_TIMERS
XTI_DNI_Support	_POSIX_PII_XTI

c

Package POSIX_Page_Alignment (2.10)

This package is specific to this standard.

Package POSIX_Permissions (5.1)

Ada Name	C Name
Access_Permission_Set	*file permission bits*
Get_Allowed_Process_Permissions	*umask()*
Group_Execute	S_IXGRP
Group_Permission_Set	S_IRWXG
Group_Read	S_IRGRP
Group_Write	S_IWGRP
Others_Execute	S_IXOTH
Others_Permission_Set	S_IRWXO
Others_Read	S_IROTH
Others_Write	S_IWOTH
Owner_Execute	S_IXUSR
Owner_Permission_Set	S_IRWXU
Owner_Read	S_IRUSR
Owner_Write	S_IWUSR
Permission_Set	*mode_t*
Set_Allowed_Process_Permissions	*umask()*
Set_Group_ID_Set	S_ISGID
Set_Group_ID	S_ISGID
Set_User_ID_Set	S_ISUID
Set_User_ID	S_ISUID

Package POSIX_Process_Environment (4.3)

Ada Name	C Name
Argument_List	*argv*
Change_Working_Directory	*chdir()*
Copy_From_Current_Environment	*environ*
Copy_To_Current_Environment	*environ*
Environment_Value_Of	*getenv()*
Get_Working_Directory	*getcwd()*
Is_Environment_Variable	*getenv()*

Package POSIX_Process_Identification (4.1)

Ada Name	C Name
Create_Process_Group	*setpgid()*
Create_Session	*setsid()*

Package POSIX_Process_Primitives (3.1)

Ada Name **C Name**

Package POSIX_Process_Scheduling (13.2)

Ada Name **C Name**

Get_Scheduling_Parameters . . . . . *sched_getparam()*
Get_Scheduling_Policy . . . . . *sched_getscheduler()*
Sched_Other . . . . . SCHED_OTHER
Sched_RR . . . . . SCHED_RR
Scheduling_Parameters . . . . . *sched_param*
Set_Scheduling_Parameters . . . . . *sched_setparam()*
Set_Scheduling_Policy . . . . . *sched_setscheduler()*
Yield . . . . . *sched_yield()*

Package POSIX_Process_Times (4.2)

Ada Name	C Name
Descendants_System_CPU_Time_Of	*tms_cstime*
Descendants_User_CPU_Time_Of	*tms_cutime*
Elapsed_Real_Time	*times()*
Get_Process_Times	*times()*
Process_Times	*tms*
System_CPU_Time_Of	*tms_stime*
Tick_Count	*clock_t*
Ticks_Per_Second	CLK_TCK
Ticks_Per_Second	*sysconf()*, _SC_CLK_TCK
User_CPU_Time_Of	*tms_utime*

Package POSIX_Semaphores (11.1)

Ada Name	C Name
Close	*sem_close()*
Finalize	*sem_destroy()*
Get_Value	*sem_getvalue()*
Initialize	*sem_init()*
Open_Or_Create	*sem_open()*, O_CREAT
Open	*sem_open()*
Post	*sem_post()*
Semaphore	*sem_t*
Try_Wait	*sem_trywait()*
Unlink_Semaphore	*sem_unlink()*
Wait	*sem_wait()*

Package POSIX_Shared_Memory_Objects (12.4)

Ada Name	C Name
Open_Or_Create_Shared_Memory	*shm_open()*
Open_Shared_Memory	*shm_open()*
Unlink_Shared_Memory	*shm_unlink()*

Package POSIX_Signals (3.3)

Ada Name	C Name

Signal_Set	*sigset_t*
Signal_Source	SI_ASYNCIO
Signal_Source	SI_MESGQ
Signal_Source	SI_QUEUE
Signal_Source	SI_TIMER
Signal_Source	SI_USER
Signal_Stop	SIGSTOP
Signal_Terminal_Input	SIGTTIN
Signal_Terminal_Output	SIGTTOU
Signal_Terminal_Stop	SIGTSTP
Signal_Terminate	SIGTERM
Signal_User_1	SIGUSR1
Signal_User_2	SIGUSR2
Signal_Data	*sigval*
Stopped_Child_Signal_Enabled	*sigaction()*, SA_NOCLDSTOP
Unblock_Signals	*sigprocmask()*
Unignore_Signal	*sigaction()*, SIG_IGN

Package POSIX_Sockets (18.4)

Ada Name	C Name
Accept_Connection	*accept()*
Ancillary_Data_Lost	MSG_CTRUNC
Bind	*bind()*
Connect	*connect()*
Connection_Queue_Length_Maximum	SOMAXCONN
Control_Message	*cmsghdr*
Create	*socket()*
Create_Pair	*socketpair()*
Datagram_Socket	SOCK_DGRAM
Do_Not_Route	MSG_DONTROUTE
End_Of_Message	MSG_EOR
Further_Receives_Disallowed	SHUT_RD
Further_Sends_And_Receives_Disallowed	SHUT_RDWR
Further_Sends_Disallowed	SHUT_WR
Get_Canonical_Name	AI_CANONNAME
Get_Peer_Name	*getpeername()*
Get_Socket_Address_Info	*getaddrinfo()*
Get_Socket_Broadcast	*getsockopt()*, SO_BROADCAST
Get_Socket_Debugging	*getsockopt()*, SO_DEBUG
Get_Socket_Error_Status	*getsockopt()*, SO_ERROR
Get_Socket_Keep_Alive	*getsockopt()*, SO_KEEPALIVE
Get_Socket_Linger_Time	*getsockopt()*, SO_LINGER
Get_Socket_Linger_Time	*linger*
Get_Socket_Name	*getsockname()*
Get_Socket_No_Routing	*getsockopt()*, SO_DONTROUTE

C

Package POSIX_Sockets_Internet (D.1.3)

Ada Name	C Name
Broadcast_Internet_Address	INADDR_BROADCAST
Close_Network_Database_Connection	*endnetent()*
Close_Protocol_Database_Connection	*endprotoent()*
Get_Header_Included	*getsockopt()*, IP_HDRINCL
Get_Initial_Time_To_Live	*getsockopt()*, IP_TTL
Get_IP_Header_Options	*getsockopt()*, IP_OPTIONS
Get_Keep_Alive_Interval	*getsockopt()*, TCP_KEEPALIVE
Get_Network_Info_By_Address	*getnetbyaddr()*
Get_Network_Info_By_Name	*getnetbyname()*
Get_No_Delay	*getsockopt()*, TCP_NODELAY
Get_Protocol_Info_By_Name	*getprotobyname()*
Get_Protocol_Info_By_Number	*getprotobynumber()*
Get_Receive_Destination_Address	*getsockopt()*, IP_RECVDSTADDR
Get_Retransmit_Time_Maximum	*getsockopt()*, TCP_MAXRXT
Get_Segment_Size_Maximum	*getsockopt()*, TCP_MAXSEG
Get_Standardized_Urgent_Data	*getsockopt()*, TCP_STDURG
Get_Type_Of_Service	*getsockopt()*, IP_TOS
High_Reliability	IPTOS_RELIABILITY
High_Throughput	IPTOS_THROUGHPUT
ICMP	IPPROTO_ICMP
Internet_Address	*in_addr*
Internet_Address_To_String	*inet_ntoa()*
Internet_Port	*in_port_t*
Internet_Protocol	PF_INET
Internet_Socket_Address	*sockaddr_in*
IP_Options_Buffer	*ip_opts*
Is_Internet_Address	INADDR_NONE
Loopback_Internet_Address	INADDR_LOOPBACK
Low_Delay	IPTOS_LOWDELAY
Network_Info	*netent*
Open_Network_Database_Connection	*setnetent()*
Open_Protocol_Database_Connection	*setprotoent()*
Protocol_Info	*protoent*
Raw	IPPROTO_RAW
Set_Header_Included	*setsockopt()*, IP_HDRINCL
Set_Initial_Time_To_Live	*setsockopt()*, IP_TTL
Set_IP_Header_Options	*setsockopt()*, IP_OPTIONS

Package POSIX_Sockets_ISO (D.1.2)

Ada Name	C Name

Suppress_Error_PDUs	CLNP_NO_ERR
TP_Acknowledgment_Strategy	TPACK_EACH
TP_Acknowledgment_Strategy	TPACK_WINDOW
TP_Class_0	TP_CLASS_0
TP_Class_1	TP_CLASS_1
TP_Class_2	TP_CLASS_2
TP_Class_3	TP_CLASS_3
TP_Class_4	TP_CLASS_4
TP_Retransmit_Strategy	TPRX_EACH
TP_Retransmit_Strategy	TPRX_FASTSTART
TP_Retransmit_Strategy	TPRX_USE_CW
Transport_Level	SOL_TRANSPORT
Use_Congestion_Window	TPRX_USE_CW

Package POSIX_Sockets_Local (D.1.1)

Ada Name	C Name
Local_Protocol	PF_LOCAL
Local_Socket_Address	*sockaddr_un*

C

Package POSIX_Supplement_to_Ada_IO (8.2)

This package is specific to this standard.

Package POSIX_Terminal_Functions (7.2)

Ada Name	C Name
After_Output_and_Input	TCSAFLUSH
After_Output	TCSADRAIN
B0	B0
B110	B110
B1200	B1200
B134	B134
B150	B150
B1800	B1800
B19200	B19200
B200	B200
B2400	B2400
B300	B300
B38400	B38400
B4800	B4800
B50	B50
B600	B600
B75	B75
B9600	B9600
Baud_Rate	*speed_t*
Bits_Per_Character	CSIZE
Both	TCIOFLUSH

Perform_Output_Processing	OPOST
Quit_Char	VQUIT
Received_But_Not_Read	TCIFLUSH
Restart_Output	TCOON
Send_Break	*tcsendbreak()*
Send_Signal_For_BG_Output	TOSTOP
Send_Two_Stop_Bits	CSTOPB
Set_Process_Group_ID	*tcsetpgrp()*
Set_Terminal_Characteristics	*tcsetattr()*
Start_Char	VSTART
Stop_Char	VSTOP
Strip_Character	ISTRIP
Suspend_Char	VSUSP
Suspend_Output	TCOOFF
Terminal_Characteristics	*termios*
Terminal_Modes	*tcflag_t*
Transmit_Start	TCION
Transmit_Stop	TCIOFF
Written_But_Not_Transmitted	TCOFLUSH

Package POSIX_Timers (14.1)

Ada Name	C Name
Absolute_Timer	TIMER_ABSTIME
Arm_Timer	*timer_settime()*
Clock_ID	*clockid_t*
Clock_Realtime	CLOCK_REALTIME
Create_Timer	*timer_create()*
Delete_Timer	*timer_delete()*
Disarm_Timer	*timer_settime()*
Get_Resolution	*clock_getres()*
Get_Timer_Overruns	*timer_getoverrun()*
Get_Timer_State	*timer_gettime()*
Get_Time	*clock_gettime()*
Set_Time	*clock_settime()*
Timer_ID	*timer_t*

Package POSIX_Unsafe_Process_Primitives (3.2)

Ada Name	C Name
Exec_Search	*execlp()*
Exec_Search	*execvp()*
Exec	*execl()*
Exec	*execle()*
Exec	*execv()*
Exec	*execve()*
Fork	*fork()*

Package POSIX_User_Database (9.1)

Ada Name	C Name
Get_User_Database_Item	*getpwnam()*
Get_User_Database_Item	*getpwuid()*
Group_ID_Of	*pw_gid*
Initial_Directory_Of	*pw_dir*
Initial_Program_Of	*pw_shell*
User_Database_Item	*passwd*
User_ID_Of	*pw_uid*
User_Name_Of	*pw_name*

Package POSIX_XTI (17.4)

Ada Name	C Name
Accept_Connection	*t_accept()*
Acknowledge_Orderly_Release	*t_rcvrel()*
Acknowledge_Orderly_Release_With_Data	*t_rcvreldata()*
All_Options	T_ALLOPT
Bind	*t_bind()*
Check_Options	T_CHECK
Close	*t_close()*
Communications_Provider_Info	*t_info*
Confirm_Connection	*t_rcvconnect()*
Connect	*t_connect()*
Connect_Request_Received	T_LISTEN
Connect_Responsed_Received	T_CONNECT
Connection_Info	*t_call*
Connection_Mode	T_COTS
Connection_Mode_With_Orderly_Release	T_COTS_ORD
Connectionless_Mode	T_CLTS
Data_Transfer	T_DATAXFER
Disconnect_Request_Received	T_DISCONNECT
Enable_Debugging	XTI_DEBUG
Expedited_Data	T_EXPEDITED
Error_In_Previously_Sent_Datagram	T_UDERR
Expedited_Data_Received	T_EXDATA
Failure	T_FAILURE
Gather_And_Send_Data	*t_sndv()*
Gather_And_Send_Data_Unit	*t_sndvudata()*
Get_Current_Options	T_CURRENT
Get_Current_State	*t_getstate()*
Get_Default_Options	T_DEFAULT
Get_Info	*t_getinfo()*
Get_Next_Option	OPT_NEXTHDR
Get_Protocol_Address	*t_getprotaddr()*
Idle	T_IDLE

C

Package POSIX_XTI_Internet (D.2.3)

Ada Name **C Name**

Maximum_Segment_Size	TCP_MAXSEG
Network_Control	T_NETCONTROL
Network_Info	*netent*
No_Delay	TCP_NODELAY
Normal	T_NOTOS
Open_Network_Database_Connection	*setnetent()*
Open_Protocol_Database_Connection	*setprotoent()*
Permit_Broadcast	IP_BROADCAST
Priority	T_PRIORITY
Protocol_Info	*protoent*
Reuse_Address	IP_REUSEADDR
Routine	T_ROUTINE
Send_Garbage	T_GARBAGE
String_To_Internet_Address	*inet_addr()*
TCP_Level	INET_TCP
Time_To_Live	IP_TTL
Type_Of_Service	IP_TOS
UDP_Checksum	UDP_CHECKSUM
UDP_Level	INET_UDP

Package POSIX_XTI_ISO (D.2.2)

Ada Name	C Name
Absolute_Requirement	T_ABSREQ
Acknowledge_Time	TCO_ACKTIME
Active_Protection	T_ACTIVEPROTECT
Alternative_Class_1	TCO_ALTCLASS1
Alternative_Class_2	TCO_ALTCLASS2
Alternative_Class_3	TCO_ALTCLASS3
Alternative_Class_4	TCO_ALTCLASS4
Class_0	T_CLASS0
Class_1	T_CLASS1
Class_2	T_CLASS2
Class_3	T_CLASS3
Class_4	T_CLASS4
Connection_Checksum	TCO_CHECKSUM
Connection_Resilience	TCO_CONNRESIL
Connection_Transit_Delay	TCO_TRANSDEL
Connectionless_Checksum	TCL_CHECKSUM
Connectionless_Transit_Delay	TCL_TRANSDEL
Default	T_PRIDFLT
Establishment_Delay	TCO_ESTDELAY
Establishment_Fail_Probability	TCO_ESTFAILPROB
Expedited_Data	TCO_EXPD
Extended_Format	TCO_EXTFORM
Flow_Control	TCO_FLOWCTRL

Package POSIX_XTI_mOSI (D.2.1)

Ada Name	C Name

Presentation_Context_Item_Element . . . . . *t_ap_syn_off*
Presentation_Context_Item_Header . . . . . *t_ap_pc_item*
Presentation_Context_List . . . . . *t_ap_pco_el*
Presentation_Context_Rejected . . . . . T_PCL_USER_REJ
Rejected_By_Peer . . . . . T_AC_U_AARE__NONE
Rejected_No_Reason_Specified . . . . . T_PCL_PREJ_RSN_NSPEC
Transfer_Syntax_Not_Supported . . . . . T_PCL_PREJ_T_SYTX_NSUP
Unrecognized_AE_Qualifier . . . . . T_AC_U_AARE_AEQ
Unrecognized_AP_Title . . . . . T_AC_U_AARE_APT

c

Package System (2.8)

This package is specific to the Ada language.

Package System_Storage_Elements (2.9)

This package is specific to the Ada language.

C.2 C-to-Ada Cross-Reference

This clause lists the Ada-language name or names that correspond most closely to each C-language name defined by the base standards.

C Name	Ada Name
accept()	POSIX_Sockets.Accept_Connection
access	POSIX_Files.Is_Accessible
access(), F_OK	POSIX_Files.Existence
access(), F_OK	POSIX_Files.Is_File_Present
addrinfo	POSIX_Sockets.Socket_Address_Info
AF_INET	*C-language specific*
AF_ISO	*C-language specific*
AF_LOCAL	*C-language specific*
AF_OSI	*C-language specific*
AF_UNSPEC	*C-language specific*
AI_CANONNAME	POSIX_Sockets.Get_Canonical_Name
AI_PASSIVE	POSIX_Sockets.Use_For_Binding
<aio.h>	POSIX_Asynchronous_IO
AIO_ALLDONE	POSIX_Asynchronous_IO.All_Done
aio_cancel()	POSIX_Asynchronous_IO.Cancel
AIO_CANCELED	POSIX_Asynchronous_IO.Canceled
aiocb	*C-language specific*
aio_error()	POSIX_Asynchronous_IO.Get_AIO_Error_Code
aio_error()	POSIX_Asynchronous_IO.Get_AIO_Status
aio_fsync()	POSIX_Asynchronous_IO.Synchronize_File
AIO_LISTIO_MAX	POSIX_Limits.List_IO_Maxima'Last
AIO_MAX	POSIX_Limits.Asynchronous_IO_Maxima'Last

c

POLLWRNORM . . . . . POSIX_Event_Management.Write_Normal c
_POSIX_AIO_LISTIO_MAX . . . . . POSIX_Limits.Portable_List_IO_Maximum
_POSIX_AIO_MAX . . . . . POSIX_Limits.Portable_Asynchronous_IO_Maximum
_POSIX_ARG_MAX . . . . . POSIX_Limits.Portable_Argument_List_Maximum
_POSIX_ASYNCHRONOUS_IO . . . . . POSIX_Options.Asynchronous_IO_Support
_POSIX_ASYNC_IO . . POSIX_Configurable_File_Limits.Asynchronous_IO_Is_Supported
_POSIX_CHILD_MAX . . . . . POSIX_Limits.Portable_Child_Processes_Maximum
_POSIX_CHOWN_RESTRICTED . . . . . POSIX_Options.Change_Owner_Restricted
_POSIX_CHOWN_RESTRICTED . . . . . .
POSIX_Configurable_File_Limits.Change_Owner_Restricted
_POSIX_CLOCKRES_MIN . . . . . POSIX_Limits.Portable_Clock_Resolution_Minimum
_POSIX_DELAYTIMER_MAX . . . . . POSIX_Limits.Portable_Timer_Overruns_Maximum
_POSIX_FD_SETSIZE . . . . . POSIX_Limits.Portable_FD_Set_Maximum c
_POSIX_FSYNC . . . . . POSIX_Options.File_Synchronization_Support
_POSIX_HIWAT . . . . . POSIX_Limits.Portable_Socket_Buffer_Maximum c
_POSIX_JOB_CONTROL . . . . . POSIX_Options.Job_Control_Support
_POSIX_LINK_MAX . . . . . POSIX_Limits.Portable_Links_Maximum
_POSIX_MAPPED_FILES . . . . . POSIX_Options.Memory_Mapped_Files_Support
_POSIX_MAX_CANON . . . . . POSIX_Limits.Portable_Input_Line_Maximum
_POSIX_MAX_INPUT . . . . . POSIX_Limits.Portable_Input_Queue_Maximum
_POSIX_MEMLOCK_RANGE . . . . . POSIX_Options.Memory_Range_Locking_Support
_POSIX_MEMLOCK . . . . . POSIX_Options.Memory_Locking_Support
_POSIX_MEMORY_PROTECTION . . . . . POSIX_Options.Memory_Protection_Support
_POSIX_MESSAGE_PASSING . . . . . POSIX_Options.Message_Queues_Support
_POSIX_MQ_OPEN_MAX . . . . . POSIX_Limits.Portable_Open_Message_Queues_Maximum
_POSIX_MQ_PRIO_MAX . . . . . POSIX_Limits.Portable_Message_Priority_Maximum
_POSIX_NAME_MAX . . . . . POSIX_Limits.Portable_Filename_Maximum
_POSIX_NGROUPS_MAX . . . . . POSIX_Limits.Portable_Groups_Maximum
_POSIX_NO_TRUNC . . . . . POSIX_Options.Filename_Truncation
_POSIX_NO_TRUNC . . . . . POSIX_Configurable_File_Limits.Filename_Is_Truncated
_POSIX_NO_TRUNC . . . . . POSIX_Options.Filename_Truncation
_POSIX_OPEN_MAX . . . . . POSIX_Limits.Portable_Open_Files_Maximum
_POSIX_PATH_MAX . . . . . POSIX_Limits.Portable_Pathname_Maximum
_POSIX_PII . . . . . *C-language specific*
_POSIX_PII_INTERNET . . . . . POSIX_Options.Internet_Protocol_Support
_POSIX_PII_INTERNET_DGRAM . . . . . POSIX_Options.Internet_Datagram_Support
_POSIX_PII_INTERNET_STREAM . . . . . POSIX_Options.Internet_Stream_Support
_POSIX_PII_NET_SUPPORT . . . . . POSIX_Options.Network_Management_Support
_POSIX_PII_OSI . . . . . POSIX_Options.ISO_OSI_Protocol_Support
_POSIX_PII_OSI_CLTS . . . . . POSIX_Options.OSI_Connectionless_Support
_POSIX_PII_OSI_COTS . . . . . POSIX_Options.OSI_Connection_Support
_POSIX_PII_OSI_M . . . . . POSIX_Options.OSI_Minimal_Support
_POSIX_PII_SOCKET . . . . . POSIX_Options.Sockets_DNI_Support
_POSIX_PII_XTI . . . . . POSIX_Options.XTI_DNI_Support c
_POSIX_PIPE_BUF . . . . . POSIX_Limits.Portable_Pipe_Limit_Maximum
_POSIX_POLL . . . . . POSIX_Options.Poll_Support c

SIGRTMAX . . . . . POSIX_Signals.Realtime_Signal'Last
SIGRTMIN . . . . . POSIX_Signals.Realtime_Signal'First
SIGSEGV . . . . . POSIX_Signals.Signal_Segmentation_Violation
SIGSEGV . . . . . .Standard.Program_Error
sigset_t . . . . . POSIX_Signals.Signal_Set
SIG_SETMASK . . . . . *C-language specific*
SIGSTOP . . . . . .POSIX_Signals.Signal_Stop
sigsuspend() . . . . . *C-language specific*
SIGTERM . . . . . POSIX_Signals.Signal_Terminate
sigtimedwait() . . . . . POSIX_Signals.Await_Signal_Or_Timeout
SIGTSTP . . . . . POSIX_Signals.Signal_Terminal_Stop
SIGTTIN . . . . . POSIX_Signals.Signal_Terminal_Input
SIGTTOU . . . . . .POSIX_Signals.Signal_Terminal_Output
SIG_UNBLOCK . . . . . *C-language specific*
SIGURG . . . . . POSIX_Signals.Signal_Out_Of_Band_Data C
SIGUSR1 . . . . . POSIX_Signals.Signal_User_1
SIGUSR2 . . . . . POSIX_Signals.Signal_User_2
sigval . . . . . .POSIX_Signals.Signal_Data
sigwaitinfo() . . . . . POSIX_Signals.Await_Signal
sigwaitinfo() . . . . . POSIX_Signals.Await_Signal
SI_ASYNCIO . . . . . POSIX_Signals.From_Async_IO
SI_MESGQ . . . . . POSIX_Signals.From_Message_Queue
SI_QUEUE . . . . . POSIX_Signals.From_Queue_Signal
SI_TIMER . . . . . POSIX_Signals.From_Timer
SI_USER . . . . . .POSIX_Signals.From_Send_Signal
S_IRGRP . . . . . .POSIX_Permissions.Group_Read
S_IROTH . . . . . POSIX_Permissions.Others_Read
S_IRUSR . . . . . .POSIX_Permissions.Owner_Read
S_IRWXG . . . . . POSIX_Permissions.Group_Permission_Set
S_IRWXO . . . . . POSIX_Permissions.Others_Permission_Set
S_IRWXU . . . . . POSIX_Permissions.Owner_Permission_Set
S_ISBLK . . . . . POSIX_File_Status.Is_Block_Special_File
S_ISBLK . . . . . .POSIX_Files.Is_Block_Special_File
S_ISCHR . . . . . POSIX_File_Status.Is_Character_Special_File
S_ISCHR . . . . . POSIX_Files.Is_Character_Special_File
S_ISDIR . . . . . .POSIX_File_Status.Is_Directory
S_ISDIR . . . . . POSIX_Files.Is_Directory
S_ISFIFO . . . . . .POSIX_File_Status.Is_FIFO
S_ISFIFO . . . . . POSIX_Files.Is_FIFO
S_ISGID . . . . . .POSIX_Permissions.Set_Group_ID
S_ISGID . . . . . .POSIX_Permissions.Set_User_ID_Set
S_ISREG . . . . . .POSIX_File_Status.Is_File
S_ISREG . . . . . POSIX_Files.Is_Regular_File
S_ISUID . . . . . POSIX_Permissions.Set_Group_ID_Set
S_ISUID . . . . . POSIX_Permissions.Set_User_ID
SI_USER . . . . . .POSIX_Signals.From_Send_Signal

C

T_INETCONTROL . . . . . POSIX_XTI_Internet.Internetwork_Control
T_INFINITE . . . . . *C-language specific*
T_INFO . . . . . *C-language specific*
t_info . . . . . POSIX_XTI.Communications_Provider_Info
T_INREL . . . . . POSIX_XTI.Incoming_Release
T_INVALID . . . . . *C-language specific*
T_IOV_MAX . . . . . POSIX_Limits.XTI_IO_Vector_Maxima'Last
t_iovec . . . . . POSIX_XTI.IO_Vector
T_ISO_APCL . . . . . POSIX_XTI_mOSI.mOSI_Connectionless_Mode
T_ISO_APCO . . . . . POSIX_XTI_mOSI.mOSI_Connection_Mode
t_kpalive . . . . . POSIX_XTI_Internet.Keep_Alive_Status
T_LDELAY . . . . . POSIX_XTI_Internet.Low_Delay
t_linger . . . . . POSIX_XTI.Linger_Info
T_LISTEN . . . . . POSIX_XTI.Connect_Request_Received
t_listen() . . . . . POSIX_XTI.Listen
T_LOCOST . . . . . POSIX_XTI_Internet.Low_Cost
t_look() . . . . . POSIX_XTI.Look
T_MORE . . . . . POSIX_XTI.More_Data
t_mosiaddr . . . . . POSIX_XTI_mOSI.mOSI_Address
T_NEGOTIATE . . . . . POSIX_XTI.Set_Options
T_NETCONTROL . . . . . POSIX_XTI_Internet.Network_Control
T_NO . . . . . POSIX_XTI_Internet.Keep_Alive_Off
T_NOPROTECT . . . . . POSIX_XTI.OSI.No_Protection
T_NOTOS . . . . . POSIX_XTI_Internet.Normal
T_NOTSUPPORT . . . . . POSIX_XTI.Not_Supported
T_NULL . . . . . *C-language specific*
t_open() . . . . . POSIX_XTI.Open
T_OPT . . . . . *C-language specific*
T_OPT_VALEN . . . . . *C-language specific*
t_opthdr . . . . . POSIX_XTI.Options_Header
T_OPTMGMT . . . . . *C-language specific*
t_optmgmt . . . . . POSIX_XTI.Options_Management_Info
t_optmgmt() . . . . . POSIX_XTI.Manage_Options
T_ORDREL . . . . . POSIX_XTI.Orderly_Release_Request_Received
T_ORDRELDATA . . . . . POSIX_XTI.Orderly_Release_Data_Supported
T_OSI_AP_IID_BIT . . . . . POSIX_XTI_mOSI.AP_Invocation_Id
T_OSI_AE_IID_BIT . . . . . POSIX_XTI_mOSI.AE_Invocation_Id
T_OUTCON . . . . . POSIX_XTI.Outgoing_Connect
T_OUTREL . . . . . POSIX_XTI.Outgoing_Release
T_OVERRIDEFLASH . . . . . POSIX_XTI_Internet.Flash_Override
T_PARTSUCCESS . . . . . POSIX_XTI.Partial_Success
T_PASSIVEPROTECT . . . . . POSIX_XTI.OSI.Passive_Protection
T_PCL_ACCEPT . . . . . POSIX_XTI_mOSI.Presentation_Context_Accepted
T_PCL_PREJ_A_SYTX_NSUP . . . . . POSIX_XTI_mOSI.Abstract_Syntax_Not_Supported
T_PCL_PREJ_LMT_DCS_EXCEED . . . . . POSIX_XTI_mOSI.Local_DCS_Limit_Exceeded
T_PCL_PREJ_RSN_NSPEC . . . . . POSIX_XTI_mOSI.Rejected_No_Reason_Specified

TBADFLAG . . . . . POSIX.Invalid_Flag
TBADNAME . . . . . POSIX.Invalid_Communications_Provider
TBADOPT . . . . . POSIX.Incorrect_Or_Illegal_Option
TBADQLEN . . . . . POSIX.Endpoint_Queue_Length_Is_Zero
TBADSEQ . . . . . POSIX.Invalid_Sequence_Number
TBUFOVFLW . . . . . POSIX.Buffer_Not_Large_Enough
tcdrain() . . . . . POSIX_Terminal_Functions.Drain
tcflag_t . . . . . POSIX_Terminal_Functions.Terminal_Modes
tcflow() . . . . . POSIX_Terminal_Functions.Flow
tcflush() . . . . . POSIX_Terminal_Functions.Discard_Data
tcgetattr() . . . . . POSIX_Terminal_Functions.Get_Terminal_Characteristics
tcgetpgrp() . . . . . POSIX_Terminal_Functions.Get_Process_Group_ID
TCIFLUSH . . . . . POSIX_Terminal_Functions.Received_But_Not_Read
TCIOFF . . . . . POSIX_Terminal_Functions.Transmit_Stop
TCIOFLUSH . . . . . POSIX_Terminal_Functions.Both
TCION . . . . . POSIX_Terminal_Functions.Transmit_Start
TCL_CHECKSUM . . . . . POSIX_XTI_ISO.Connectionless_Checksum
TCL_PRIORITY . . . . . POSIX_XTI_ISO.Priority
TCL_PROTECTION . . . . . POSIX_XTI_ISO.Protection
TCL_RESERRORRATE . . . . . POSIX_XTI_ISO.Residual_Error_Rate
TCL_TRANSDEL . . . . . POSIX_XTI_ISO.Connectionless_Transit_Delay
TCO_ACKTIME . . . . . POSIX_XTI_ISO.Acknowledge_Time
TCO_ALTCLASS1 . . . . . POSIX_XTI_ISO.Alternative_Class_1
TCO_ALTCLASS2 . . . . . POSIX_XTI_ISO.Alternative_Class_2
TCO_ALTCLASS3 . . . . . POSIX_XTI_ISO.Alternative_Class_3
TCO_ALTCLASS4 . . . . . POSIX_XTI_ISO.Alternative_Class_4
TCO_CHECKSUM . . . . . POSIX_XTI_ISO.Connection_Checksum
TCO_CONNRESIL . . . . . POSIX_XTI_ISO.Connection_Resilience
TCO_ESTDELAY . . . . . POSIX_XTI_ISO.Establishment_Delay
TCO_ESTFAILPROB . . . . . POSIX_XTI_ISO.Establishment_Fail_Probability
TCO_EXPD . . . . . POSIX_XTI_ISO.Expedited_Data
TCO_EXTFORM . . . . . POSIX_XTI_ISO.Extended_Format
TCO_FLOWCTRL . . . . . POSIX_XTI_ISO.Flow_Control
TCO_LTPDU . . . . . POSIX_XTI_ISO.TPDU_Length_Maximum
TCO_NETEXP . . . . . POSIX_XTI_ISO.Network_Expedited_Data
TCO_NETRECPTCF . . . . . POSIX_XTI_ISO.Network_Receipt_Confirmation
TCO_PREFCLASS . . . . . POSIX_XTI_ISO.Preferred_Class
TCO_PRIORITY . . . . . POSIX_XTI_ISO.Priority
TCO_PROTECTION . . . . . POSIX_XTI_ISO.Protection
TCO_REASTIME . . . . . POSIX_XTI_ISO.Reassignment_Time
TCO_RELDELAY . . . . . POSIX_XTI_ISO.Release_Delay
TCO_RELFAILPROB . . . . . POSIX_XTI_ISO.Release_Fail_Probability
TCO_RESERRORRATE . . . . . POSIX_XTI_ISO.Residual_Error_Rate
TCO_THROUGHPUT . . . . . POSIX_XTI_ISO.Throughput
TCO_TRANSDEL . . . . . POSIX_XTI_ISO.Connection_Transit_Delay
TCO_TRANSFFAILPROB . . . . . POSIX_XTI_ISO.Transfer_Fail_Probability

c

Annex D
(normative)
Protocol Mappings

This annex describes Ada mappings to facilities that support specific network protocols.

D.1 Sockets Protocol Mappings

This clause describes Ada mappings to facilities that support specific sockets network protocols.

D.1.1 Package POSIX_Sockets_Local

This package provides DNI/Socket interface mappings for local IPC. Unless otherwise specified, all the DNI/Socket calls in package POSIX_Sockets can be used for IPC. (See 18.4.) Only additional information relevant to IPC is highlighted here.

The functionality described in this subclause is optional. If the Sockets Detailed Network Interface option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

```
with POSIX,
     POSIX_IO,
     POSIX_Sockets;
package POSIX_Sockets_Local is
    -- D.1.1.1 Local IPC Protocol Family
   Local_Protocol :
      constant POSIX_Sockets.Protocol_Family := impl-def-static-expression;
   type Local_Socket_Address is private;
   type Local_Socket_Address_Pointer is access all Local_Socket_Address;
   function "+" (Pointer : Local_Socket_Address_Pointer)
     return POSIX_Sockets.Socket_Address_Pointer;
   function "+" (Pointer : POSIX_Sockets.Socket_Address_Pointer)
     return Local_Socket_Address_Pointer;
   function Is_Local_Socket_Address
      (Pointer : POSIX_Sockets.Socket_Address_Pointer)
     return Boolean;
   function Get_Socket_Path (Name : Local_Socket_Address)
     return POSIX.Pathname;
   procedure Set_Socket_Path
      (Name : in out Local_Socket_Address;
       Path : in     POSIX.Pathname);
   function Get_Socket_Name (Handle : POSIX_Sockets.Socket_Message)
     return Local_Socket_Address;
   function Get_Address (Info_Item : POSIX_Sockets.Socket_Address_Info)
     return Local_Socket_Address;
   function Get_Peer_Name (Socket : POSIX_IO.File_Descriptor)
     return Local_Socket_Address;
   function Get_Socket_Name (Socket : POSIX_IO.File_Descriptor)
     return Local_Socket_Address;
```

```
private
   implementation-defined
end POSIX_Sockets_Local;
```

D.1.1.1 Local IPC Protocol Family

D.1.1.1.1 Synopsis

```
Local_Protocol :
   constant POSIX_Sockets.Protocol_Family := impl-def-static-expression;
type Local_Socket_Address is private;
type Local_Socket_Address_Pointer is access all Local_Socket_Address;
function "+" (Pointer : Local_Socket_Address_Pointer)
  return POSIX_Sockets.Socket_Address_Pointer;
function "+" (Pointer : POSIX_Sockets.Socket_Address_Pointer)
  return Local_Socket_Address_Pointer;
function Is_Local_Socket_Address
   (Pointer : POSIX_Sockets.Socket_Address_Pointer)
  return Boolean;
function Get_Socket_Path (Name : Local_Socket_Address)
  return POSIX.Pathname;
procedure Set_Socket_Path
   (Name : in out Local_Socket_Address;
    Path : in     POSIX.Pathname);
function Get_Socket_Name (Handle : POSIX_Sockets.Socket_Message)
  return Local_Socket_Address;
function Get_Address (Info_Item : POSIX_Sockets.Socket_Address_Info)
  return Local_Socket_Address;
function Get_Peer_Name (Socket : POSIX_IO.File_Descriptor)
  return Local_Socket_Address;
function Get_Socket_Name (Socket : POSIX_IO.File_Descriptor)
  return Local_Socket_Address;
```

D.1.1.1.2 Description

The local IPC protocol family provides communication between processes on the same system using the socket interface. Addresses used for sockets in local IPC are pathnames. The family provides support for `Stream_Socket` and `Datagram_Socket` socket types using unspecified internal communication mechanisms.

Given a local IPC socket address, `Get_Socket_Path` returns the corresponding pathname. This value is set to an empty string when a socket address is created and needs to be set to a valid pathname by the application.

`Set_Socket_Path` sets the pathname in a local IPC socket address. An unspecified address for a local IPC socket shall be indicated by an empty string as the corresponding pathname.

Binding a pathname in the local domain creates a name for a socket in the file system. The `Bind` procedure shall be called with a `Local_Socket_Address` object containing the pathname for the socket. The `Bind` procedure uses the file creation mask of `POSIX_Permissions.Set_Allowed_Process_Permissions` to modify the default permissions of read/write/execute by owner/group/others. The file shall be created as if by a call to the `POSIX_Files.Create_FIFO` operation, with the exception that the resulting file shall be a socket rather than a FIFO special file.

The call shall generate the exception POSIX_Error if the corresponding call to the POSIX_Files.Create_FIFO operation would fail, and Get_Error_Code shall return the value indicated for POSIX_Files.Create_FIFO.

If the pathname of the socket is not an absolute pathname, the results are unspecified.

If the Bind operation succeeds, a name for the socket is created in the file system. When the application is finished with the socket, it should be removed (using the POSIX_Files.Unlink procedure), or subsequent attempts to bind to that socket may fail.

When connecting to a socket in the local domain, the application shall call Connect with the Name parameter (a Local_Socket_Address object) containing the pathname for the socket. The pathname must name an existing socket. The name shall be subjected to pathname resolution as for a call to the Open procedure called with a Mode value of POSIX_IO.Write_Only; if that open would fail for any reason other than the type of the file with the specified name, the Connect procedure shall fail with the corresponding value.

The type Local_Socket_Address shall be used to represent an address for this protocol family. The type Local_Socket_Address_Pointer is an access to this protocol-specific address type. The "+" operations shall convert a Local_Socket_-Address_Pointer to and from the Socket_Address_Pointer type for use with the base package operations defined for the Socket_Address_Pointer type. The return value of the "+" operations designates the same address object as the input parameter. The function Is_Local_Socket_Address shall return True if the address object designated by the specified non-null Socket_Address_Pointer is a valid Local_Socket_Address and False otherwise. The conversion operation to Local_Socket_Address_Pointer shall succeed if and only if the corresponding Is_Local_Socket_Address returns True. Otherwise, the results are undefined.

NOTE: The Null_Socket_Address constant corresponds to the Ada null literal for these operations.

Get_Socket_Name shall return the name (address) associated with a socket. Get_-Peer_Name shall return the socket address of the peer connected to a socket. For Get_Socket_Name and Get_Peer_Name, the Socket parameter is an open file descriptor referring to a socket. A call to Get_Socket_Name for a socket in the Ground state shall return the same value as that of a new socket address returned by Create.

Get_Address returns the Address attribute of a Socket_Address_Info object as a Local_Socket_Address object. Get_Socket_Name returns the Name attribute of a Socket_Message object as a Local_Socket_Address object.

D.1.1.1.3 Error Handling

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Incorrect_Address_Type

The type of the object designated by the return value is not appropriate for

the address format of this socket.

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: The `Socket` parameter is not valid.

`No_Buffer_Space`
: Insufficient resources were available in the system to perform the operation.

`Not_Connected`
: The socket is not connected or otherwise has not had the peer prespecified.

`Not_A_Socket`
: The file descriptor `Socket` does not refer to a socket.

`Invalid_Argument`
: A call was made to `Bind` with a socket that is already bound to a pathname.

`Not_A_Socket`
: A call was made to `Connect` with a pathname that exists but is not a socket.

`Connection_Refused`
: A call was made to `Connect` with a pathname that exists and is a socket, but no open file descriptor for the named socket exists.

`Wrong_Protocol_Type`
: A call was made to `Connect` function with a pathname that exists and is a socket, but the socket is not of the same type as the socket attempting the connection.

D.1.1.2 Stream sockets for local IPC

D.1.1.2.1 Description

Local stream sockets provide a flow controlled, reliable, bidirectional connection-based service that does not preserve message boundaries. Local stream sockets support all states in Figure D.1. Local stream sockets are created with an empty pathname as a local (bound) address. A protocol family of `Local_Protocol` and a socket type of `Stream_Socket` are specified on the `Create` call.

Local stream socket names follow the conventions for `Bind` described in D.1.1.1.2.

Local stream socket names follow conventions for the `Connect` procedure described in D.1.1.1.2. If the pathname exists and is a socket, but the named socket is not in the Listening state or the connection queue for the named socket has reached its limit, the connection attempt shall fail. The call to `Connect` shall complete without blocking if the target socket exists and is in the Listening state; it shall return without waiting for a peer process to call the `Accept_Connection` procedure on the listening socket. If no local socket address has been bound to the socket, the local socket address shall not be bound by the call to the `Connect` procedure.

If a socket in the Ground state connects to a Listening socket, the address returned by `Accept_Connection` shall specify the peer address as an empty string. The local address of the new socket shall be the same as that of the listening socket.

If the application makes a call to the `Shutdown` procedure to cease output, an end-of-file shall be indicated to the peer socket once any pending data have been read. If the application makes a call to `Shutdown` with a `Shutdown_Mode` of `Further_Receives_Disallowed`, no action shall be taken. When a local stream socket is closed in the Listening state pending connections to that socket that have not been returned by `Accept_Connection` function shall be aborted.

Neither an immediate disconnect nor a normal disconnect on a local stream socket shall cause data to be discarded. Any data that have been sent shall be in the receive queue of the peer socket. The `Close` procedure shall not block.

It is implementation defined whether the local stream protocol supports out-of-band data.

No protocol options are defined for local stream sockets.

D.1.1.2.2 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Connection_Refused`

A call was made to `Connect` with a pathname that exists and is a socket, but the named socket is not in the Listening state or the connection queue for the named socket has reached its limit.

`Connection_Reset`

A local stream socket was closed in the Listening state, and pending connections to that socket not returned by `Accept_Connection` were aborted.

`Is_Already_Connected`

A call was made to `Connect` with a pathname of a socket that is already connected.

`Option_Not_Supported`

A send or receive function specified the `Process_OOB_Data` and out-of-band data are not supported.

D.1.1.3 Datagram sockets for local IPC

D.1.1.3.1 Description

Local `Datagram_Socket` sockets provide a connectionless mode service that preserves message boundaries, but does not necessarily provide error detection. Local datagram sockets support all states in Figure D.2. If an application sends a datagram using a socket in the Ground state, the operation shall complete normally, but the socket shall remain in the Ground state. The source address of the datagram sent shall be an empty string. A protocol family of `Local_Protocol` and a socket type of `Datagram_Socket` are specified on the `Create` call.

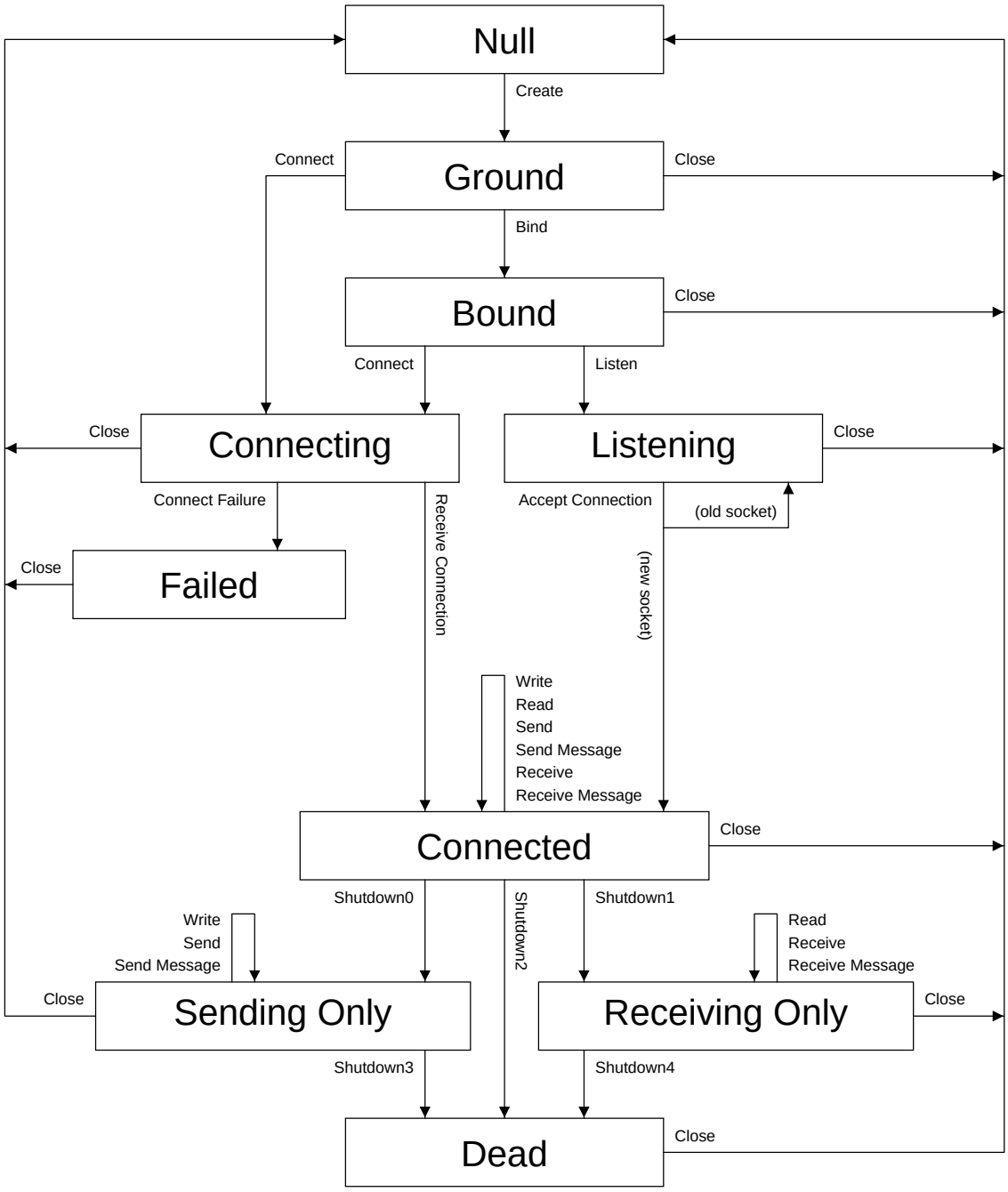

Figure D.1 – Local IPC (and TCP) Stream Sockets State Diagram

Local datagram socket names follow the conventions for `Bind` described in D.1.1.1.2.

Local `Datagram_Socket` socket names follow the conventions for `Connect` described in D.1.1.1.2. If no local socket address has been bound to the socket, the local socket address shall not be bound by the call to `Connect`. A connectionless-mode socket in the Open state can still receive datagrams from sources other than the peer.

Closing a local connectionless-mode socket shall have no effect on data previously sent.

Local connectionless-mode sockets shall not support out-of-band data.

No protocol options are defined for local connectionless-mode sockets.

D.1.1.3.2 Error Handling

If the following condition occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Option_Not_Supported`
: A send or receive function specified `Process_OOB_Data` and out-of-band data are not supported.

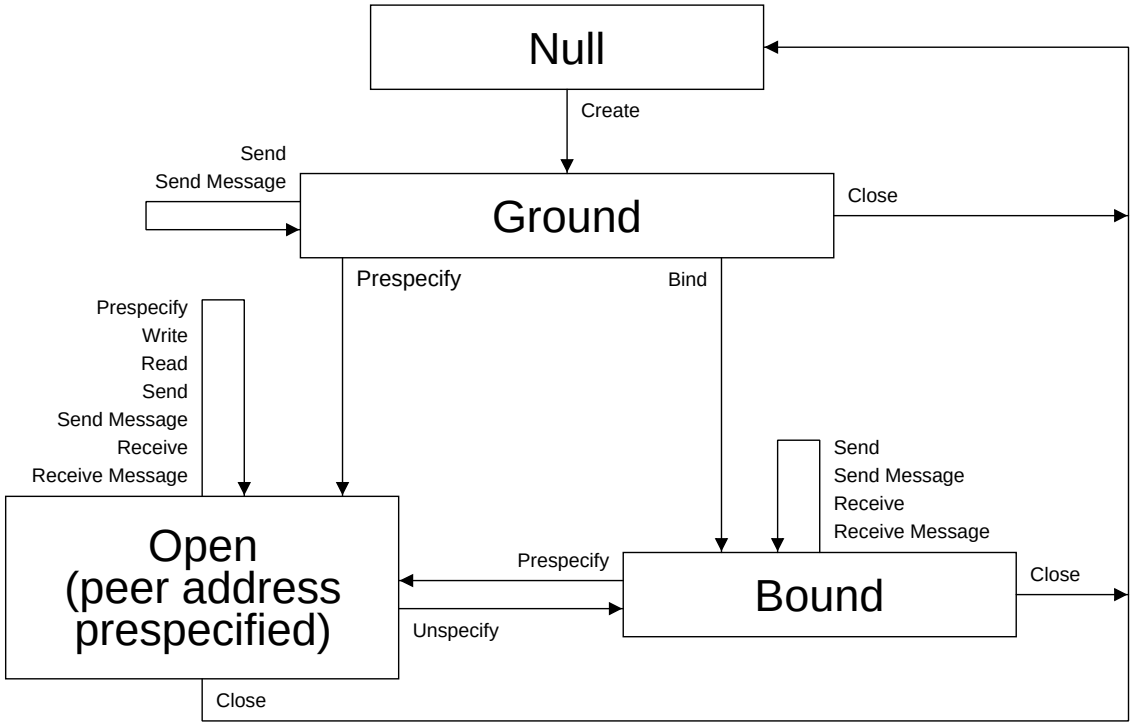

Figure D.2 – Local IPC Datagram Sockets State Diagram

D.1.2 Package `POSIX_Sockets_ISO`

This package provides the DNI/Socket interface mappings for ISO transport protocols. Unless otherwise specified, all the DNI/Socket calls in package `POSIX_Sockets` can be used. Only additional information relevant to the ISO transport protocol is highlighted here.

The functionality described in this subclause is optional. If either the Sockets Detailed Network Interface option or the ISO/OSI Protocol option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this subclause.

```
with POSIX,
     POSIX_IO,
     POSIX_Sockets;
package POSIX_Sockets_ISO is
   -- D.1.2.1 ISO Protocol Family
   ISO_Protocol           :
      constant POSIX_Sockets.Protocol_Family := impl-def-static-expression;
   ISO_Transport_Protocol :
      constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
   Connectionless_Mode_Transport_Protocol :
      constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
   Connectionless_Mode_Network_Protocol :
      constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
   type ISO_Socket_Address is private;
   type ISO_Socket_Address_Pointer is access all ISO_Socket_Address;
   function "+" (Pointer : ISO_Socket_Address_Pointer)
     return POSIX_Sockets.Socket_Address_Pointer;
   function "+" (Pointer : POSIX_Sockets.Socket_Address_Pointer)
     return ISO_Socket_Address_Pointer;
   function Is_ISO_Socket_Address
      (Pointer : POSIX_Sockets.Socket_Address_Pointer)
     return Boolean;
   type ISO_Address is new POSIX.Octet_Array;
   type Presentation_Selector is new POSIX.Octet_Array;
   type Session_Selector is new POSIX.Octet_Array;
   type Transport_Selector is new POSIX.Octet_Array;
   type GOSIP_Selector is new POSIX.Octet_Array;
   function Get_ISO_Address (Name : ISO_Socket_Address)
     return ISO_Address;
   procedure Set_ISO_Address
      (Name    : in out ISO_Socket_Address;
       Address : in     ISO_Address);
   function Get_Presentation_Selector (Name : ISO_Socket_Address)
     return Presentation_Selector;
   procedure Set_Presentation_Selector
      (Name     : in out ISO_Socket_Address;
       Selector : in     Presentation_Selector);
   function Get_Session_Selector (Name : ISO_Socket_Address)
     return Session_Selector;
   procedure Set_Session_Selector
      (Name     : in out ISO_Socket_Address;
       Selector : in     Session_Selector);
   function Get_Transport_Selector (Name : ISO_Socket_Address)
     return Transport_Selector;
   procedure Set_Transport_Selector
      (Name     : in out ISO_Socket_Address;
       Selector : in     Transport_Selector);
   function Get_GOSIP_Selector (Name : ISO_Socket_Address)
     return GOSIP_Selector;
   procedure Set_GOSIP_Selector
      (Name     : in out ISO_Socket_Address;
       Selector : in     GOSIP_Selector);
   function Get_Socket_Name (Handle : POSIX_Sockets.Socket_Message)
     return ISO_Socket_Address;
   function Get_Address (Info_Item : POSIX_Sockets.Socket_Address_Info)
     return ISO_Socket_Address;
   function Get_Peer_Name (Socket : POSIX_IO.File_Descriptor)
     return ISO_Socket_Address;
```

```
function Get_Socket_Name (Socket : POSIX_IO.File_Descriptor)
  return ISO_Socket_Address;
-- D.1.2.2 Connectionless-Mode ISO Sockets Protocols
type CL_Options is new POSIX.Octet_Array;
type CL_Flags is new POSIX.Option_Set;
No_Segmentation     : constant CL_Flags := implementation-defined;
Suppress_Error_PDUs : constant CL_Flags := implementation-defined;
No_Checksum         : constant CL_Flags := implementation-defined;
function Get_CL_Options
   (Socket : POSIX_IO.File_Descriptor)
  return CL_Options;
procedure Set_CL_Options
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in CL_Options);
function Get_CL_Flags
   (Socket : POSIX_IO.File_Descriptor)
  return CL_Flags;
procedure Set_CL_Flags
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in CL_Flags);
-- D.1.2.3 TP - ISO Transport Protocol
type TP_Flags is new POSIX.Option_Set;
Public_Data_Network_QOS : constant TP_Flags := implementation-defined;
Peer_On_Same_Network    : constant TP_Flags := implementation-defined;
Expedited_Data_Present  : constant TP_Flags := implementation-defined;
function Get_TP_Flags
   (Socket : POSIX_IO.File_Descriptor)
  return TP_Flags;
function Get_Connection_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.Octet_Array;
procedure Set_Connection_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.Octet_Array);
function Get_Disconnect_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.Octet_Array;
procedure Set_Disconnect_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.Octet_Array);
function Get_Confirmation_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.Octet_Array;
procedure Set_Confirmation_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.Octet_Array);
type TP_Ancillary_Data_Type is
   (Connection_Data, Disconnect_Data, Confirmation_Data);
type TP_Ancillary_Data (Kind : TP_Ancillary_Data_Type;
                        Size : Positive) is private;
procedure Set_Ancillary_Data
   (Message : in out POSIX_Sockets.Socket_Message;
    Object  : in     TP_Ancillary_Data);
function Get_Ancillary_Data
   (Message : POSIX_Sockets.Socket_Message)
  return TP_Ancillary_Data;
procedure Set_Ancillary_Data_Array
   (Object : in out TP_Ancillary_Data;
```

```
    Data   : in     POSIX.Octet_Array);
function Get_Ancillary_Data_Array
   (Object : TP_Ancillary_Data)
  return POSIX.Octet_Array;
type Connection_Parameters is private;
function Get_Connection_Parameters
   (Socket : POSIX_IO.File_Descriptor)
  return Connection_Parameters;
procedure Set_Connection_Parameters
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Connection_Parameters);
function Get_Retransmit_Number
   (Object : Connection_Parameters)
  return Natural;
procedure Set_Retransmit_Number
   (Object : in out Connection_Parameters;
    To     : in     Natural);
type Window_Size is range 128 .. 16384;
function Get_Window_Size
   (Object : Connection_Parameters)
  return Window_Size;
procedure Set_Window_Size
   (Object : in out Connection_Parameters;
    To     : in     Window_Size);
type TPDU_Size is range 7 .. 13;
function Get_TPDU_Size
   (Object : Connection_Parameters)
  return TPDU_Size;
procedure Set_TPDU_Size
   (Object : in out Connection_Parameters;
    To     : in     TPDU_Size);
type TP_Acknowledgment_Strategy is
   (Acknowledge_Window, Acknowledge_Each);
function Get_Acknowledgment_Strategy
   (Object : Connection_Parameters)
  return TP_Acknowledgment_Strategy;
procedure Set_Acknowledgment_Strategy
   (Object : in out Connection_Parameters;
    To     : in     TP_Acknowledgment_Strategy);
type TP_Retransmit_Strategy is
   (Retransmit_Each_Packet, Use_Congestion_Window, Fast_Start);
function Get_Retransmit_Strategy
   (Object : Connection_Parameters)
  return TP_Retransmit_Strategy;
procedure Set_Retransmit_Strategy
   (Object : in out Connection_Parameters;
    To     : in     TP_Retransmit_Strategy);
type TP_Class is new POSIX.Option_Set;
TP_Class_0 : constant TP_Class := implementation-defined;
TP_Class_1 : constant TP_Class := implementation-defined;
TP_Class_2 : constant TP_Class := implementation-defined;
TP_Class_3 : constant TP_Class := implementation-defined;
TP_Class_4 : constant TP_Class := implementation-defined;
function Get_TP_Class
   (Object : Connection_Parameters)
  return TP_Class;
procedure Set_TP_Class
   (Object : in out Connection_Parameters;
```

```
      To     : in     TP_Class);
   function Get_Extended_Format
      (Object : Connection_Parameters)
     return Boolean;
   procedure Set_Extended_Format
      (Object : in out Connection_Parameters;
       To     : in     Boolean);
   function Get_Expedited_Service
      (Object : Connection_Parameters)
     return Boolean;
   procedure Set_Expedited_Service
      (Object : in out Connection_Parameters;
       To     : in     Boolean);
   function Get_Negotiate_Checksums
      (Object : Connection_Parameters)
     return Boolean;
   procedure Set_Negotiate_Checksums
      (Object : in out Connection_Parameters;
       To     : in     Boolean);
   function Get_Signal_Disconnections
      (Object : Connection_Parameters)
     return Boolean;
   procedure Set_Signal_Disconnections
      (Object : in out Connection_Parameters;
       To     : in     Boolean);
   function Get_Protect_Parameters
      (Object : Connection_Parameters)
     return Boolean;
   procedure Set_Protect_Parameters
      (Object : in out Connection_Parameters;
       To     : in     Boolean);
   type TP_Network_Service is implementation-defined-integer;
   ISO_Connectionless          : constant TP_Network_Service
      := impl-def-static-expression;
   ISO_Connection              : constant TP_Network_Service
      := impl-def-static-expression;
   ISO_Connectionless_Over_X25 : constant TP_Network_Service
      := impl-def-static-expression;
   IP_Connectionless           : constant TP_Network_Service
      := impl-def-static-expression;
   function Get_Network_Service
      (Object : Connection_Parameters)
     return TP_Network_Service;
   procedure Set_Network_Service
      (Object : in out Connection_Parameters;
       To     : in     TP_Network_Service);
private
   implementation-defined
end POSIX_Sockets_ISO;
```

D.1.2.1 ISO Protocol Family

D.1.2.1.1 Synopsis

```
   ISO_Protocol           :
      constant POSIX_Sockets.Protocol_Family := impl-def-static-expression;
   ISO_Transport_Protocol :
```

```
   constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
Connectionless_Mode_Transport_Protocol :
   constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
Connectionless_Mode_Network_Protocol :
   constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
type ISO_Socket_Address is private;
type ISO_Socket_Address_Pointer is access all ISO_Socket_Address;
function "+" (Pointer : ISO_Socket_Address_Pointer)
  return POSIX_Sockets.Socket_Address_Pointer;
function "+" (Pointer : POSIX_Sockets.Socket_Address_Pointer)
  return ISO_Socket_Address_Pointer;
function Is_ISO_Socket_Address
   (Pointer : POSIX_Sockets.Socket_Address_Pointer)
  return Boolean;
type ISO_Address is new POSIX.Octet_Array;
type Presentation_Selector is new POSIX.Octet_Array;
type Session_Selector is new POSIX.Octet_Array;
type Transport_Selector is new POSIX.Octet_Array;
type GOSIP_Selector is new POSIX.Octet_Array;
function Get_ISO_Address (Name : ISO_Socket_Address)
  return ISO_Address;
procedure Set_ISO_Address
   (Name    : in out ISO_Socket_Address;
    Address : in     ISO_Address);
function Get_Presentation_Selector (Name : ISO_Socket_Address)
  return Presentation_Selector;
procedure Set_Presentation_Selector
   (Name     : in out ISO_Socket_Address;
    Selector : in     Presentation_Selector);
function Get_Session_Selector (Name : ISO_Socket_Address)
  return Session_Selector;
procedure Set_Session_Selector
   (Name     : in out ISO_Socket_Address;
    Selector : in     Session_Selector);
function Get_Transport_Selector (Name : ISO_Socket_Address)
  return Transport_Selector;
procedure Set_Transport_Selector
   (Name     : in out ISO_Socket_Address;
    Selector : in     Transport_Selector);
function Get_GOSIP_Selector (Name : ISO_Socket_Address)
  return GOSIP_Selector;
procedure Set_GOSIP_Selector
   (Name     : in out ISO_Socket_Address;
    Selector : in     GOSIP_Selector);
function Get_Socket_Name (Handle : POSIX_Sockets.Socket_Message)
  return ISO_Socket_Address;
function Get_Address (Info_Item : POSIX_Sockets.Socket_Address_Info)
  return ISO_Socket_Address;
function Get_Peer_Name (Socket : POSIX_IO.File_Descriptor)
  return ISO_Socket_Address;
function Get_Socket_Name (Socket : POSIX_IO.File_Descriptor)
  return ISO_Socket_Address;
```

D.1.2.1.2 Description

The ISO protocol family is a collection of protocols that uses the ISO address format. The ISO family provides protocol support for the `Sequenced_Packet_Socket` abstraction through the TP protocol, (ISO/IEC 8073 {4}), for the `Datagram_Socket` abstraction through the connectionless-mode transport protocol (*CLTP*, ISO/IEC 8602 {9}), and for the `Raw_Socket` abstraction by providing direct access to the connectionless-mode network protocol (*CLNP*, ISO/IEC 8473-1 {7}). These protocols are used when the `Protocol` parameter in the `Create` call is omitted and the `Domain` parameter has the value `ISO_Protocol`.

ISO addresses are based upon ISO/IEC 8348/AD2 {B4}. Endpoint addresses in the ISO protocol family utilize the `ISO_Socket_Address` object, which includes the following attributes.

ISO Address

The network address (NSAP) used by the endpoint shall be stored in this attribute. ISO network addresses are limited to 20 octets in length. ISO network addresses can take any format. `Set_ISO_Address` sets the ISO network address in an `ISO_Socket_Address` object. `Get_ISO_Address` returns the ISO address.

Presentation Selector

`Set_Presentation_Selector` sets the presentation selector in an `ISO_Socket_Address` object. `Get_Presentation_Selector` returns the current presentation selector. The presentation selector attribute is ignored for TP protocol sockets.

Session Selector

`Set_Session_Selector` sets the session selector in an `ISO_Socket_Address` object. `Get_Session_Selector` returns the session selector. The session selector attribute is ignored for TP protocol sockets.

Transport Selector

An ISO transport address is similar to an Internet address in that it contains a network address portion and a portion that the transport layer uses to multiplex its services among clients. In the ISO domain, the latter portion is called a transport selector (also known at one time as a transport suffix). Transport selectors may be of (almost) arbitrary size. `Set_Transport_Selector` sets the transport selector in an `ISO_Socket_Address` object. `Get_Transport_Selector` returns the transport selector.

GOSIP Selector

`Set_GOSIP_Selector` sets the GOSIP v2 selector in an `ISO_Socket_Address` object. `Get_GOSIP_Selector` returns the GOSIP v2 selector.

The type `ISO_Socket_Address` shall be used to represent an address for this protocol family. The type `ISO_Socket_Address_Pointer` is an access to this protocol-specific address type. The `"+"` operations shall convert a `ISO_Socket_Address_Pointer` to and from the `Socket_Address_Pointer` type for use with the base package operations defined for the `Socket_Address_Pointer` type. The return

value of the "+" operations designates the same address object as the input parameter. The function Is_ISO_Socket_Address shall return True if the address object designated by the specified non-null Socket_Address_Pointer is a valid ISO_Socket_Address and False otherwise. The conversion operation to ISO_Socket_Address_Pointer shall succeed if and only if the corresponding Is_ISO_Socket_Address returns True. Otherwise, the results are undefined.

NOTE: The Null_Socket_Address constant corresponds to the Ada null literal for these operations.

Get_Socket_Name shall return the name (address) associated with a socket. Get_Peer_Name shall return the socket address of the peer connected to a socket. For Get_Socket_Name and Get_Peer_Name, the Socket parameter is an open file descriptor referring to a socket. A call to Get_Socket_Name for a socket in the Ground state shall return the same value as that of a new socket address returned by Create.

Get_Address returns the Address attribute of a Socket_Address_Info object as an ISO_Socket_Address object. Get_Socket_Name returns the Name attribute of a Socket_Message object as an ISO_Socket_Address object.

D.1.2.1.3 Error Handling

If the following condition is detected, the exception POSIX_Error shall be raised with the corresponding error code:

Incorrect_Address_Type
: The type of the object designated by the return value is not appropriate for the address format of this socket.

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Bad_File_Descriptor
: The Socket parameter is not valid.

No_Buffer_Space
: Insufficient resources were available in the system to perform the operation.

Not_Connected
: The socket is not connected or otherwise has not had the peer prespecified.

Not_A_Socket
: The file descriptor Socket does not refer to a socket.

D.1.2.2 Connectionless-Mode ISO Sockets Protocols

D.1.2.2.1 Synopsis

```
type CL_Options is new POSIX.Octet_Array;
type CL_Flags is new POSIX.Option_Set;
No_Segmentation     : constant CL_Flags := implementation-defined;
Suppress_Error_PDUs : constant CL_Flags := implementation-defined;
No_Checksum         : constant CL_Flags := implementation-defined;
```

```
function Get_CL_Options
   (Socket : POSIX_IO.File_Descriptor)
  return CL_Options;
procedure Set_CL_Options
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in CL_Options);
function Get_CL_Flags
   (Socket : POSIX_IO.File_Descriptor)
  return CL_Flags;
procedure Set_CL_Flags
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in CL_Flags);
```

D.1.2.2.2 Description

The functionality described in this subclause is optional. If the OSI Connectionless option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

CLNP is the connectionless-mode network protocol used by the connectionless-mode network service. This protocol is specified in ISO/IEC 8473-1 {7}. It is accessible only through a socket of type `Raw_Socket`. CLNP sockets are connectionless-mode, and are normally used by `Send` with the optional `To` parameter and `Receive` with the optional `From` parameter, though `Connect` may also be used to fix the destination for future packets (in which case `Read` or `Receive` and `Write` or `Send` may be used).

Outgoing packets automatically have a CLNP header prefixed to them. Incoming packets received by the application contain the full CLNP header.

CLTP is a simple unreliable datagram protocol that is accessed via the `Datagram_Socket` abstraction for the ISO protocol family. CLTP sockets are connectionless mode and are normally used with `Send` with the optional `To` parameter and `Receive` with the optional `From` parameter. However, `Connect` may also be used to fix the destination for future packets (in which case the `Receive` or `Read`, and `Send` or `Write` may be used).

CLTP address formats are identical to those used by TP. In particular CLTP provides a service selector in addition to the normal ISO NSAP. The CLTP selector space is separate from the TP selector space (*i.e.*, a CLTP selector may not be connected to a TP selector).

ISO CLNP and ISO CLTP support all states in Figure D.3.

`Set_CL_Flags` sets CLNP and CLTP option flags. `Get_CL_Flags` returns current CLNP and CLTP option flags. The option flags are represented as a set of symbols of type `POSIX.Option_Set`, using the following names.

`No_Segmentation`
: Do not allow segmentation if this option is enabled.

`Suppress_Error_PDUs`
: Suppress error PDUs if this option is enabled.

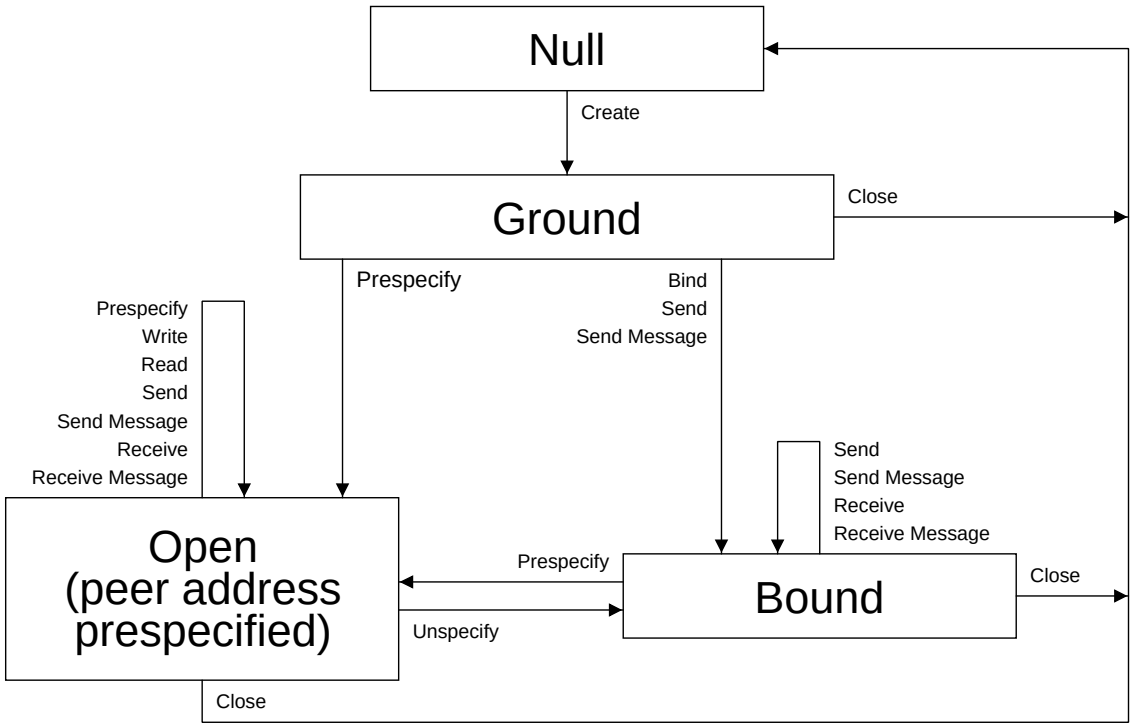

Figure D.3 – ISO (and UDP) Connectionless Sockets State Diagram

`No_Checksum`

Do not generate the checksum if this option is enabled.

The operations `"+"`, `"-"`, `">"`, `"<"`, `">="`, `"<="`, and `Empty_Set` are available on the type `CL_Flags` via the derived type semantics of Ada, from the operations available for `POSIX.Option_Set`. The appropriate operations can be used to create and examine a set containing the required flags.

`Set_CL_Options` sets the connectionless mode protocol options. The options must be formatted exactly as specified by 7.5 of ISO/IEC 8473-1 {7}. Once an option has been set, it will be sent on all packets until a different option is set. `Get_CL_Options` returns current options. When a packet is received with the globally unique quality of service option present in the `CL_Options` object and the congestion experienced bit is set, then the transport congestion control procedure defined in ISO/IEC 8073 {4} is called.

D.1.2.2.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Is_Already_Connected`

The socket is already connected.

`Not_Connected`

The socket has not had the peer prespecified.

No_Buffer_Space

Insufficient memory available in the system for an internal data structure.

Address_In_Use

The specified address is already in use.

Address_Not_Available

No network interface exists for the specified network address.

D.1.2.3 TP - ISO Transport Protocol

D.1.2.3.1 Synopsis

```
type TP_Flags is new POSIX.Option_Set;
Public_Data_Network_QOS : constant TP_Flags := implementation-defined;
Peer_On_Same_Network    : constant TP_Flags := implementation-defined;
Expedited_Data_Present  : constant TP_Flags := implementation-defined;
function Get_TP_Flags
   (Socket : POSIX_IO.File_Descriptor)
  return TP_Flags;
function Get_Connection_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.Octet_Array;
procedure Set_Connection_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.Octet_Array);
function Get_Disconnect_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.Octet_Array;
procedure Set_Disconnect_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.Octet_Array);
function Get_Confirmation_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX.Octet_Array;
procedure Set_Confirmation_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX.Octet_Array);
type TP_Ancillary_Data_Type is
   (Connection_Data, Disconnect_Data, Confirmation_Data);
type TP_Ancillary_Data (Kind : TP_Ancillary_Data_Type;
                        Size : Positive) is private;
procedure Set_Ancillary_Data
   (Message : in out POSIX_Sockets.Socket_Message;
    Object  : in     TP_Ancillary_Data);
function Get_Ancillary_Data
   (Message : POSIX_Sockets.Socket_Message)
  return TP_Ancillary_Data;
procedure Set_Ancillary_Data_Array
   (Object : in out TP_Ancillary_Data;
    Data   : in     POSIX.Octet_Array);
function Get_Ancillary_Data_Array
   (Object : TP_Ancillary_Data)
  return POSIX.Octet_Array;
type Connection_Parameters is private;
function Get_Connection_Parameters
   (Socket : POSIX_IO.File_Descriptor)
  return Connection_Parameters;
```

```
procedure Set_Connection_Parameters
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Connection_Parameters);
function Get_Retransmit_Number
   (Object : Connection_Parameters)
  return Natural;
procedure Set_Retransmit_Number
   (Object : in out Connection_Parameters;
    To     : in     Natural);
type Window_Size is range 128 .. 16384;
function Get_Window_Size
   (Object : Connection_Parameters)
  return Window_Size;
procedure Set_Window_Size
   (Object : in out Connection_Parameters;
    To     : in     Window_Size);
type TPDU_Size is range 7 .. 13;
function Get_TPDU_Size
   (Object : Connection_Parameters)
  return TPDU_Size;
procedure Set_TPDU_Size
   (Object : in out Connection_Parameters;
    To     : in     TPDU_Size);
type TP_Acknowledgment_Strategy is
   (Acknowledge_Window, Acknowledge_Each);
function Get_Acknowledgment_Strategy
   (Object : Connection_Parameters)
  return TP_Acknowledgment_Strategy;
procedure Set_Acknowledgment_Strategy
   (Object : in out Connection_Parameters;
    To     : in     TP_Acknowledgment_Strategy);
type TP_Retransmit_Strategy is
   (Retransmit_Each_Packet, Use_Congestion_Window, Fast_Start);
function Get_Retransmit_Strategy
   (Object : Connection_Parameters)
  return TP_Retransmit_Strategy;
procedure Set_Retransmit_Strategy
   (Object : in out Connection_Parameters;
    To     : in     TP_Retransmit_Strategy);
type TP_Class is new POSIX.Option_Set;
TP_Class_0 : constant TP_Class := implementation-defined;
TP_Class_1 : constant TP_Class := implementation-defined;
TP_Class_2 : constant TP_Class := implementation-defined;
TP_Class_3 : constant TP_Class := implementation-defined;
TP_Class_4 : constant TP_Class := implementation-defined;
function Get_TP_Class
   (Object : Connection_Parameters)
  return TP_Class;
procedure Set_TP_Class
   (Object : in out Connection_Parameters;
    To     : in     TP_Class);
function Get_Extended_Format
   (Object : Connection_Parameters)
  return Boolean;
procedure Set_Extended_Format
   (Object : in out Connection_Parameters;
    To     : in     Boolean);
function Get_Expedited_Service
   (Object : Connection_Parameters)
  return Boolean;
```

```
procedure Set_Expedited_Service
   (Object : in out Connection_Parameters;
    To     : in     Boolean);
function Get_Negotiate_Checksums
   (Object : Connection_Parameters)
  return Boolean;
procedure Set_Negotiate_Checksums
   (Object : in out Connection_Parameters;
    To     : in     Boolean);
function Get_Signal_Disconnections
   (Object : Connection_Parameters)
  return Boolean;
procedure Set_Signal_Disconnections
   (Object : in out Connection_Parameters;
    To     : in     Boolean);
function Get_Protect_Parameters
   (Object : Connection_Parameters)
  return Boolean;
procedure Set_Protect_Parameters
   (Object : in out Connection_Parameters;
    To     : in     Boolean);
type TP_Network_Service is implementation-defined-integer;
ISO_Connectionless          : constant TP_Network_Service
   := impl-def-static-expression;
ISO_Connection              : constant TP_Network_Service
   := impl-def-static-expression;
ISO_Connectionless_Over_X25 : constant TP_Network_Service
   := impl-def-static-expression;
IP_Connectionless           : constant TP_Network_Service
   := impl-def-static-expression;
function Get_Network_Service
   (Object : Connection_Parameters)
  return TP_Network_Service;
procedure Set_Network_Service
   (Object : in out Connection_Parameters;
    To     : in     TP_Network_Service);
```

D.1.2.3.2 Description

The functionality described in this subclause is optional. If the OSI Connection option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

The TP protocol provides reliable, flow-controlled, two-way transmission of data and record boundaries. It is an octet-stream protocol and is accessed according to the `Sequenced_Packet_Socket` abstraction.

Sockets utilizing the TP protocol are either active or passive. Active sockets initiate connections to passive sockets. By default TP sockets are created active; to create a passive socket, `Listen` must be used after binding the socket with `Bind`.

Only passive sockets may use `Accept_Connection` to accept incoming connections. Only active sockets may use `Connect` to initiate connections.

ISO TP supports all the states in Figure 18.1.

The TP protocol makes use of a standard ISO address format, including a NSAP (see 2.2.3.44) and a transport service entity selector. Class 4 may also make use of the Internet address family.

Passive sockets may underspecify their location to match incoming connection requests sent to multiple network addresses that refer to the same end system. This technique, termed *wildcard addressing*, allows a single server to provide service to clients on multiple networks. To create a socket that listens on multiple network addresses, the NSAP portion of the bound address shall not have been set via `Set_ISO_Address`. The Transport Selector may still be specified at this time via `Set_Transport_Selector`; if the Transport Selector is not specified the system will assign one. Once a connection has been established, the address of the socket is fixed by the location of the peer entity. The network address assigned to the socket is an address associated with the network interface through which packets are being transmitted and received.

If the EOT (*i.e.*, end of transmission) SDU is not needed, the normal `Read` and `Write` may be used.

If the TP entity encounters asynchronous events that cause a transport connection to be closed, such as timing out while retransmitting a connect request TPDU, or receiving a DR TPDU, the TP entity issues a `POSIX_Signals.Signal_Out_Of_Band_Data` signal, indicating that disconnection has occurred. If the signal is issued during a system call, the system call may be interrupted, in which case `POSIX_Error` is raised with error code `Interrupted_Operation`. If the signal `POSIX_Signals.Signal_Out_Of_Band_Data` is being handled by reading from the socket and an `Accept_Connection` timed out, the read may result in error code `Not_A_Socket` because the `Accept_Connection` function had not yet returned a socket descriptor when the signal was handled. `Timed_Out` (or some other value appropriate to the type of error) is returned if `POSIX_Signals.Signal_Out_Of_Band_Data` is blocked for the duration of the system call.

An application program should take one of the following approaches: If the program is servicing only one connection, it can block or ignore `POSIX_Signals.Signal_Out_Of_Band_Data` during connection establishment. The advantage of this approach is that the error code returned by the exception is somewhat meaningful. The disadvantage of this approach is that if ignored, disconnection and expedited data indications could be missed. For some programs missed disconnection and expedited data indications are not a problem. If the program is servicing more than one connection at a time, or expedited data have arrived, or both, the program may elect to service `POSIX_Signals.Signal_Out_Of_Band_Data`. It can use the `Get_TP_Flags` function to see whether the signal was due to the arrival of expedited data (`Expedited_Data_Present` is set in `TP_Flags`) or due to a disconnection (the error `Not_Connected` is generated).

TP supports several options to control such things as negotiable options in the protocol and protocol strategies.

In the following list of options, the Disconnect Data socket option and the Confirmation Data socket option may be set after a connection is established. Other options must be used before the connection is established, in other words, before calling `Connect`

or Accept_Connection. All options may be examined before or after a connection is established.

Some of the options in the following list may be sent and received as socket message ancillary data. The Set_Ancillary_Data procedure prepares a socket message to send or receive ancillary data. The Get_Ancillary_Data function retrieves the ancillary data as an object of type TP_Ancillary_Data. This object is a composite type with a discriminant of type TP_Ancillary_Data_Type to define the category of the ancillary data, and another discriminant that specifies the size in octets of the ancillary data. The Set_Ancillary_Data_Array procedure and the Get_Ancillary_Data_Array function manipulate an Octet_Array object that contains the ancillary data.

Connection Data

Data associated with a connection request. The value shall be an array of octets as sent or received by the protocol implementation. Get_Connection_Data may be used to retrieve the application data of a connection request on a passive socket, after having done Accept_Connection without implying confirmation of the connection. The data may also be retrieved by issuing a Receive_Message request for ancillary data only, without implying confirmation of the connection (see 18.4.1.3). The ancillary data are specified with a TP_Ancillary_Data_Type of Connection_Data.

Disconnect Data

Data associated with a disconnect. The value shall be an array of octets as sent or received by the protocol implementation. Disconnect data may be sent by the side initiating the close, but not by the passive side (passive with respect to the closing of the connection). Therefore, there is no need to read disconnect data after calling Close. Disconnect data may be sent by Set_Disconnect_Data or by issuing a Send_Message request specifying ancillary data only. The ancillary data are specified with a TP_Ancillary_Data_Type of Disconnect_Data.

Confirmation Data

Data associated with connection confirmation. The value shall be an array of octets as sent or received by the protocol implementation. Confirmation data may also be sent by a Set_Confirmation_Data or by issuing a Send_Message request for ancillary data only. The ancillary data are specified with a TP_Ancillary_Data_Type of Disconnect_Data. Sending of connect confirm data will cause the connection to be confirmed rather than rejected.

TP Flags

Get_TP_Flags returns the value of the following options as type TP_Flags. The operations "+", "-", ">", "<", ">=", "<=", and Empty_Set are available on the type TP_Flags via the derived type semantics of Ada, from the operations available for POSIX.Option_Set. The appropriate operations can be used to create and examine a set containing the following flags.

Public_Data_Network_QOS

Set when the quality of the network service is similar to that of a public data network.

Peer_On_Same_Network
: Set when the peer TP entity is considered to be on the same network as the local TP entity.

Expedited_Data_Present
: Set when expedited data are present.

Connection Parameters
: A group of parameters to be gotten or set for a connection. The value shall be an object of type Connection_Parameters, which may be examined and set by the operations Get_Connection_Parameters and Set_Connection_Parameters.

The object Connection_Parameters is the parameter used with Get_Connection_Parameters and Set_Connection_Parameters for the Connection Parameters socket option. The attributes of the Connection_Parameters object shall be accessed by the corresponding "Get_" and "Set_" functions and procedures defined in package POSIX_Sockets_ISO. The Connection_Parameters object shall contain at least the following attributes:

Retransmit Number
: The number of times a TPDU will be retransmitted before the local TP entity closes a connection. The default value is 6.

Window Size
: The buffer space limits in octets for incoming and outgoing data. There is no way to specify different limits for incoming and outgoing paths. The actual window size at any time during the lifetime of a connection is a function of the buffer size limit, the negotiated maximum TPDU size, and the rate at which the application program receives data. This parameter applies only to Class 4. The value must be between 128 and 16384. The default is 4096 octets.

TPDU Size
: A value between 7 and 13 specifying the logarithm (base 2) of the maximum TPDU size to be negotiated. The default is 12 for Class 4 and 11 for Class 0. The TP standard (ISO/IEC 8473-1 {7}) gives an upper bound of 13 for Class 4 and 11 for Class 0.

Acknowledgment Strategy
: This parameter applies only to Class 4. Two acknowledgment strategies are supported: Acknowledge_Each means each data TPDU is acknowledged with an AK TPDU. Acknowledge_Window means that upon receipt of the packet that represents the high edge of the last window advertised, and AK TPDU is generated. The default value is Acknowledge_Window.

Retransmit Strategy
: The default value is either Use_Congestion_Window or Fast_Start over connectionless network protocols. The default is Use_Congestion_Window over connection oriented network protocols. This parameter applies only to Class 4.

  Retransmit_Each_Packet means when a retransmission timer expires, to retransmit each packet in the send window rather than just the first unacknowledged packet. Use_Congestion_Window means to use a congestion

window strategy borrowed from Van Jacobson's congestion window strategy for TCP {B10}. The congestion window size is set to one whenever retransmission occurs.

`Fast_Start` means to begin sending the maximum amount of data permitted by the peer (subject to availability). The alternative is to start sending slowly by pretending the window of the peer is smaller than it is and letting it slowly grow up to the real window size of the peer, which is intended to smooth the effect of new connections on a congested network by preventing a transport connection from suddenly overloading the network with a burst of packets. This strategy is also due to Van Jacobson {B10}.

TP Class

This attribute is an `Option_Set` for the class. The set includes one or both of the values `TP_Class_4` and `TP_Class_0`. The higher class indicated is the preferred class. If only one class is indicated, negotiation will not occur during connection establishment. The default is `TP_Class_4` or `TP_Class_0`.

Extended Format

This attribute indicates that extended format shall be negotiated. This attribute applies only to Class 4. The default is `False`.

Expedited Service

This attribute indicates that the expedited data transport service will be negotiated. This attribute applies only to Class 4. The default is `True`.

Negotiate Checksums

This attribute indicates the the use of checksums will be negotiated. This attribute applies only to Class 4. The default value is `True`.

Signal Disconnections

This attribute indicates that the local TP entity shall issue indications (signals) when a TP connection is disconnected. The default value is `True`.

Protect Parameters

If `True`, the TP entity will not override any of the other values given in this object. If the values cannot be used, the TP entity will drop, disconnect, or refuse to establish the connection to which this object pertains. The default value is `False`.

Network Service

This attribute indicates which network service is to be used. `ISO_Connectionless` indicates the connectionless network service provided by CLNP (ISO/IEC 8473-1 {7}). `ISO_Connection` indicates the connection-oriented network service provided by X.25 (ISO/IEC 8208 {5}) and ISO/IEC 8878 {10}. `ISO_Connectionless_Over_X25` indicates the connectionless network service running over a connection-oriented subnetwork service: CLNP (ISO/IEC 8473-3 {8}) over X.25 (ISO/IEC 8208 {5}). `IP_Connectionless` indicates the Internet connectionless network service provided by IP (RFC 791 {13}). The default value is `ISO_Connectionless`.

D.1.2.3.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Is_Already_Connected`

The socket is already connected.

`Not_Connected`

The socket has not had the peer prespecified.

The `Get_TP_Flags` function was called, indicating that a signal was due to a disconnection.

D.1.3 Package `POSIX_Sockets_Internet`

This package provides the DNI/Socket interface mappings for Internet transport protocols. Unless otherwise specified, all the facilities in package `POSIX_Sockets` can be used. Only additional information relevant to the Internet transport protocol is highlighted here.

The functionality described in this subclause is optional. If either the Sockets Detailed Network Interface option or the Internet Protocol option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this subclause.

```
with POSIX,
     POSIX_IO,
     POSIX_Sockets;
package POSIX_Sockets_Internet is
   -- D.1.3.1 Inet - IP Protocol Family
   Internet_Protocol :
      constant POSIX_Sockets.Protocol_Family := impl-def-static-expression;
   ICMP : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
   TCP  : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
   UDP  : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
   Raw  : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
   type Internet_Socket_Address is private;
   type Internet_Socket_Address_Pointer is access all Internet_Socket_Address;
   function "+" (Pointer : Internet_Socket_Address_Pointer)
     return POSIX_Sockets.Socket_Address_Pointer;
   function "+" (Pointer : POSIX_Sockets.Socket_Address_Pointer)
     return Internet_Socket_Address_Pointer;
   function Is_Internet_Socket_Address
      (Pointer : POSIX_Sockets.Socket_Address_Pointer)
     return Boolean;
   type Internet_Port is implementation-defined-integer;
   Unspecified_Internet_Port : constant Internet_Port;
   function Get_Internet_Port (Name : Internet_Socket_Address)
     return Internet_Port;
   procedure Set_Internet_Port
      (Name : in out Internet_Socket_Address;
       Port : in     Internet_Port);
   type Internet_Address is private;
   Unspecified_Internet_Address : constant Internet_Address;
   Loopback_Internet_Address    : constant Internet_Address;
```

```
Broadcast_Internet_Address   : constant Internet_Address;
function Get_Internet_Address (Name : Internet_Socket_Address)
  return Internet_Address;
procedure Set_Internet_Address
   (Name    : in out Internet_Socket_Address;
    Address : in     Internet_Address);
function Get_Socket_Name (Handle : POSIX_Sockets.Socket_Message)
  return Internet_Socket_Address;
function Get_Address (Info_Item : POSIX_Sockets.Socket_Address_Info)
  return Internet_Socket_Address;
function Get_Peer_Name (Socket : POSIX_IO.File_Descriptor)
  return Internet_Socket_Address;
function Get_Socket_Name (Socket : POSIX_IO.File_Descriptor)
  return Internet_Socket_Address;
-- D.1.3.2 Internet Address Support Functions
-- Internet Address Manipulation
function String_To_Internet_Address (Address : POSIX.POSIX_String)
  return Internet_Address;
function Is_Internet_Address (Address : POSIX.POSIX_String)
  return Boolean;
function Internet_Address_To_String (Address : Internet_Address)
  return POSIX.POSIX_String;
-- Network Database Functions
type Network_Info is private;
type Network_Number is range implementation-defined;
Unspecified_Network_Number : constant Network_Number;
type Database_Array is new POSIX.Octet_Array;
type Database_Array_Pointer is access all Database_Array;
function Get_Name (Info_Item : Network_Info)
  return POSIX.POSIX_String;
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Network_Alias (Info_Item : Network_Info);
function Get_Family (Info_Item : Network_Info)
  return POSIX_Sockets.Protocol_Family;
function Get_Network_Number (Info_Item : Network_Info)
  return Network_Number;
function Get_Network_Info_By_Address
   (Number  : Network_Number;
    Family  : POSIX_Sockets.Protocol_Family;
    Storage : Database_Array_Pointer)
  return Network_Info;
function Get_Network_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Network_Info;
procedure Open_Network_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Network_Database_Connection;
-- Network Protocol Database Functions
type Protocol_Info is private;
function Get_Name (Info_Item : Protocol_Info)
  return POSIX.POSIX_String;
```

```
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Protocol_Alias (Info_Item : Protocol_Info);
function Get_Protocol_Number (Info_Item : Protocol_Info)
  return POSIX_Sockets.Protocol_Number;
function Get_Protocol_Info_By_Number
   (Number  : POSIX_Sockets.Protocol_Number;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
function Get_Protocol_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
procedure Open_Protocol_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Protocol_Database_Connection;
-- D.1.3.4 Internet Transmission Control Protocol
subtype Keep_Alive_Time is POSIX.Seconds range 1 .. POSIX.Seconds'Last;
function Get_Keep_Alive_Interval
   (Socket : POSIX_IO.File_Descriptor)
  return Keep_Alive_Time;
procedure Set_Keep_Alive_Interval
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Keep_Alive_Time);
function Get_No_Delay
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_No_Delay
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
subtype Socket_Retransmit_Time is POSIX.Seconds range
   implementation-defined;
Wait_Forever : constant Socket_Retransmit_Time := impl-def-static-expression;
Retransmit_Time_Default
            : constant Socket_Retransmit_Time := impl-def-static-expression;
function Get_Retransmit_Time_Maximum
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Retransmit_Time;
procedure Set_Retransmit_Time_Maximum
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Retransmit_Time);
function Get_Segment_Size_Maximum
   (Socket : POSIX_IO.File_Descriptor)
  return Positive;
function Get_Standardized_Urgent_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_Standardized_Urgent_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
-- D.1.3.6 Internet Protocol
function IP_Header_Options_In_Use
   (Socket : POSIX_IO.File_Descriptor)
  return Boolean;
procedure Reset_IP_Header_Options
   (Socket : in POSIX_IO.File_Descriptor);
```

```
type IP_Options_Buffer is private;
function Get_IP_Header_Options
   (Socket : POSIX_IO.File_Descriptor)
  return IP_Options_Buffer;
procedure Set_IP_Header_Options
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in IP_Options_Buffer);
function Get_First_Hop
   (Options : IP_Options_Buffer)
  return Internet_Address;
procedure Set_First_Hop
   (Options : in out IP_Options_Buffer;
    Address : in     Internet_Address);
function Get_IP_Options
   (Options : IP_Options_Buffer)
  return POSIX.Octet_Array;
procedure Set_IP_Options
   (Options : in out IP_Options_Buffer;
    Buffer  : in     POSIX.Octet_Array);
type IP_Type_Of_Service is private;
Low_Delay        : constant IP_Type_Of_Service;
High_Throughput  : constant IP_Type_Of_Service;
High_Reliability : constant IP_Type_Of_Service;
Unspecified      : constant IP_Type_Of_Service;
type Time_To_Live is range 0 .. 255;
function Get_Type_Of_Service
   (Socket : POSIX_IO.File_Descriptor)
  return IP_Type_Of_Service;
procedure Set_Type_Of_Service
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in IP_Type_Of_Service);
function Get_Initial_Time_To_Live
   (Socket : POSIX_IO.File_Descriptor)
  return Time_To_Live;
procedure Set_Initial_Time_To_Live
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Time_To_Live);
function Get_Receive_Destination_Address
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_Receive_Destination_Address
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
type IP_Ancillary_Data is private;
type IP_Ancillary_Data_Pointer is access all IP_Ancillary_Data;
procedure Set_Ancillary_Data
   (Message : in out POSIX_Sockets.Socket_Message;
    Data    : in     IP_Ancillary_Data_Pointer);
function Get_Destination_Address
   (Data : IP_Ancillary_Data)
  return Internet_Address;
function Get_Header_Included
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_Header_Included
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
```

```
private
   implementation-defined
end POSIX_Sockets_Internet;
```

D.1.3.1 Inet - IP Protocol Family

D.1.3.1.1 Synopsis

```
Internet_Protocol :
   constant POSIX_Sockets.Protocol_Family := impl-def-static-expression;
ICMP : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
TCP  : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
UDP  : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
Raw  : constant POSIX_Sockets.Protocol_Number := impl-def-static-expression;
type Internet_Socket_Address is private;
type Internet_Socket_Address_Pointer is access all Internet_Socket_Address;
function "+" (Pointer : Internet_Socket_Address_Pointer)
  return POSIX_Sockets.Socket_Address_Pointer;
function "+" (Pointer : POSIX_Sockets.Socket_Address_Pointer)
  return Internet_Socket_Address_Pointer;
function Is_Internet_Socket_Address
   (Pointer : POSIX_Sockets.Socket_Address_Pointer)
  return Boolean;
type Internet_Port is implementation-defined-integer;
Unspecified_Internet_Port : constant Internet_Port;
function Get_Internet_Port (Name : Internet_Socket_Address)
  return Internet_Port;
procedure Set_Internet_Port
   (Name : in out Internet_Socket_Address;
    Port : in     Internet_Port);
type Internet_Address is private;
Unspecified_Internet_Address : constant Internet_Address;
Loopback_Internet_Address    : constant Internet_Address;
Broadcast_Internet_Address   : constant Internet_Address;
function Get_Internet_Address (Name : Internet_Socket_Address)
  return Internet_Address;
procedure Set_Internet_Address
   (Name    : in out Internet_Socket_Address;
    Address : in     Internet_Address);
function Get_Socket_Name (Handle : POSIX_Sockets.Socket_Message)
  return Internet_Socket_Address;
function Get_Address (Info_Item : POSIX_Sockets.Socket_Address_Info)
  return Internet_Socket_Address;
function Get_Peer_Name (Socket : POSIX_IO.File_Descriptor)
  return Internet_Socket_Address;
function Get_Socket_Name (Socket : POSIX_IO.File_Descriptor)
  return Internet_Socket_Address;
```

D.1.3.1.2 Description

The IP family, designated by the constant `Internet_Protocol`, is a collection of protocols layered atop the IP transport layer, and utilizing the Internet address format, `Internet_Address`. The Internet family provides protocol support for the `Stream_Socket`, `Datagram_Socket`, and `Raw_Socket` socket types; the `Raw_Socket` interface provides access to the Internet protocol.

The following constants identify specific Internet protocols. These constants shall be defined to the values used in the IP header protocol identification field when sending a packet for that protocol. These constants may be used to identify the indicated protocols for the `Create` operation. This list also indicates the default protocols when a socket is created in the Internet family with the `Protocol` parameter omitted or set to `Default_Protocol`. (The default protocol for sockets created with type `Sequenced_Packet_Socket` is unspecified.)

`ICMP`

Protocol number for ICMP.

`TCP`

Protocol number for TCP. This protocol is the default for sockets created with type `Stream_Socket`.

`UDP`

Protocol number for UDP. This protocol is the default for sockets created with type `Datagram_Socket`.

`Raw`

Default Protocol number for raw IP packets. This protocol is the default for sockets created with type `Raw_Socket`.

A raw interface to the Internet protocol is available by creating an Internet socket of type `Raw_Socket`. The default protocol for type `Raw_Socket` shall be identified in the IP header with the value `Raw`. Applications should not use the default protocol when creating a socket with type `Raw_Socket`, but should identify a specific protocol by value. The ICMP control protocol is accessible from a raw socket by specifying a value of `ICMP` for protocol.

The format of Internet addresses is defined in RFC 791 {13}. The representations of Internet addresses, network identifiers, and host identifiers shall be implementation defined. Internet socket addresses are represented by an object of type `Internet_Socket_Address`.

A value of type `Internet_Port` forms a component of the socket address for Internet sockets. Given an Internet socket address, `Get_Internet_Port` returns the corresponding Internet Port. This value is set to `Unspecified_Internet_Port` when a socket address is created and needs to be set to a valid port number by the application. `Set_Internet_Port` sets the Internet Port value in an Internet socket address.

A value of type `Internet_Address` forms a component of the socket address for Internet sockets. Given an Internet socket address, `Get_Internet_Address` returns the corresponding Internet address. This value is set to `Unspecified_Internet_Address` when a socket address is created and must set to a valid address by the application. `Set_Internet_Address` sets the Internet address value in an Internet socket address.

NOTE: Objects of type `Internet_Address` and `Internet_Port` shall be in host byte order. Any required conversion of these objects to network byte order shall be performed by the implementation of this binding.

The Internet Address attribute of the `Internet_Socket_Address` object indicates an Internet host address, which might be an individual host address, a group address, or

an unspecified address (indicated by the value Unspecified_Internet_Address). Every Internet host has one or more Internet addresses that are considered to be local to that system. The Internet_Port attribute of the Internet_Socket_-Address object indicates a protocol port number. An endpoint is identified by an Internet host address, a port number, and the protocol; thus, the same port number may be used by more than one protocol without conflict.

Sockets in the Internet family shall be created with unspecified local and remote (peer) host addresses and port numbers. An unspecified host address shall be represented by the value Unspecified_Internet_Address; an unspecified port number shall be represented by the constant Unspecified_Internet_Port.

An application may use the Bind procedure to bind its local address or port number or both. The Bind procedure shall be called with a Socket_Address_Pointer parameter with the format of a Internet_Socket_Address object. If the Internet Port attribute of the object has the value zero, the implementation shall choose an unused port number. If the Internet Address attribute contains the value Unspecified_Internet_Address, the local address for the socket shall remain unspecified. Sockets with the local address Unspecified_Internet_Address shall receive datagrams or connection requests sent to any Internet address for the local host system if no socket with the same protocol and port number is bound to the specific address to which the incoming datagram or connection request is sent. This technique, termed *wildcard addressing*, allows a single server to provide service to clients connecting to any address of a *multihomed* host (a host with multiple IP addresses). If a call to the Bind procedure specifies an address other than Unspecified_Internet_Address that is not an address for the local system, the error Address_Not_Available will result.

When an attempt is made to bind a socket address to a socket with a type other than Raw_Socket and the port component of the specified socket address is already in use by another socket using the same protocol, the result shall be as shown in Table D.1. In that table,

— "First Socket Address" is the address of an existing socket using the same protocol as the socket specified in the call. It is the address to which the socket is bound or Unspecified_Internet_Address if neither Bind nor Connect has been called successfully, or if Bind has been called successfully with an address of Unspecified_Internet_Address and there has been no subsequent connection, or if a socket has been disconnected (for UDP, by connecting to an empty address).

— "Second Socket Bind" refers to the socket specified in the Bind call.

— *Address 1* and *Address 2* represent two distinct local IP addresses.

— The last two columns specify the result depending on the value of the Socket Reuse Addresses socket option for the second socket.

If multiple sockets using the same protocol are bound to the specified port, the attempt shall fail if, for any of those sockets, an error is indicated according to Table D.1.

The Socket Reuse Addresses socket option allows a local port number to be reused only when the port number is in use only as the local port of connected sockets. In

other words, each other socket with the same local port shall have specified local and remote addresses and remote port number. This option allows servers to terminate and restart their listening socket even when connections exist for their service. All servers that listen on a specific port should issue this option to allow graceful restart. A port can not be reused if it is already in use on an unconnected socket (a socket that does not have the remote host address specified); therefore, two different servers cannot both listen on the same port and produce conflicting results for applications.

NOTE: This restriction combines with the way addresses are specified (in order to make a connection from a specified port, the local port must first be specified via `Bind`*) to create a subtle race condition. This race condition occurs when two processes are both trying to connect from a specific local port to different remote ports. Only one process at a time can have a bound but unconnected socket. If two processes both try to bind to the local port and then create a connection, one process will succeed and the other process will fail. Because of this race condition, bind operations that return an error should be retried a number of times.*

If an Internet socket for which the `Bind` procedure has not been called successfully is used in a successful call to the `Send` or `Send_Message` functions, the implementation shall bind an unused local port to the socket. If an Internet socket for which the `Bind` procedure has not been called is used in a successful call to the `Connect` procedure, the implementation shall bind an unused local port to the socket and shall bind an address for the local system to the socket.

The destination address in a call to the `Connect`, `Send`, or `Send_Message` functions shall have the format of a `Internet_Socket_Address` object. The host address may be specified as `Unspecified_Internet_Address` to mean the host where the application is currently executing, and shall be interpreted as if it were one of the Internet addresses for the local system. The distinguished address `Broadcast_Internet_Address` shall be interpreted as if it were the broadcast address on the primary network if the first network configured supports broadcast (see RFC 791 {13}).

If the local address of a socket is `Unspecified_Internet_Address` at the time of a call to the `Send` or `Send_Message` procedures, the implementation shall chose a local host address as the source address for any packets sent. If the local address of a socket is `Unspecified_Internet_Address` at the time of a call to the `Connect` procedure, the implementation shall bind an address for the local system to the socket.

The type `Internet_Socket_Address` shall be used to represent an address for this protocol family. The type `Internet_Socket_Address_Pointer` is an access to this protocol-specific address type. The `"+"` operations shall convert a `Internet_Socket_Address_Pointer` to and from the `Socket_Address_Pointer` type for use with the base package operations defined for the `Socket_Address_Pointer` type. The return value of the `"+"` operations designates the same address object as the input parameter. The function `Is_Internet_Socket_Address` shall return `True` if the address object designated by the specified non-null `Socket_Address_Pointer` is a valid `Internet_Socket_Address` and `False` otherwise. The conversion operation to `Internet_Socket_Address_Pointer` shall succeed if and only if the corresponding `Is_Internet_Socket_Address` returns `True`. Otherwise, the results are undefined.

NOTE: The `Null_Socket_Address` *constant corresponds to the Ada* `null` *literal for these operations.*

Table D.1 – Port Number Re-Use

First Socket Address	First Socket Connected	Second Socket Bind	Second Bind Result Based on Socket Reuse Addresses Option	
			OFF	ON
Address 1	NO	Address 1	ERROR	ERROR
Address 1	NO	Address 2	OK	OK
Address 1	NO	Unspecified_-Internet_Address	ERROR	OK
Unspecified_-Internet_Address	NO	Address 1	ERROR	OK
Unspecified_-Internet_Address	NO	Unspecified_-Internet_Address	ERROR	ERROR
Address 1	YES	Address 1	ERROR	OK
Address 1	YES	Address 2	OK	OK
Address 1	YES	Unspecified_-Internet_Address	ERROR	OK

`Get_Socket_Name` shall return the name (address) associated with a socket. `Get_-Peer_Name` shall return the socket address of the peer connected to a socket. For `Get_Socket_Name` and `Get_Peer_Name`, the `Socket` parameter is an open file descriptor referring to a socket. A call to `Get_Socket_Name` for a socket in the Ground state shall return the same value as that of a new socket address returned by `Create`.

`Get_Address` returns the Address attribute of a `Socket_Address_Info` object as an `Internet_Socket_Address` object. `Get_Socket_Name` returns the Name attribute of a `Socket_Message` object as an `Internet_Socket_Address` object.

D.1.3.1.3 Error Handling

If the following condition is detected, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Incorrect_Address_Type`
: The type of the object designated by the return value is not appropriate for the address format of this socket.

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Bad_File_Descriptor`
: The `Socket` parameter is not valid.

`No_Buffer_Space`
: Insufficient resources were available in the system to perform the operation.

`Not_Connected`
: The socket is not connected or otherwise has not had the peer prespecified.

`Not_A_Socket`
: The file descriptor `Socket` does not refer to a socket.

`Address_Not_Available`

A call to `Bind` specified an address other than `Unspecified_Internet_Address` that is not an address for the local system.

D.1.3.2 Internet Address Support Functions

D.1.3.2.1 Synopsis

```
-- Internet Address Manipulation
function String_To_Internet_Address (Address : POSIX.POSIX_String)
  return Internet_Address;
function Is_Internet_Address (Address : POSIX.POSIX_String)
  return Boolean;
function Internet_Address_To_String (Address : Internet_Address)
  return POSIX.POSIX_String;
-- Network Database Functions
type Network_Info is private;
type Network_Number is range implementation-defined;
Unspecified_Network_Number : constant Network_Number;
type Database_Array is new POSIX.Octet_Array;
type Database_Array_Pointer is access all Database_Array;
function Get_Name (Info_Item : Network_Info)
  return POSIX.POSIX_String;
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Network_Alias (Info_Item : Network_Info);
function Get_Family (Info_Item : Network_Info)
  return POSIX_Sockets.Protocol_Family;
function Get_Network_Number (Info_Item : Network_Info)
  return Network_Number;
function Get_Network_Info_By_Address
   (Number  : Network_Number;
    Family  : POSIX_Sockets.Protocol_Family;
    Storage : Database_Array_Pointer)
  return Network_Info;
function Get_Network_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Network_Info;
procedure Open_Network_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Network_Database_Connection;
-- Network Protocol Database Functions
type Protocol_Info is private;
function Get_Name (Info_Item : Protocol_Info)
  return POSIX.POSIX_String;
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Protocol_Alias (Info_Item : Protocol_Info);
function Get_Protocol_Number (Info_Item : Protocol_Info)
  return POSIX_Sockets.Protocol_Number;
function Get_Protocol_Info_By_Number
   (Number  : POSIX_Sockets.Protocol_Number;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
function Get_Protocol_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
procedure Open_Protocol_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Protocol_Database_Connection;
```

D.1.3.2.2 Description

The functionality described in this subclause is optional. If the Network Management option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

The functions described in this subclause shall convert Internet addresses between `Internet_Address` objects and strings in Internet address dot notation. Internet address dot notation (also known as *dotted decimal notation*, see RFC 1020 {B22} and RFC 1983{B23}) is the common notation for Internet addresses where portions of the address are represented by decimal digits separated with periods (leading zeros are allowed).

The `String_To_Internet_Address` function shall convert the string `Address`, in Internet address dot notation, to an object of type `Internet_Address`.

The `Internet_Address_To_String` function shall convert the Internet address specified by `Address` to a string in Internet address dot notation.

The `Is_Internet_Address` function shall return `True` if the string `Address` is a valid Internet address in Internet address dot notation.

For the `String_To_Internet_Address` function, values specified in Internet address dot notation take the following forms:

a.b.c.d

When there are four parts, each is interpreted as an octet of data, and they are assigned to the four octets of the Internet address. The first part of the string is assigned to the most significant octet of the address.

a.b.c

When there are three parts, each of the first two is interpreted as an octet of data, and they are assigned to the two most significant octets of the Internet address. The third part is interpreted as a 16-bit quantity and is assigned to the two least significant octets of the address.

NOTE: This form is convenient for specifying a three-part Class B network address in the form nethi.netlo.host.

a.b

When there are two parts, the first part is interpreted as an octet of data and is assigned to the most significant octet of the network address. The second part is interpreted as a 24-bit quantity and assigned to the least significant three octets of the network address.

NOTE: This form is convenient for specifying a Class A network address in the form net.host.

a

When there is only one part, it is interpreted as a 32-bit quantity and is assigned to the whole network address without any rearrangement.

The `Get_Network_Info_By_Address` function and the `Get_Network_Info_By_-Name` function shall return a `Network_Info` object, the attributes of which shall contain information about a network. The `Network_Info` object shall include at least the following attributes:

Name

Name of the network. The function `Get_Name` shall return this attribute.

Alias Names

A list of alternative network names.

The application program instantiates the generic procedure `For_Every_Network_Alias` with an actual procedure for the generic formal procedure `Action`. When called, the instance shall call the actual procedure supplied for `Action` once for each element in the associated list.

`Action` shall be able to force termination of the generic instance either by setting `Quit` to `True` or by raising an exception. Prior to calling `Action`, the instance shall set `Quit` to `False`. Exceptions raised by `Action` shall terminate iteration and shall be propagated back to the caller of the instance. After an exception is raised by `Action` or `Action` returns with `Quit` set to `True`, no more calls to `Action` shall occur.

Family

The protocol family of the network. The function `Get_Family` shall return this attribute.

Network Number

The network number. The `Get_Network_Number` function shall return this attribute.

The `Get_Network_Info_By_Address` function shall search for information associated with the network specified by `Number` with the protocol family specified by `Family`, opening a connection to the database if necessary.

The `Get_Network_Info_By_Name` function shall search for information about the network name, opening a connection to the database if necessary.

The `Open_Network_Database_Connection` procedure shall open a connection to the database. If the `Stay_Open` parameter is `True`, the connection to the network database need not be closed after calls to the "`Get_Network_Info_`" functions and the implementation may maintain an open file descriptor for the database.

The `Close_Network_Database_Connection` procedure shall close the connection to the database, releasing any open file descriptor.

`Get_Network_Info_By_Address` and `Get_Network_Info_By_Name` shall return a `Network_Info` object. If the search was not successful, the `Network_Info` object contains empty strings for the Name and Alias Names attributes, `Unspecified_Protocol_Family` for the Family attribute, and `Unspecified_Network_Number` for the Network Number attribute.

Functions `Get_Protocol_Info_By_Name` and `Get_Protocol_Info_By_Number` shall return a `Protocol_Info` object, the attributes of which contain information about a network protocol. The `Protocol_Info` object shall include at least the following attributes.

Name

Official name of the protocol. The function `Get_Name` shall return this attribute.

Alias Names

A list of alternative protocol names.

The application program instantiates the generic procedure `For_Every_Protocol_Alias` with an actual procedure for the generic formal procedure `Action`. When called, the instance shall call the actual procedure supplied for `Action` once for each element in the associated list.

`Action` shall be able to force termination of the generic instance either by setting `Quit` to `True` or by raising an exception. Prior to calling `Action`, the instance shall set `Quit` to `False`. Exceptions raised by `Action` shall terminate iteration and shall be propagated back to the caller of the instance. After an exception is raised by `Action` or `Action` returns with `Quit` set to `True`, no more calls to `Action` shall occur.

Protocol Number

The protocol number. The function `Get_Protocol_Number` shall return this attribute.

The `Get_Protocol_Info_By_Name` function shall search for information associated with a protocol specified by `Name`, opening a connection to the database if necessary.

The `Get_Protocol_Info_By_Number` function shall search for information associated with a protocol specified by `Number`, opening a connection to the database if necessary.

The `Open_Protocol_Database_Connection` procedure shall open a connection to the database. If the `Stay_Open` parameter is true, the connection to the network protocol database need not be closed after calls to the "`Get_Protocol_Info_By_`" functions and the implementation may maintain an open file descriptor for the database.

The `Close_Protocol_Database_Connection` procedure shall close the connection to the database, releasing any open file descriptor.

`Get_Protocol_Info_By_Name` and `Get_Protocol_Info_By_Number` shall return a `Protocol_Info` object. If the search was not successful, the `Protocol_Info` object contains empty strings for the Name and Alias Names attributes and `Default_Protocol` for the Protocol Number attribute.

The functions that return the `Network_Info` and `Protocol_Info` objects include a `Storage` parameter, which points to a `Database_Array` object. This object provides static storage for the alias name lists that may be dynamically allocated by the underlying operating system services.

NOTE: Methods to estimate the size of the storage required for the alias name list are implementation defined. In practice, these lists are usually rather short. The application should not deallocate this extra storage before deallocating `Network_Info` and `Protocol_Info` objects.

NOTE: The storage lifetime of `Network_Info` and `Protocol_Info` objects is unspecified. Due to limitations in the operating system services underlying this binding, no guarantees about the behavior of these objects related to reentrancy or multithreaded safety are made.

D.1.3.3 Error Handling

`Constraint_Error` may be raised by the functions that return the `Network_Info` and `Protocol_Info` objects for implementation-defined reasons, including the situation where the `Database_Array` object supplied via the `Storage` parameter is too small to store the entire alias name list.

D.1.3.4 Internet Transmission Control Protocol

D.1.3.4.1 Synopsis

```
subtype Keep_Alive_Time is POSIX.Seconds range 1 .. POSIX.Seconds'Last;
function Get_Keep_Alive_Interval
   (Socket : POSIX_IO.File_Descriptor)
  return Keep_Alive_Time;
procedure Set_Keep_Alive_Interval
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Keep_Alive_Time);
function Get_No_Delay
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_No_Delay
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
subtype Socket_Retransmit_Time is POSIX.Seconds range
   implementation-defined;
Wait_Forever : constant Socket_Retransmit_Time := impl-def-static-expression;
Retransmit_Time_Default
            : constant Socket_Retransmit_Time := impl-def-static-expression;
function Get_Retransmit_Time_Maximum
   (Socket : POSIX_IO.File_Descriptor)
  return Socket_Retransmit_Time;
procedure Set_Retransmit_Time_Maximum
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Socket_Retransmit_Time);
function Get_Segment_Size_Maximum
   (Socket : POSIX_IO.File_Descriptor)
  return Positive;
function Get_Standardized_Urgent_Data
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_Standardized_Urgent_Data
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
```

D.1.3.4.2 Description

The functionality described in this subclause is optional. If the Internet Stream option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this subclause.

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is an octet-stream protocol used to support the `Stream_Socket` abstraction.

The TCP protocol service supports all of the states in Figure D.1. The receipt of a TCP connection request (a segment with the SYN bit that matches a listening TCP

socket) shall cause an acknowledgment of the SYN and completion of the protocol connection before the Accept_Connection procedure may return the connection. If the three-way handshake does not complete and the connection does not reach the Connected state, the connection shall not be returned as a new socket via the Accept_Connection call; the behavior of the interface shall be as if the connection request had never been received.

TCP uses the standard Internet endpoint address formats and conventions for connecting to a remote endpoint described in D.1.3.1.

If the application makes a call to Shutdown to cease output on a socket that has established a connection, the protocol implementation shall send a FIN indication after sending all data queued for output, indicating the shutdown event to the remote endpoint. If the application makes a call to Shutdown to cease input, the TCP protocol implementation shall take no action.

NOTE: When an application does a shutdown for input, no action is taken by the TCP protocol implementation, as no protocol mechanism exists for this event. However, if new data are received for a connection in the Sending Only state or the Dead state, TCP should abort the connection with an RST indication, as the data cannot be delivered successfully.

When a TCP socket is closed in the Listening state, pending connections to that socket that have not been returned by the Accept_Connection procedure shall be aborted by sending a segment containing the RST flag.

An immediate disconnect that is initiated by closing a TCP socket with the Socket Linger Time socket option enabled, a linger time of zero, and an existing connection to a peer (in states Connected, Sending Only, or Receiving Only) shall cause the connection to be aborted by sending a segment containing the RST flag.

A normal disconnect initiated by closing a connected TCP socket with the Socket Linger Time socket option disabled or with a linger time other than zero shall not cause data to be discarded. When a normal disconnect is initiated, the implementation shall attempt to deliver any buffered data followed by a FIN indication whether the socket blocks awaiting the completion of the disconnect.

Once the FIN has been acknowledged, the disconnect shall be considered complete. If the Socket Linger Time socket option is disabled, the close of a socket shall return immediately, and the protocol implementation shall continue to attempt delivery of any buffered data and the FIN indication until they are acknowledged or the protocol implementation times out. If the Socket Linger Time socket option is enabled with a time other than zero, the close operation shall block until buffered data and the FIN are acknowledged or until the linger time has expired. However, the protocol shall continue to attempt delivery of any buffered data and FIN after the close operation returns until any data and FIN are acknowledged or the implementation times out. The implementation may retain protocol state from the socket to await expiration of protocol timers after the disconnect.

The TCP protocol supports a notion of a pointer to urgent data; data preceding and at the urgent pointer are considered to be urgent. A TCP connection for which an urgent pointer has been received is considered to be in urgent mode until the data at the urgent pointer have been received. However, there is no notion in TCP of out-of-band data. Process_OOB_Data is used instead to send data in urgent mode, and

(when not using the Socket OOB Data Inline socket option) to receive a specific octet of urgent data.

By default, when an application uses the Send or Send_Message functions and specifies Process_OOB_Data, then the TCP urgent pointer is set to the sequence number of the octet following the data presented to the function. As TCP currently specifies the urgent pointer as the last octet of urgent data rather than the first octet following the urgent data, one additional octet of data after the data sent with Process_OOB_Data specified is also part of the urgent data. Thus, an application that wishes to send two octets of data in urgent mode may send the first octet with Process_OOB_Data specified and then send the second octet of data in a separate call without Process_OOB_Data specified. The interpretation of Process_OOB_Data on send operations for TCP can be modified by setting the Standardized Urgent Data socket option. Using that option, it is possible to send a single octet of data in urgent mode.

When the TCP implementation receives a new urgent pointer and the application has set a process or process group to receive signals from the socket, the POSIX_Signals.Signal_Out_Of_Band_Data signal shall be generated for that process or process group when the protocol is first notified of the new urgent pointer. Not all of the urgent data have necessarily been received by the protocol at this time, as urgent data are subject to normal flow control.

The subsequent handling of incoming data in urgent mode depends upon the value of the Socket OOB Data Inline socket option for the socket. If the Socket OOB Data Inline socket option is not set for the socket, incoming data shall be processed normally until the octet with the sequence number one lower than the urgent pointer is received. The implementation shall then remove that octet from the normal data stream and shall place an out-of-band data mark in the socket receive queue in the place of that octet. The implementation shall then make the octet removed from the data stream available via a call to the Receive or Receive_Message function specifying Process_OOB_Data. If the octet before the urgent pointer has not yet arrived and if POSIX_IO.Non_Blocking is not set for the socket descriptor, it is implementation defined whether a receive call with Process_OOB_Data specified shall fail with an error of Would_Block or shall block. However, the call shall not block if the octet with the sequence number one lower than the urgent pointer cannot be received due to the flow control window.

If the Socket OOB Data Inline socket option is set for the socket, incoming data shall be processed normally until the octet with the sequence number one lower than the urgent pointer is received. The implementation shall then place an out-of-band data mark in the socket receive queue immediately before that octet; it shall then continue to place data into the receive queue normally. All data shall thus be placed into the receive queue in the normal order.

If an application calls the Receive or Receive_Message procedure specifying Process_OOB_Data when no out-of-band data are pending (either before any urgent pointer is detected or after reading the most recent out-of-band data mark without the Peek_Only) option, an error shall occur.

The result of a receive with Process_OOB_Data specified on a socket that has the Socket OOB Data Inline socket option set and is in the urgent data condition is undefined.

Applications receiving data sent in urgent mode using TCP should use the Socket OOB Data Inline socket option.

Because the amount of urgent data can exceed the available buffering, an application that does not use the Socket OOB Data Inline socket option shall read all pending urgent data in sequence, using receive calls that do not specify `Process_OOB_Data`. Reading the data immediately enables the implementation to free buffer space for additional data to be sent by the peer. After each receive operation in urgent mode, the application shall then test whether the previously returned data reached the urgent mark by calling the `Socket_Is_At_OOB_Mark` function. Once the mark has been reached, the application may call `Receive` or `Receive_Message` with `Process_OOB_Data` specified to receive the out-of-band octet.

Between the time that an urgent pointer is first detected and the time that the out-of-band data mark is removed from the socket receive queue, a call to the `Select_File` procedure testing this socket descriptor for exceptional conditions shall return an indication that an exceptional condition exists. At all other times, an exceptional indication shall not be indicated on a socket using TCP.

For TCP sockets, the implementation shall support the Socket Debugging socket option, the Socket Reuse Addresses socket option, and the Socket OOB Data Inline socket option. It is implementation defined whether the implementation supports the Socket Routing socket option on sockets using TCP.

It is implementation defined whether the Socket Keep Alive socket option has effect on sockets using TCP. It is unspecified whether the value of the Socket Broadcast socket option has any effect on sockets using TCP. A socket using TCP shall support all options defined for the Internet protocol except the Receive Destination Address socket option and the Header Included socket option.

The following list identifies the TCP-specific options, the types of the option value parameters associated with each option, the functions and procedures used to manipulate the option, the default values for the options, and a synopsis of the meaning of the option value parameter:

Keep Alive Interval

> `Set_Keep_Alive_Interval` specifies the amount of time in seconds between keep-alive probes. This option is effective only when the Socket Keep Alive socket option has been enabled on the socket and the implementation supports the use of the Socket Keep Alive socket option with TCP. The default value of the Keep Alive Interval socket option shall be implementation defined but shall not be less than 7200 (since the unit is seconds, 7200 means two hours). `Get_Keep_Alive_Interval` returns the current value of this option.

No Delay

> Under most circumstances, TCP sends data immediately (subject to flow control) when the application calls an output function. However, when outstanding data have not yet been acknowledged, the implementation may gather small amounts of output to be sent in a single, larger packet once an acknowledgment is received or a full packet can be constructed. For some

clients, such as window systems that send a stream of mouse events that receive no replies, this packetization may cause significant delays. Therefore, the implementation shall inhibit this packetization by using Set_No_Delay to set the No Delay socket option to Enabled.

Applications should minimize the use of this option due to possible performance degradation. The default value for the No Delay socket option is Disabled.

Retransmit Time Maximum

Set_Retransmit_Time_Maximum specifies the amount of time in seconds before the connection is broken once TCP determines it is retransmitting data. The implementation may round the specified value up to the next time at which it would retransmit under the current conditions (including the estimated round trip time and back-off values). Get_Retransmit_Time_Maximum returns the current value of this option. A value of Retransmit_Time_Default is the default and indicates the use of the system default behavior, which may depend upon the estimated round-trip time for the connection. A value of Wait_Forever means to wait forever.

Segment Size Maximum

Get_Segment_Size_Maximum returns the current segment size, in octets, of the TCP connection. It is implementation defined whether the value of this option can be set by an application, and if so, within what range of values. The default value of this option is unspecified.

Standardized Urgent Data

Set_Standardized_Urgent_Data influences the behavior of the TCP implementation when sending out-of-band (urgent) data. By default, a send specifying Process_OOB_Data causes the urgent pointer to be set to the sequence number of the next octet after the data presented. The current TCP definition of the URG field does not allow transmission of a single octet in urgent mode. However, when the Standardized Urgent Data socket option has a value of Enabled, a send specifying Process_OOB_Data shall cause the urgent pointer to be set to the sequence number of the last data octet presented. The default value of the Standardized Urgent Data socket option shall be Disabled.

If a send operation on a TCP socket specifies the Do_Not_Route flag, the results are undefined. If the Socket Routing socket option is set to Disabled for the socket and the implementation supports this option for TCP, the normal routing mechanism shall be ignored for outgoing TCP packets. If the destination address refers to a destination on a directly attached network interface, that interface shall be used to deliver the datagram to the destination; otherwise, an error shall occur.

D.1.3.4.3 Error Handling

If any of the following conditions occurs, the exception POSIX_Error shall be raised with the corresponding error code:

Network_Unreachable

A send operation on a TCP socket specified the Do_Not_Route flag with undefined results.

`Would_Block`

A `Receive` procedure specified the `Process_OOB_Data` before the urgent pointer has arrived, and `POSIX_IO.Non_Blocking` is not set for the socket descriptor.

`Invalid_Argument`

An application called `Receive` or `Receive_Message` specifying `Process_OOB_Data` when no out-of-band data are pending.

D.1.3.5 User Datagram Protocol

D.1.3.5.1 Description

UDP is a simple unreliable datagram protocol that is used to support the `Datagram_Socket` abstraction for the Internet protocol family. UDP sockets are connectionless mode, and are normally used by `Send` with the `To` parameter and `Receive` with the `From` parameter.

The UDP service supports all states in Figure D.3.

UDP uses the standard Internet endpoint address formats and conventions described in D.1.3.1 and further specified in this subclause.

In the Ground state, the local and remote addresses and ports for the endpoint shall all be unbound (port values of zero, addresses of `Unspecified_Internet_Address`).

In the Ground state, a successful call to the `Bind` procedure shall bind a local port number. If the address specified with the `Bind` call contains a port number of zero, an unused local port number shall be bound to the endpoint. If the host address specified with `Bind` is `Unspecified_Internet_Address`, the local address for the endpoint shall remain unspecified.

If the `Send` with the `To` parameter or `Send_Message` functions are successfully called for a UDP socket in the Ground state, an unused local port number shall be bound to the endpoint.

The local address for the endpoint shall remain unbound. Once a local port number has been bound to the endpoint, it shall not be unbound until a Close event.

UDP uses the standard Internet conventions when connecting to a remote endpoint as described in D.1.3.1 and further specified here.

When a `Send` with the `To` parameter or a `Send_Message` call is made in the Bound state on a UDP socket for which no local host address has been bound, the protocol shall select a source address for the outgoing datagram, and the local address for the endpoint shall remain unbound. If the local address has been bound before a `Send` with the `To` parameter or `Send_Message` call in the Bound state, that address shall be used as the source address of the datagram.

When the `Connect` procedure is successfully called for a UDP socket, the remote address and port number shall be set for the endpoint. If the local port has not been bound, an unused local port number shall be bound to the endpoint. If the local

address has not been bound, a local address shall be selected and bound. In the Open state, output operations shall not specify a destination address, and all datagrams shall be sent to the prespecified peer. Only datagrams from the prespecified peer shall be received.

If `Connect` is called for a UDP socket in the Open state, the behavior shall be as if the socket is placed in the Bound state, the remote address and port and the local address are set to the Unbound state, leaving the local port number unchanged. The `Connect` call is then made in this state. If the `Connect` is unsuccessful because the address family is invalid or because the destination port and/or address is unspecified, the socket shall remain in the Bound state with no peer address or port and with the local address set to `Unspecified_Internet_Address`. An error shall be returned as specified for this function. The local port number shall be unchanged.

Closing a UDP socket shall have no effect on data previously sent.

UDP does not support out-of-band data; send or receive operations that specify `Process_OOB_Data` shall generate the error `Invalid_Argument`.

The implementation shall support the Socket Broadcast socket option and the Socket Reuse Addresses socket option on sockets using UDP. It is implementation defined whether the implementation supports the Socket Routing socket option on sockets using UDP. It is unspecified whether the Socket Keep Alive socket option, the Socket Debugging socket option, and the Socket OOB Data Inline socket option have any effect on sockets using UDP. A socket using UDP shall support all options defined for the Internet protocol (see D.1.3.6) except the IP Header Included socket option. No options are specified for the UDP protocol.

If a send operation specifies the `Do_Not_Route` flag or if the Socket Routing socket option is set to `Disabled` for the socket during a send operation, and if the implementation supports disabling routing on sockets using UDP, the normal routing mechanism shall be ignored for the output datagram. If the destination address refers to a destination on a directly attached network interface, that interface shall be used to deliver the datagram to the destination; otherwise, the operation shall generate the error `Network_Unreachable`.

If a send operation attempts to send a datagram to a broadcast (see RFC 791 {13}) address for a local network using a socket for which the Socket Broadcast socket option has not been set, the function may generate the error `Permission_Denied`. If a send operation attempts to send a datagram to a broadcast address for a local network, and the datagram cannot be sent on that network without fragmentation, the function may generate the error `Message_Too_Long`.

D.1.3.5.2 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Invalid_Argument`

A send or receive specified `Process_OOB_Data`.

`Message_Too_Long`

A send operation attempted to send a datagram to a broadcast address for

a local network, and the datagram cannot be sent on that network without fragmentation.

Network_Unreachable

A send operation specified Do_Not_Route or the Socket Routing socket option is set to Disabled for the socket, and the destination address does not refer to a destination on a directly attached network interface.

Permission_Denied

A send operation attempted to send a datagram to a broadcast address for a local network using a socket for which the Socket Broadcast socket option has not been set.

D.1.3.6 Internet Protocol

D.1.3.6.1 Synopsis

```
function IP_Header_Options_In_Use
   (Socket : POSIX_IO.File_Descriptor)
  return Boolean;
procedure Reset_IP_Header_Options
   (Socket : in POSIX_IO.File_Descriptor);
type IP_Options_Buffer is private;
function Get_IP_Header_Options
   (Socket : POSIX_IO.File_Descriptor)
  return IP_Options_Buffer;
procedure Set_IP_Header_Options
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in IP_Options_Buffer);
function Get_First_Hop
   (Options : IP_Options_Buffer)
  return Internet_Address;
procedure Set_First_Hop
   (Options : in out IP_Options_Buffer;
    Address : in     Internet_Address);
function Get_IP_Options
   (Options : IP_Options_Buffer)
  return POSIX.Octet_Array;
procedure Set_IP_Options
   (Options : in out IP_Options_Buffer;
    Buffer  : in     POSIX.Octet_Array);
type IP_Type_Of_Service is private;
Low_Delay        : constant IP_Type_Of_Service;
High_Throughput  : constant IP_Type_Of_Service;
High_Reliability : constant IP_Type_Of_Service;
Unspecified      : constant IP_Type_Of_Service;
type Time_To_Live is range 0 .. 255;
function Get_Type_Of_Service
   (Socket : POSIX_IO.File_Descriptor)
  return IP_Type_Of_Service;
procedure Set_Type_Of_Service
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in IP_Type_Of_Service);
function Get_Initial_Time_To_Live
   (Socket : POSIX_IO.File_Descriptor)
  return Time_To_Live;
procedure Set_Initial_Time_To_Live
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in Time_To_Live);
```

```
function Get_Receive_Destination_Address
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_Receive_Destination_Address
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
type IP_Ancillary_Data is private;
type IP_Ancillary_Data_Pointer is access all IP_Ancillary_Data;
procedure Set_Ancillary_Data
   (Message : in out POSIX_Sockets.Socket_Message;
    Data    : in     IP_Ancillary_Data_Pointer);
function Get_Destination_Address
   (Data : IP_Ancillary_Data)
  return Internet_Address;
function Get_Header_Included
   (Socket : POSIX_IO.File_Descriptor)
  return POSIX_Sockets.Socket_Option_Value;
procedure Set_Header_Included
   (Socket : in POSIX_IO.File_Descriptor;
    To     : in POSIX_Sockets.Socket_Option_Value);
```

D.1.3.6.2 Description

The Internet Procol is the network layer protocol used by the Internet protocol family. It may also be accessed through a raw socket when developing new protocols or special purpose applications.

Raw IP sockets are connectionless-mode and are normally used by Send with the To parameter and Receive with the From parameter, although Connect may also be used to fix the destination for future packets. Raw IP sockets support all the states in Figure 18.1.

If the Protocol parameter on the call to Create is omitted or specified as Default_Protocol, the default protocol Raw is used for outgoing packets, and only incoming packets destined for that protocol are received. If the Protocol parameter is used, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header appended in front of them (based on the destination address and the protocol number with which the socket is created), unless the IP Header Included socket option has been set. Incoming packets are received with IP header and options intact.

If a send operation specifies the Do_Not_Route flag or the Socket Routing socket option is set to Disabled for the socket during a send operation, and the implementation supports disabling socket routing, the normal routing mechanism shall be ignored for the output datagram. If the destination address refers to a destination on a directly attached network interface, that interface shall be used to deliver the datagram to the destination; otherwise, the operation shall generate the error Network_Unreachable.

Options may be set for the Internet protocol when using higher level protocols that are based on IP (such as TCP and UDP).

The following list identifies the IP-specific options, the types of the option value parameters associated with each option, the functions and procedures used to manipulate the option, the default values for the options, and a synopsis of the meaning of the option value parameter:

IP Header Options

This option may be used to provide Internet protocol options to be transmitted in the IP header of each outgoing packet or to examine the header options of incoming packets. The option value is set with the `Set_IP_Header_Options` procedure and retrieved with the `Get_IP_Header_Options` function. The option value is an `IP_Options_Buffer` object with the following attributes:

First Hop

The first-hop gateway Internet address, which is used when a list of addresses for the Source Route socket option is specified. This attribute must be set when using the Source Route socket option. `Set_First_Hop` may be used to set this attribute. The `Get_First_Hop` function shall return the first-hop IP address if a Source Route socket option is present. Otherwise it shall return the value `Unspecified_Internet_Address`.

IP Options

An array of octets to hold the IP options. The maximum number of octets, as determined by the header format, is 40. `Set_IP_Options` and `Get_IP_Options` may be used to set and examine this attribute.

The format of the IP Options attribute is that specified by the Internet protocol (see RFC 791 {13}), as it should be appended to the standard IP header, with one exception: the list of addresses for the Source Route socket option must have the First Hop attribute set to the first-hop gateway, and the beginning of the IP Options attribute set to the list of gateways followed by additional IP options. At most one Source Route socket option shall be included in the protocol options. If a Source Route socket option is included, the implementation shall extract and remove the first-hop gateway address from the option list and then adjust the size accordingly before inserting the option list into datagrams.

NOTE: For further details on IP options and source routes, see RFC 791 {13} and RFC 1122 {18}.

The procedure `Reset_IP_Header_Options` disables any previous options. Function `IP_Header_Options_In_Use` returns `False` if no options are currently in use.

Type Of Service

`Set_Type_Of_Service` is used to set the type-of-service field in the IP header for outgoing packets. The default value for the Type Of Service socket option is `Unspecified`. `Get_Type_Of_Service` returns the current value of this option.

Initial Time To Live

`Set_Initial_Time_To_Live` is used to set the time-to-live field in the IP header for outgoing packets. The default value for the Initial Time To Live

socket option is implementation defined. `Get_Initial_Time_To_Live` returns the current value of this option.

Receive Destination Address

`Set_Receive_Destination_Address` is used to enable a `Receive_Message` call on a `Datagram_Socket` socket to return the destination IP address for incoming datagrams as ancillary data. The `Set_Ancillary_Data` procedure prepares a socket message to receive ancillary data by storing a pointer to an object of type `IP_Ancillary_Data`. The `Get_Destination_Address` function retrieves the ancillary data as an object of type `Internet_Address`. The default value of the Receive Destination Address socket option shall be `Disabled`. `Get_Receive_Destination_Address` returns the current value of this option.

IP Header Included

`Set_Header_Included` is used to enable the complete IP header to be included with the data on send operations. This option shall be used only with the `Raw_Socket` type. It is implementation defined whether the IP Header Included socket option is supported. If it is not supported, an attempt to set the option shall result in an error. `Get_Header_Included` returns the current value of this option.

The program must set all the fields of the IP header, including the IP version number, the header length, the packet identification field, and the data offset. If the packet identification attribute is set to zero, the implementation shall fill in an appropriate value. If the source address is set to `Unspecified_Internet_Address`, the implementation shall choose an appropriate address. The default value of the IP Header Included socket option shall be `Disabled`.

D.1.3.6.3 Error Handling

If any of the following conditions occurs, the exception `POSIX_Error` shall be raised with the corresponding error code:

`Network_Unreachable`

A send operation specified `Do_Not_Route` or the Socket Routing socket option is set to `Disabled` for the socket, and the destination address does not refer to a destination on a directly attached network interface.

`Option_Not_Supported`

`Set_Header_Included` was called and the IP Header Included socket option is not supported by the implementation.

D.2 XTI Protocol Mappings

This clause describes Ada mappings to facilities that support specific XTI network protocols.

D.2.1 Package POSIX_XTI_mOSI

This package provides the DNI/XTI interface mappings for the minimal 7-layer OSI stack. Unless otherwise specified, all the DNI/XTI calls in package POSIX_XTI can be used for mOSI. Only additional information relevant to this protocol is highlighted in this subclause.

The functionality described in this subclause is optional. If either the XTI Detailed Network Interface option or the OSI Minimal option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior shall be as specified in this subclause.

```
with POSIX,
     POSIX_XTI;
package POSIX_XTI_mOSI is
    -- D.2.1.4 mOSI Naming and Addressing
   type mOSI_XTI_Address is private;
   type mOSI_XTI_Address_Pointer is access all mOSI_XTI_Address;
   function "+" (Pointer : mOSI_XTI_Address_Pointer)
     return POSIX_XTI.XTI_Address_Pointer;
   function "+" (Pointer : POSIX_XTI.XTI_Address_Pointer)
     return mOSI_XTI_Address_Pointer;
   function Is_mOSI_XTI_Address
      (Pointer : POSIX_XTI.XTI_Address_Pointer)
     return Boolean;
   mOSI_Address_Length_Maximum : constant Natural := implementation-defined;
   type AP_Invocation_Id is new POSIX.Octet_Array;
   type AE_Invocation_Id is new POSIX.Octet_Array;
   type AP_Title is new POSIX.Octet_Array;
   type AE_Qualifier is new POSIX.Octet_Array;
   type Presentation_Address is new POSIX.Octet_Array;
   type mOSI_Address_Flags is new POSIX.Option_Set;
   AP_Invocation_Id_Valid : constant mOSI_Address_Flags := implementation-defined;
   AE_Invocation_Id_Valid : constant mOSI_Address_Flags := implementation-defined;
   function Get_Flags
      (Address : mOSI_XTI_Address)
     return mOSI_Address_Flags;
   procedure Set_Flags
      (Address : in out mOSI_XTI_Address;
       To      : in     mOSI_Address_Flags);
   function Get_AP_Invocation_Id
      (Address : mOSI_XTI_Address)
     return AP_Invocation_Id;
   procedure Set_AP_Invocation_Id
      (Address : in out mOSI_XTI_Address;
       To      : in     AP_Invocation_Id);
   function Get_AE_Invocation_Id
      (Address : mOSI_XTI_Address)
     return AE_Invocation_Id;
   procedure Set_AE_Invocation_Id
      (Address : in out mOSI_XTI_Address;
       To      : in     AE_Invocation_Id);
   function Get_AP_Title
      (Address : mOSI_XTI_Address)
     return AP_Title;
```

```
function Get_AE_Qualifier
   (Address : mOSI_XTI_Address)
  return AE_Qualifier;
function Get_Presentation_Address
   (Address : mOSI_XTI_Address)
  return Presentation_Address;
procedure Set_OSI_Address
   (Address : in out mOSI_XTI_Address;
    AP      : in     AP_Title;
    AE      : in     AE_Qualifier;
    PA      : in     Presentation_Address);
procedure Get_Address
   (Info_Item : in     POSIX_XTI.Connection_Info;
    Address   : in out mOSI_XTI_Address);
-- D.2.1.5 mOSI Options
mOSI_Connection_Mode     :
   constant POSIX_XTI.Option_Level := implementation-defined;
mOSI_Connectionless_Mode :
   constant POSIX_XTI.Option_Level := implementation-defined;
Application_Context      :
   constant POSIX_XTI.Option_Name := implementation-defined;
Presentation_Context     :
   constant POSIX_XTI.Option_Name := implementation-defined;
type Object_Identifier is new POSIX.Octet_Array;
type Application_Context_Name is new Object_Identifier;
type Presentation_Context_List is private;
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Application_Context_Name;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    Value       : in     Application_Context_Name);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Presentation_Context_List;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    Value       : in     Presentation_Context_List);
-- Presentation Context Definition and Result List
type Presentation_Context_Item is private;
type Presentation_Item_Id is implementation-defined-integer;
procedure Set_Presentation_Id
   (Item : in out Presentation_Context_Item;
    To   : in     Presentation_Item_Id);
function Get_Presentation_Id
   (Item : Presentation_Context_Item)
  return Presentation_Item_Id;
type Negotiation_Result is private;
Presentation_Context_Accepted : constant Negotiation_Result;
Presentation_Context_Rejected : constant Negotiation_Result;
Rejected_No_Reason_Specified  : constant Negotiation_Result;
Abstract_Syntax_Not_Supported : constant Negotiation_Result;
Transfer_Syntax_Not_Supported : constant Negotiation_Result;
Local_DCS_Limit_Exceeded      : constant Negotiation_Result;
```

```
procedure Set_Negotiation_Result
   (Item : in out Presentation_Context_Item;
    To   : in     Negotiation_Result);
function Get_Negotiation_Result
   (Item : Presentation_Context_Item)
  return Negotiation_Result;
type Syntax_Object_List is private;
procedure Set_Syntax_Object
   (Item : in out Presentation_Context_Item;
    To   : in     Syntax_Object_List);
function Get_Syntax_Object
   (Item : Presentation_Context_Item)
  return Syntax_Object_List;
Empty_Presentation_Context_List : constant Presentation_Context_List;
procedure Make_Empty (List : in out Presentation_Context_List);
procedure Add
   (List : in out Presentation_Context_List;
    Item : in     Presentation_Context_Item);
generic
    with procedure Action
       (Item : in out Presentation_Context_Item;
        Quit : in out Boolean);
procedure For_Every_Presentation_Context_Item
   (List : in Presentation_Context_List);
function Length (List : Presentation_Context_List)
  return Natural;
function Element
   (List  : Presentation_Context_List;
    Index : Positive)
  return Presentation_Context_Item;
Empty_Syntax_Object_List : constant Syntax_Object_List;
procedure Make_Empty (List : in out Syntax_Object_List);
procedure Add
   (List : in out Syntax_Object_List;
    Item : in    Object_Identifier);
generic
    with procedure Action
       (Object : in     Object_Identifier;
        Quit   : in out Boolean);
procedure For_Every_Object_Identifier
   (List : in Syntax_Object_List);
function Length (List : Syntax_Object_List)
  return Natural;
function Element
   (List  : Syntax_Object_List;
    Index : Positive)
  return Object_Identifier;
-- D.2.1.11 XTI Functions
Rejected_By_Peer          :
   constant POSIX_XTI.Reason_Code := implementation-defined;
AC_Name_Not_Supported     :
   constant POSIX_XTI.Reason_Code := implementation-defined;
Unrecognized_AP_Title     :
   constant POSIX_XTI.Reason_Code := implementation-defined;
Unrecognized_AE_Qualifier :
   constant POSIX_XTI.Reason_Code := implementation-defined;
Authentication_Required   :
   constant POSIX_XTI.Reason_Code := implementation-defined;
```

```
   Aborted_By_Peer           :
      constant POSIX_XTI.Reason_Code := implementation-defined;
   No_Common_Version         :
      constant POSIX_XTI.Reason_Code := implementation-defined;
private
   implementation-defined
end POSIX_XTI_mOSI;
```

D.2.1.1 Application Contexts

An *application context name* identifies a set of tasks to be performed by an application. It is exchanged during association establishment with the purpose of conveying a common understanding of the work to be done. This parameter is exposed to offer some negotiation capabilities to the application and to increase the chances of interoperability. When receiving an unsuitable or unknown value from a peer application, the application may propose an alternate value or decide to terminate the association prematurely. A default value (in the form of an object identifier) is provided, identifying a generic XTI-mOSI application. Its value can be found in D.2.1.10.

D.2.1.2 Presentation Contexts

A *presentation context* is the association of an abstract syntax with a transfer syntax. The presentation context is used by the application to identify how the data are structured and by the OSI application layer to identify how the data should be encoded/decoded.

A *generic presentation context* is defined for a stream oriented, unstructured, data transfer service with *null encoding*:

— *abstract syntax*: The single data type of this abstract syntax is a sequence of octets that are defined in the application protocol specification as being consecutive octets on a stream oriented communications mechanism without regard for any semantic or other boundaries.

— *transfer syntax*: The data value shall be represented as an octet-aligned presentation data value. If two or more data values are concatenated together they are considered to be a single (longer) data value. (The rule expressed in the preceding sentence is called the *null encoding rule*.)

The object identifiers for this generic presentation context can be found in D.2.1.10.

D.2.1.3 Presentation Context Definition and Result Lists

As negotiation occurs between the peer OSI application layers, the presentation context or contexts proposed by the application need not be accepted.

The Presentation Context Definition and Result List for each of the proposed presentation contexts indicate whether it is accepted and, if it is not, provides a reason code. The application may choose to terminate the association prematurely if it does not suit its requirements.

D.2.1.4 mOSI Naming and Addressing

D.2.1.4.1 Synopsis

```
type mOSI_XTI_Address is private;
type mOSI_XTI_Address_Pointer is access all mOSI_XTI_Address;
function "+" (Pointer : mOSI_XTI_Address_Pointer)
  return POSIX_XTI.XTI_Address_Pointer;
function "+" (Pointer : POSIX_XTI.XTI_Address_Pointer)
  return mOSI_XTI_Address_Pointer;
function Is_mOSI_XTI_Address
   (Pointer : POSIX_XTI.XTI_Address_Pointer)
  return Boolean;
mOSI_Address_Length_Maximum : constant Natural := implementation-defined;
type AP_Invocation_Id is new POSIX.Octet_Array;
type AE_Invocation_Id is new POSIX.Octet_Array;
type AP_Title is new POSIX.Octet_Array;
type AE_Qualifier is new POSIX.Octet_Array;
type Presentation_Address is new POSIX.Octet_Array;
type mOSI_Address_Flags is new POSIX.Option_Set;
AP_Invocation_Id_Valid : constant mOSI_Address_Flags := implementation-defined;
AE_Invocation_Id_Valid : constant mOSI_Address_Flags := implementation-defined;
function Get_Flags
   (Address : mOSI_XTI_Address)
  return mOSI_Address_Flags;
procedure Set_Flags
   (Address : in out mOSI_XTI_Address;
    To      : in     mOSI_Address_Flags);
function Get_AP_Invocation_Id
   (Address : mOSI_XTI_Address)
  return AP_Invocation_Id;
procedure Set_AP_Invocation_Id
   (Address : in out mOSI_XTI_Address;
    To      : in     AP_Invocation_Id);
function Get_AE_Invocation_Id
   (Address : mOSI_XTI_Address)
  return AE_Invocation_Id;
procedure Set_AE_Invocation_Id
   (Address : in out mOSI_XTI_Address;
    To      : in     AE_Invocation_Id);
function Get_AP_Title
   (Address : mOSI_XTI_Address)
  return AP_Title;
function Get_AE_Qualifier
   (Address : mOSI_XTI_Address)
  return AE_Qualifier;
function Get_Presentation_Address
   (Address : mOSI_XTI_Address)
  return Presentation_Address;
procedure Set_OSI_Address
   (Address : in out mOSI_XTI_Address;
    AP      : in     AP_Title;
    AE      : in     AE_Qualifier;
    PA      : in     Presentation_Address);
procedure Get_Address
   (Info_Item : in     POSIX_XTI.Connection_Info;
    Address   : in out mOSI_XTI_Address);
```

D.2.1.4.2 Description

The `mOSI_XTI_Address` object (used in `Bind`, `Connect`, and `Accept_Connection`) is a combined naming and addressing object, identifying one end or the other of the association. The address part is a presentation address. The calling and called addresses are required parameters while the use of a responding address is optional (see D.2.1.12).

The type `mOSI_XTI_Address` shall be used to represent an address for this protocol family. The type `mOSI_XTI_Address_Pointer` is an access to this protocol-specific address type. The `"+"` operations shall convert a `mOSI_XTI_Address_Pointer` to and from the `XTI_Address_Pointer` type for use with the base package operations defined for the `XTI_Address_Pointer` type. The return value of the `"+"` operations designates the same address object as the input parameter. The function `Is_mOSI_XTI_Address` shall return `True` if the address object designated by the specified non-null `XTI_Address_Pointer` is a valid `mOSI_XTI_Address` and `False` otherwise. The conversion operation to `mOSI_XTI_Address_Pointer` shall succeed if and only if the corresponding `Is_mOSI_XTI_Address` returns `True`. Otherwise, the results are undefined.

NOTE: The `Null_XTI_Address` constant corresponds to the Ada `null` literal for these operations.

The name parts AP Title and AE Qualifier are always optional.

NOTE: ISO directory facilities, when available, can relate the name parts (identifying specific applications) to the addresses of the real locations where they can be accessed.

The `mOSI_XTI_Address` object has at least the following attributes:

Flags

An `mOSI_Address_Flags` object indicating the presence of the corresponding invocation identifier in the PDU. The following flags are defined:

`AP_Invocation_Id_Valid`

The contents of the AP Invocation Identifier attribute is valid.

`AE_Invocation_Id_Valid`

The contents of the AE Invocation Identifier attribute is valid.

The operations `"+"`, `"-"`, `">"`, `"<"`, `">="`, `"<="`, and `Empty_Set` are available on the type `mOSI_Address_Flags` via the derived type semantics of Ada, from the operations available for `POSIX.Option_Set`. The appropriate operations can be used to create and examine a set containing the required flags. Function `Get_Flags` returns the current value for this attribute and procedure `Set_Flags` shall set this attribute.

AP Invocation Identifier

Function `Get_AP_Invocation_Id` returns the current value for this attribute and procedure `Set_AP_Invocation_Id` shall set this attribute.

AE Invocation Identifier

Function `Get_AE_Invocation_Id` returns the current value for this attribute, and procedure `Set_AE_Invocation_Id` shall set this attribute.

AP Title

Function `Get_AP_Title` returns the current value for this attribute, and procedure `Set_AP_Title` shall set this attribute.

AE Qualifier

Function `Get_AE_Qualifier` returns the current value for this attribute, and procedure `Set_AE_Qualifier` shall set this attribute.

Presentation Address

Function `Get_Presentation_Address` returns the current value for this attribute, and procedure `Set_Presentation_Address` shall set this attribute.

D.2.1.5 mOSI Options

D.2.1.5.1 Synopsis

```
mOSI_Connection_Mode     :
   constant POSIX_XTI.Option_Level := implementation-defined;
mOSI_Connectionless_Mode :
   constant POSIX_XTI.Option_Level := implementation-defined;
Application_Context      :
   constant POSIX_XTI.Option_Name := implementation-defined;
Presentation_Context     :
   constant POSIX_XTI.Option_Name := implementation-defined;
type Object_Identifier is new POSIX.Octet_Array;
type Application_Context_Name is new Object_Identifier;
type Presentation_Context_List is private;
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Application_Context_Name;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    Value       : in     Application_Context_Name);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Presentation_Context_List;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    Value       : in     Presentation_Context_List);
-- Presentation Context Definition and Result List
type Presentation_Context_Item is private;
type Presentation_Item_Id is implementation-defined-integer;
procedure Set_Presentation_Id
   (Item : in out Presentation_Context_Item;
    To   : in     Presentation_Item_Id);
function Get_Presentation_Id
   (Item : Presentation_Context_Item)
  return Presentation_Item_Id;
type Negotiation_Result is private;
Presentation_Context_Accepted : constant Negotiation_Result;
Presentation_Context_Rejected : constant Negotiation_Result;
Rejected_No_Reason_Specified  : constant Negotiation_Result;
Abstract_Syntax_Not_Supported : constant Negotiation_Result;
Transfer_Syntax_Not_Supported : constant Negotiation_Result;
Local_DCS_Limit_Exceeded      : constant Negotiation_Result;
```

```
procedure Set_Negotiation_Result
   (Item : in out Presentation_Context_Item;
    To   : in     Negotiation_Result);
function Get_Negotiation_Result
   (Item : Presentation_Context_Item)
  return Negotiation_Result;
type Syntax_Object_List is private;
procedure Set_Syntax_Object
   (Item : in out Presentation_Context_Item;
    To   : in     Syntax_Object_List);
function Get_Syntax_Object
   (Item : Presentation_Context_Item)
  return Syntax_Object_List;
Empty_Presentation_Context_List : constant Presentation_Context_List;
procedure Make_Empty (List : in out Presentation_Context_List);
procedure Add
   (List : in out Presentation_Context_List;
    Item : in     Presentation_Context_Item);
generic
    with procedure Action
       (Item : in out Presentation_Context_Item;
        Quit : in out Boolean);
procedure For_Every_Presentation_Context_Item
   (List : in Presentation_Context_List);
function Length (List : Presentation_Context_List)
  return Natural;
function Element
   (List  : Presentation_Context_List;
    Index : Positive)
  return Presentation_Context_Item;
Empty_Syntax_Object_List : constant Syntax_Object_List;
procedure Make_Empty (List : in out Syntax_Object_List);
procedure Add
   (List : in out Syntax_Object_List;
    Item : in    Object_Identifier);
generic
    with procedure Action
       (Object : in     Object_Identifier;
        Quit   : in out Boolean);
procedure For_Every_Object_Identifier
   (List : in Syntax_Object_List);
function Length (List : Syntax_Object_List)
  return Natural;
function Element
   (List  : Syntax_Object_List;
    Index : Positive)
  return Object_Identifier;
```

D.2.1.5.2 Description

Options are formatted according to the `Protocol_Option` object as described in 17.3.2. The overloaded `Get_Value` function and `Set_Option` procedure are used to examine and set attributes in the `Protocol_Option` object. An OSI provider compliant with this specification supports all, none, or a subset of the options defined in D.2.1.13 and D.2.1.14. An implementation may restrict the use of any of the options by offering them in privileged or read-only mode. The following options are common to both connection-mode and connectionless-mode service:

Application Context Name

This option defines the application context name (see D.2.1.1). Its value is an object with type `Application_Context_Name`.

Presentation Context List

This option defines the Presentation Context Definition and Result List (see D.2.1.3). Its value is a `Presentation_Context_List` object, which is a list of `Presentation_Context_Item`s. The generic procedure `For_Every_Presentation_Context_Item`, described later in this subclause, provides a way to iterate through this list. Each `Presentation_Context_Item` includes the following attributes:

Presentation Item Id

A unique value identifying the item. The procedure `Set_Presentation_Id` and the function `Get_Presentation_Id` are used to set and retrieve the value of this attribute.

Negotiation Result

A value with type `Negotiation_Result` containing one of the following negotiation results:

`Presentation_Context_Accepted`
: Accepted

`Presentation_Context_Rejected`
: Rejected by peer application

`Rejected_No_Reason_Specified`
: Provisional reject: no reason given

`Abstract_Syntax_Not_Supported`
: Provisional reject: abstract syntax not supported

`Transfer_Syntax_Not_Supported`
: Provisional reject: transfer syntax not supported

`Local_DCS_Limit_Exceeded`
: Provisional reject: local limit on the DCS exceeded

The procedure `Set_Negotiation_Result` and the function `Get_Negotiation_Result` are used to set and retrieve the value of this attribute.

Syntax Object Identifier List

An object with type `Syntax_Object_List` containing a list of object identifiers (see D.2.1.10) for the syntax elements. The procedure `Set_Syntax_Object` and the function `Get_Syntax_Object` are used to set and retrieve the value of this attribute.

The first element in this list shall refer to the abstract syntax, the second to the first transfer syntax, and so on. The generic procedure `For_Every_Object_Identifier`, described later in this subclause, provides a way to iterate through this list.

`Presentation_Context_List` and `Syntax_Object_List` objects shall have the initial values `Empty_Presentation_Context_List` and `Empty_Syntax_Object_List`, respectively. The procedure `Make_Empty` shall set the list indicated by `List` to the appropriate initial value, freeing any dynamically allocated storage associated

with the object. The procedure `Add` shall add the item specified by `Item` to the list indicated by `List`. The first item added after a `Make_Empty` shall be the first item in the list. The order of subsequent items shall be the order they are added with `Add`. The add order is preserved until the next `Make_Empty` operation. The functions `Length` and `Element` return the length and nth item of a list, respectively.

The application program instantiates the generic procedures `For_Every_Presentation_Context_Item` and `For_Every_Object_Identifier` with an actual procedure for the generic formal procedure `Action`. When called, the instance shall call the actual procedure supplied for `Action` once for each element in the associated list.

`Action` shall be able to force termination of the generic instance either by setting `Quit` to `True` or by raising an exception. Prior to calling `Action`, the instance shall set `Quit` to `False`. Exceptions raised by `Action` shall terminate iteration and shall be propagated back to the caller of the instance. After an exception is raised by `Action` or `Action` returns with `Quit` set to `True`, no more calls to `Action` within this call of the generic procedure shall occur.

Some of the XTI QOS options defined for the ISO transport connection-mode or connectionless-mode Service may be made available to mOSI applications. These options are defined in D.2.2.2 and D.2.2.3.

The QOS parameters are passed directly by the OSI upper layers to the transport layer. These options can thus be used to specify OSI upper layers quality of service parameters via XTI.

This facility is implementation dependent. If an attempt is made to specify an unsupported option, `Manage_Options` returns `Not_Supported` as the Status attribute for the `Protocol_Option` object.

None of these options shall be available with an ISO-over-TCP communications provider.

D.2.1.6 Functional Units, Versions and Protocol Mechanisms

The implementation shall negotiate:

— Session: kernel, full duplex, version 2, or version 1 if version 2 not supported, no segmentation.

  Other session protocol mechanisms are out of scope, except basic concatenation which is mandatory and transparent to the application.

— Presentation: kernel, normal mode

— ACSE: kernel

If invalid (nonnegotiable) options are requested by the peer and detected by the provider once the association is already established (such as the ACSE presentation context missing in the DCS), the association shall be rejected via an A-(P)-ABORT generated by the implementation.

D.2.1.7 Mandatory and Optional Parameters

If the local Presentation Address is not passed to `Bind` in the `Request_Address` parameter, then it shall be returned in the `Response_Address` parameter.

The remote (called) Presentation Address shall be explicitly set by the application. In `Connect`, this is the Address attribute of the `Send` parameter (`Connection_Info` object).

The following parameters are mandatory for the protocol machine, but default values shall be provided. If the application does not set the corresponding parameter, the default value shall be used. The default value can be changed through `Manage_Options` (see D.2.1.5).

— Application Context Name

— Presentation Context List

The presentation context of ACSE is required and used. The application should not request it as the implementation will insert it automatically in the context list.

If the application does not specifically request an application context name via the Option attribute of the `Call` parameter of `Accept_Connection` (that is, for the A-ASSOCIATE response), the implementation shall use the application context name that was received in the A-ASSOCIATE indication.

The following parameters are optional for the protocol. If the application does not set them otherwise, they shall be omitted from the outgoing protocol stream.

— Local AP Title (in `Bind`, Address attribute of the `Request` parameter)

— Called AP Title (in `Connect`, Address attribute of the Send parameter)

— Responding AP Title (if `Accept_Connection` specifies a new accepting endpoint, in the protocol address bound to `Responding_Endpoint`)

— Local AE Qualifier (in `Bind`, the Address attribute of the `Request` parameter)

— Called AE Qualifier (in `Connect`, the Address attribute of the `Send` parameter)

— Responding AE Qualifier (if `Accept_Connection` specifies a new accepting endpoint, in the protocol address bound to `Responding_Endpoint`)

— Local AP Invocation Identier and AE Invocation Identifier (in `Bind`, Address attribute of the `Request` parameter)

— Called AP Invocation Identifier and AE Invocation Identifiers (in `Connect`, Address attribute of the `Send` parameter)

— Responding AP Invocation Identifier and AE Invocation Identifier (if `Accept_Connection` specifies a new accepting endpoint, in the protocol address bound to `Responding_Endpoint`).

The following parameters are optional for the protocol machine and not supported through the XTI interface. Their handling is implementation defined. Received values in the incoming protocol stream, if any, are discarded:

— ACSE Protocol Version (default = version 1)

— Presentation Protocol Version (default = version 1)

— ACSE Implementation Information

— Session Connection Identifiers

During association establishment (that is, before the XTI-mOSI provider negotiates acceptance of a single abstract syntax/transfer syntax pair), an XTI-mOSI application initiating the association shall only send a single presentation data value in the application information parameter. The XTI-mOSI provider shall ensure that the first abstract syntax and transfer syntax pair being negotiated is the one required for its encoding.

D.2.1.8 Association Establishment

XTI shall not support the concept of a negative association establishment, *i.e.*, the equivalent of a negative A-ASSOCIATE response. An XTI-mOSI implementation shall not generate an AARE-APDU.

To reject an association request, the responding application issues a `Send_Disconnect_Request`, which maps to an A-ABORT. However, a negative A-ASSOCIATE confirm (AARE-APDU) may be received from a non-XTI OSI peer. The negative A-ASSOCIATE confirm event is mapped to `Retrieve_Disconnect_Info`.

D.2.1.9 Encoding Responsibility

The application shall be responsible for encoding and decoding the AP Title and AE Qualifier.

Where an object identifier is represented within an option, the application is responsible for encoding and decoding the object identifier value.

D.2.1.10 Object Identifiers

For the default abstract syntax, transfer syntax, and application context, this annex uses object identifiers that are specified in ISO/IEC ISP 11188-3 {11}. The descriptions provided in this Annex are informative only.

— The following ASN.1 object identifier is defined in ISO/IEC ISP 11188-3 {11} for the default abstract syntax for mOSI.

{*iso(1) standard(0) culr(11188) mosi(3) default-abstract-syntax(1) version(1)*}

This object identifier can be used as the abstract syntax when the application protocol (above ACSE) can be treated as single PDVs.

Each PDV is a sequence of consecutive octets without regard for semantic or other boundaries. The object identifier can also be used when, for pragmatic reasons, the actual abstract syntax of the application is not identified in presentation layer negotiation.

NOTE: Applications specified using ASN.1 should not use the default abstract syntax.

NOTE: As this object identifier is used by all applications using the default abstract syntax for mOSI, it cannot be used to differentiate between applications. One of the ACSE parameters, for example, AE Title or Presentation Address, can be used to differentiate between applications.

— If the default transfer syntax and the abstract syntax are identical, the ASN.1 object identifier for the default abstract syntax is used as the object identifier for the default transfer syntax for mOSI. If they are not identical, the ASN.1 object identifier for the default transfer syntax shall be as follows:

{*iso(1) standard(0) culr(11188) mosi(3) default-transfer-syntax(2) version(1)*}

In the presentation data value of the PDV list of the ISO presentation protocol or in the encoding of User Information of the ACSE Protocol, only octet-aligned or arbitrary shall be used for the default transfer syntax for mOSI. The syntax *single-ASN1-type* shall not be used for the default transfer syntax for mOSI.

— The following ASN.1 object identifier is defined in ISO/IEC ISP 11188-3 {11} for the default application context for mOSI.

{*iso(1) standard(0) culr(11188) mosi(3) default-application-context(3) version(1)*}

This application context supports the execution of any application using the default abstract syntax for mOSI.

D.2.1.11 XTI Functions

D.2.1.11.1 Synopsis

```
Rejected_By_Peer          :
   constant POSIX_XTI.Reason_Code := implementation-defined;
AC_Name_Not_Supported     :
   constant POSIX_XTI.Reason_Code := implementation-defined;
Unrecognized_AP_Title     :
   constant POSIX_XTI.Reason_Code := implementation-defined;
Unrecognized_AE_Qualifier :
   constant POSIX_XTI.Reason_Code := implementation-defined;
Authentication_Required   :
   constant POSIX_XTI.Reason_Code := implementation-defined;
Aborted_By_Peer           :
   constant POSIX_XTI.Reason_Code := implementation-defined;
No_Common_Version         :
   constant POSIX_XTI.Reason_Code := implementation-defined;
```

D.2.1.11.2 Description

The meanings of the constants of type `Reason_Code` are as follows:

`Accept_Connection`

If `Listening_Endpoint` is not equal to `Responding_Endpoint`, then `Responding_Endpoint` should either be in the Unbound state or be in the Idle state and be bound to the same address as `Listening_Endpoint` with the endpoint queue length parameter set to zero.

The address passed to `Bind` in `Request_Address` or returned from `Bind` in `Response_Address` when `Responding_Endpoint` is bound can be different from the address corresponding to `Listening_Endpoint`. The Options attribute of the `Connection_Info` object can be used to change the application context name received.

`Bind`

The `Request_Address` parameter represents the local Presentation Address and optionally the local AP Title and local AE Qualifier.

This local Address attribute shall be used, depending on the XTI primitive, as the calling, called, or responding address. The called address is different from the responding address only when two different file descriptors (*e.g.*, `Listening_Endpoint`, `Responding_Endpoint`) bound to different addresses are used.

`Close`

Any connections that are still active at the endpoint shall be abnormally terminated. The peer applications shall be informed of the disconnection by a `Disconnect_Request_Received` event. The value of the disconnect reason shall be `Aborted_By_Peer`.

`Connect`

The Address attribute of the `Send` parameter (`Connection_Info` object) specifies the called Presentation Address. The Address attribute of the `Receive` parameter specifies the responding Presentation Address, and can also be used to assign values for the called AP Title, called AE Qualifier, called AP Invocation Identifier and called AE Invocation Identifier. The Options attribute of the `Send` parameter can be used to request an application context name or presentation context different from the default value.

`Get_Address`

The Address attribute returned by `Get_Address` from a `Connection_Info` object shall be a protocol-specific `mOSI_XTI_Address` object.

`Get_Info`

The information supported by `Get_Info` shall reflect the characteristics of the transport connection or, if no connection is established, the default characteristics of the underlying OSI layers. In all possible states except Data Transfer, the function `Get_Info` shall return, in parameter `Communications_Provider_Info`, the same information as was returned by `Open`. In the Data Transfer state, however, the information returned in the Max Size Connect Data and Max Size Disconnect Data attributes may differ.

The parameters of the `Get_Info` function are summarized in Table D.2. The attribute values of the `Communications_Provider_Info` object for the `Get_Info` function shall reflect the mOSI provider particularities. The values returned in the Max Size Connect Data and Max Size Disconnect Data attributes in the Data Transfer state may differ from the values returned by `Open`. Negotiation takes place during association establishment; and, as a result, these values may be reduced. For Max Size Connect Data, this change of value may be indicated by the provider, but is of little use to the application. Sending SDUs of zero length is not supported by mOSI. Therefore, the value of the corresponding flag in the CP Flags attribute shall be zero.

`Get_Protocol_Address`

The protocol addresses are naming and addressing parameters as defined in D.2.1.4.

`Get_Current_State`

There are no special considerations for mOSI providers.

`Listen`

The Address attribute of the `Call` parameter (`Connection_Info` object) contains the remote calling Presentation Address, and optionally the remote

calling AP Title and calling AE Qualifier, and the calling AP Invocation Identifier and calling AE Invocation Identifier if received.

Incoming application data encoded as multiple presentation data values shall generate the `Illegal_Data_Range` error.

`Look`

Since expedited data are not supported for an mOSI provider, `Expedited_Data_Received` and `Okay_To_Send_Expedited_Data` events cannot occur.

`Open`

`Open` shall be called as the first step in the initialization of a communications endpoint. This function shall return various default characteristics of the underlying OSI layers.

The parameters of the `Open` procedure are summarized in Table D.2. The values of the parameters in the `Communications_Provider_Info` object shall reflect mOSI limitations as follows. The values returned in the Max Size Connect Data and Max Size Disconnect Data attributes shall be limited by the version of the session supported by the mOSI provider.

NOTE: They are generally much larger than those supported by an ISO transport or TCP communications provider.

Sending SDUs of zero length is not supported by mOSI. Therefore, the value of the corresponding flag in the CP Flags attribute shall be zero.

NOTE: The name (device file) parameter passed to `Open` will be different when the application accesses an mOSI provider from what it is when the application accesses an ISO transport communications provider.

Table D.2 – `Communications_Provider_Info` Returned by `Get_Info` and `Open`, mOSI

Attribute	Connection Mode	Connectionless Mode
Max_Size_Protocol_Address	*x* (1)	*x* (1)
Max_Size_Protocol_Options	*x* (1)	*x* (1)
Max_Size_Service_Data_Unit	*infinite* (2)	*infinite* (2)
Max_Size_Service_Expedited_Data_Unit	*invalid* (3)	*invalid* (3)
Max_Size_Connect_Data	*x* (1)	*invalid* (3)
Max_Size_Disconnect_Data	*x* (1)	*invalid* (3)
Service_Type	`Connectionless_Mode_With_Orderly_Release`	`Connectionless_Mode`
CP_Flags	`Empty_Set`	`Empty_Set`

NOTES:

(1) Either the corresponding function (`Protocol_Addresses_Are_Valid`, `Protocol_Options_Are_Valid`, `Connect_Data_Is_Valid`, `Disconnect_Data_Is_Valid`) returns `False` or the value of the attribute is set to an integral number *x* greater than zero.

(2) The corresponding function (`SDU_Is_Infinite`) returns `True`.

(3) The corresponding function (`SDU_Is_Valid`, `Connect_Data_Is_Valid`, `Disconnect_Data_Is_Valid`) returns `False`.

`Manage_Options`

The options available with mOSI providers are described in D.2.1.5.

`Receive`

The `Flags` parameter shall not be set to `Expedited_Data`, as expedited data transfer is not supported.

`Confirm_Connection`

The Address attribute of the `Call` parameter (`Connection_Info` object) shall specify the remote responding presentation address, the remote responding AP Title, the AE Qualifier, the AP Invocation Identifier and the AE Invocation Identifier if present.

The Options attribute can also contain an Application Context Name and/or Presentation Context Definition and Result List.

`Retrieve_Disconnect_Info`

Possible values for disconnect reason codes (as returned by `Retrieve_Disconnect_Info`) are as follows:

`Rejected_By_Peer`
: Connection rejected by peer application: no reason given

`AC_Name_Not_Supported`
: Connection rejected: application context name not supported

`Unrecognized_AP_Title`
: Connection rejected: AP Title not recognized

`Unrecognized_AE_Qualifier`
: Connection rejected: AE Qualifier not recognized

`Authentication_Required`
: Connection rejected: authentication required

`Aborted_By_Peer`
: Aborted by peer provider: no reason given

`No_Common_Version`
: Connection rejected: no common version

`Acknowledge_Orderly_Release`

With this primitive, application data cannot be received on normal release; any application data in the received flow shall be discarded.

`Receive_Data_Unit`

The Address parameter shall specify the remote Presentation Address and, optionally: the remote AP Title, the AE Qualifier, the AP Invocation Identifier and the AE Invocation Identifier. If the `More_Data` flag is set, an additional `Receive_Data_Unit` call is needed to retrieve the entire AUNIT-DATA service unit. Only normal data shall be returned via the `Receive_Data_Unit` call.

`Retrieve_Data_Unit_Error`

This procedure is not supported by a mOSI provider since badly formed A-UNIT-DATA APDUs are discarded.

`Send`

Zero-length SDUs are not supported.

Since expedited data transfer is not supported for a mOSI provider, the `Flags` parameter shall not have `Expedited_Data` set.

`Send_Disconnect_Request`

There are no special considerations for mOSI providers.

`Initiate_Orderly_Release`

With this primitive, application data cannot be sent on normal release.

`Send_Data_Unit`

The Address parameter shall specify the remote Presentation Address and, optionally: the remote AP Title, the AE Qualifier, the AP Invocation Identifier, and AE Invocation Identifiers. Only normal data shall be sent via the `Send_Data_Unit` call.

`Synchronize_Endpoint`

There are no special considerations for mOSI providers.

`Unbind`

There are no special considerations for mOSI providers.

D.2.1.12 Mapping XTI Functions to ACSE/Presentation Services

The mapping of XTI functions to ACSE/Presentation services is shown in Table D.3, Table D.4, Table D.5, and Table D.6. The entries in these tables are to be interpreted as follows:

— The definition of which parameters are mandatory and which are optional can be found in ISO/IEC ISP 11188-3 {11}.

— An entry {Bind} in the XTI Call column indicates that the information is passed across the API when `Bind` is called and is passed across the communications interface following a subsequent call to another XTI function.

— An entry {internal} in the XTI Call column indicates that the information is not explicitly passed across the API but affects the behavior of the implementation.

— An entry {discarded} in the XTI Call column indicates that the information is discarded by the API implementation.

— The notation “req” in the Service column indicates a request.

— The notation “ind” in the Service column indicates an indication.

— The notation “rsp+” in the Service column indicates a positive response.

— The notation “rsp-” in the Service column indicates a negative response.

— The notation “cnf+” in the Service column indicates a positive confirmation.

— The notatino “cnf-” in the Service column indicates a negative confirmation.

Table D.3 – XTI and ACSE/Presentation Services

XTI call	Parameter	Service	Parameter
Connect		A-ASSOCIATE req	
	Send/Address		Called Presentation Address
	Send/Address (1)		Called AP Title
	Send/Address (1)		Called AE Qualifier
	Send/Address		Called AP Invocation Identifier
	Send/Address		Called AE Invocation Identifier
	Send/Options (2)		Application Context Name
	Send/Options (3)		Presentation Context Definition and Result List
	Send/User_Data		User Information
{*Bind*}	*Request&Return/Address*		Calling Presentation Address
{*Bind*}	*Request&Return/Address*		Calling AP Title
{*Bind*}	*Request&Return/Address*		Calling AE Qualifier
Listen		A-ASSOCIATE ind	
	Call/Address		Calling Presentation Address
	Call/Address (1)		Calling AP Title
	Call/Address (1)		Calling AE Qualifier
	Call/Options		Application Context Name
	Call/Options (4)		Presentation Context Definition and Result List
	Call/User_Data		User Information
{*Bind*}	*Request&Return/Address*		Called Presentation Address
{*Bind*}	*Request&Return/Address (1)*		Called AP Title
{*Bind*}	*Request&Return/Address (1)*		Called AE Qualifier
{*Bind*}	*Request&Return/Address*		Calling AP Invocation Identifier
{*Bind*}	*Request&Return/Address*		Calling AE Invocation Identifier
Accept		A-ASSOCIATE rsp+	
	Call/Address(not used)		Calling Presentation Address
	Call/Options		Application Context Name
	Call/Options		Presentation Context Definition and Result List
	Call/User_Data		User Information
{*internal*}	*::="accepted"*		Result
{*Bind*}	*Request&Return/Address*		Responding Presentation Address
{*Bind*}	*Request&Return/Address (1)*		Responding AP Title
{*Bind*}	*Request&Return/Address (1)*		Responding AE Qualifier
{*Bind*}	*Request&Return/Address*		Responding AP Invocation Identifier
{*Bind*}	*Request&Return/Address*		Responding AE Invocation Identifier

Table D.3 – XTI and ACSE/Presentation Services *(continued)*

{*not sent*}		A-ASSOCIATE rsp-	
Connect	*(synchronous mode)*	A-ASSOCIATE cnf+	
Connect	*(synchronous mode)*	A-ASSOCIATE cnf+	
	Receive/Address		Responding Presentation Address
	Receive/Address		Responding AP Title
	Receive/Address		Responding AE Qualifier
	Receive/Address		Responding AP Invocation Identifier
	Receive/Address		Responding AE Invocation Identifier
	Receive/Options		Application Context Name
	Receive/Options		Presentation Context Definition and Result List
	Receive/User_Data		User Information
{*internal*}	*::="accepted"*		Result
{*internal*}	*::="ACSE service-user"*		Result Source
Confirm_-Connection	*(asynchronous mode)*	A-ASSOCIATE cnf+	
	Call/Address		Responding Presentation Address
	Call/Address		Responding AP Title
	Call/Address		Responding AE Qualifier
	Call/Address		Responding AP Invocation Identifier
	Call/Address		Responding AE Invocation Identifier
	Call/Options		Application Context Name
	Call/Options		Presentation Context Definition and Result List
	Call/User_Data		User Information
{*discarded*}	*::="accepted"*		Result
{*discarded*}	*::="ACSE service-user"*		Result Source-diagnostic
Retrieve_-Disconnect_-Information		A-ASSOCIATE cnf-	
	Info/User_Data		User Information
	Info/Reason (5)		Result
{*internal*}	*ACSE serv-user/pres serv-prov*		Result Source-diagnostic
{*discarded*}			Application Context Name
{*discarded*}			Presentation Context Definition and Result List

NOTES:

(1) If either the AP Title or AE Qualifier is selected for sending, the other must be selected.

(2) Options attribute or, if no option specified, default value.

(3) Options attribute or, if no option specified, default value, with ACSE added by provider.

(4) Options attribute with ACSE context removed from the list passed to application.

(5) Combines Result and Result Source-diagnostic.

Table D.4 – XTI mOSI Connection-Mode Data Transfer Services

XTI call	Parameter	Service	Parameter
Send		P-DATA req	
	Buffer		User Data
Receive		P-DATA ind	
	Buffer		User Data

Table D.5 – XTI and Association Release Services

XTI call	Parameter	Service	Parameter
Initiate_Orderly_Release_With_Data		A-RELEASE req	
	Info/Reason		Reason
	Info/User_Data		User Information
Acknowledge_Orderly_Release_With_Data		A-RELEASE ind	
	Info/Reason		Reason
	Info/User_Data		User Information
Initiate_Orderly_Release_With_Data		A-RELEASE rsp	
	Info/Reason		Reason
	Info/User_Data		User Information
Acknowledge_Orderly_Release_With_Data		A-RELEASE cnf	
	Info/Reason		Reason
	Info/User_Data		User Information
Send_Disconnect_Request		A-ABORT req	
	n/s		Diagnostic
	Info/User_Data		User Information
Retrieve_Disconnect_Info		A-ABORT ind	
	Info/Reason		Diagnostic
	Info/User_Data		User Information
Retrieve_Disconnect_Info		A-P-ABORT ind	
	Info/Reason		Diagnostic

Table D.5 relies on the assumption that the XTI-mOSI provider supports the orderly release facility with application data (*i.e.*, `Initiate_Orderly_Release_With_Data` and `Acknowledge_Orderly_Release_With_Data`). When orderly release is not supported, User Data shall not be sent, Reason shall be supplied via an internal mechanism with A-RELEASE request and response, and User Data and Reason received in A-RELEASE indication and confirmation shall be discarded.

D.2.1.13 Connection-Mode Service

Figure D.4 shows the XTI state diagram for connection-mode mOSI communications (see also Figure 17.1).

The protocol level for all subsequent options is `mOSI_Connection_Mode`.

All options have end-to-end significance (see 17.3). They can be negotiated in the XTI states Idle and Incoming Connect, and shall be read-only in all other states except Uninitialized.

The objects referenced are specified in D.2.1.5.

Table D.6 – XTI Connectionless-Mode ACSE Services

XTI call	Parameter	Service	Parameter
Send_-Data_Unit		A-UNIT-DATA source	
	Data/Address		Called Presentation Address
	Data/Address		Called AP Title
	Data/Address		Called AE Qualifier
	Data/Address		Called AP Invocation Identifier
	Data/Address		Called AE Invocation Identifier
	Data/Options (1)		Application Context Name
	Data/Options (2)		Presentation Context Definition and Result List
	Data/User_Data		User Information
{*Bind*}	*Request&Return/Address*		Calling Presentation Address
{*Bind*}	*Request&Return/Address*		Calling AP Title
{*Bind*}	*Request&Return/Address*		Calling AE Qualifier
{*Bind*}	*Request&Return/Address*		Calling AP Invocation Identifier
{*Bind*}	*Request&Return/Address*		Calling AE Invocation Identifier
Receive_-Data_Unit		A-UNIT-DATA sink	
	Data/Address		Calling Presentation Address
	Data/Address		Calling AP Title
	Data/Address		Calling AE Qualifier
	Data/Address		Calling AP Invocation Identifier
	Data/Address		Calling AE Invocation Identifier
	Data/Options		Application Context Name
	Data/Options (3)		Presentation Context Definition and Result List
	Data/User_Data		User Information
{*Bind*}	*Request&Return/Address*		Called Presentation Address
{*Bind*}	*Request&Return/Address*		Called AP Title
{*Bind*}	*Request&Return/Address*		Called AE Qualifier
{*Bind*}	*Request&Return/Address*		Called AP Invocation Identifier
{*Bind*}	*Request&Return/Address*		Called AE Invocation Identifier

NOTES:

(1) Data/Options or, if no option specified, default value.

(2) Data/Options or, if no option specified, default value with ACSE added by provider.

(3) Data/Options with ACSE context removed from the list passed to application.

The following options are defined for connection-mode mOSI communications.

Application Context Name

This option shall consist of an object identifier in BER encoded form. It shall identify the application context name.

A default value shall be provided for the Application Context Name XTI option option. It denotes the application context name for a generic XTI-mOSI application. It is defined in D.2.1.5.

The application can propose through the Application Context Name XTI option

option a value different from the default one. The application can also use this option to check the value returned by the peer application and decide whether the association should be kept or terminated.

Presentation Context List

This option is used to propose a presentation context, giving its abstract and transfer syntax, and to hold the result of negotiation of a presentation context. It is a variable sized option. Its format is described in D.2.1.5.

A default shall be provided for the Presentation Context List XTI option option. This default value is a list with one presentation context: the stream-oriented, unstructured, data transfer service with null encoding. The abstract syntax is the default abstract syntax and the transfer syntax is the default transfer syntax, as specified in D.2.1.5. The codes for the result of negotiation and reason for rejection are defined in D.2.1.5. After reading the Presentation Context List XTI option, the responding applicatiion can either continue or terminate the association.

Only a single abstract syntax and transfer syntax shall be used by XTI-mOSI. When `Accept_Connection` is called, this syntax is assumed to be the first usable abstract syntax and the first transfer syntax for that abstract syntax.

When initiating a connection, the application shall propose one or more presentation contexts, each comprising an abstract syntax and one or more transfer syntaxes, in the Presentation Context Definition and Result List XTI option (or shall omit this option to select the defaults) and shall call `Connect`.

If the connection is accepted, the Presentation Context Definition and Result List XTI option shall be updated to reflect the results of negotiation for each element of the presentation context list, and a single presentation context shall be selected.

If the responder accepts multiple presentation contexts, the XTI-mOSI provider shall abort the connection on receipt of the A-ASSOCIATE confirm.

When responding to a connection request, the application can specifically mark presentation contexts as rejected using the Result attribute and can reorder the syntax array to select a single transfer syntax.

When `Accept_Connection` is called, the first presentation context marked as accepted shall be selected, and all other contexts omitted or not marked rejected-user are marked as by the provider as rejected (`Local_DCS_Limit_Exceeded`). In an accepted context, the provider shall accept the first (or only remaining) transfer syntax.

On return to the application from `Listen`, all supportable presentation contexts shall be marked as accepted in the Presentation Context List XTI option, and all unsupportable contexts shall be marked as rejected-provider. Having the contexts already marked as accepted or rejected-provider permits the application to return the same option value on `Accept_Connection` (or leave it unchanged) in order to select the first available abstract syntax and transfer syntax.

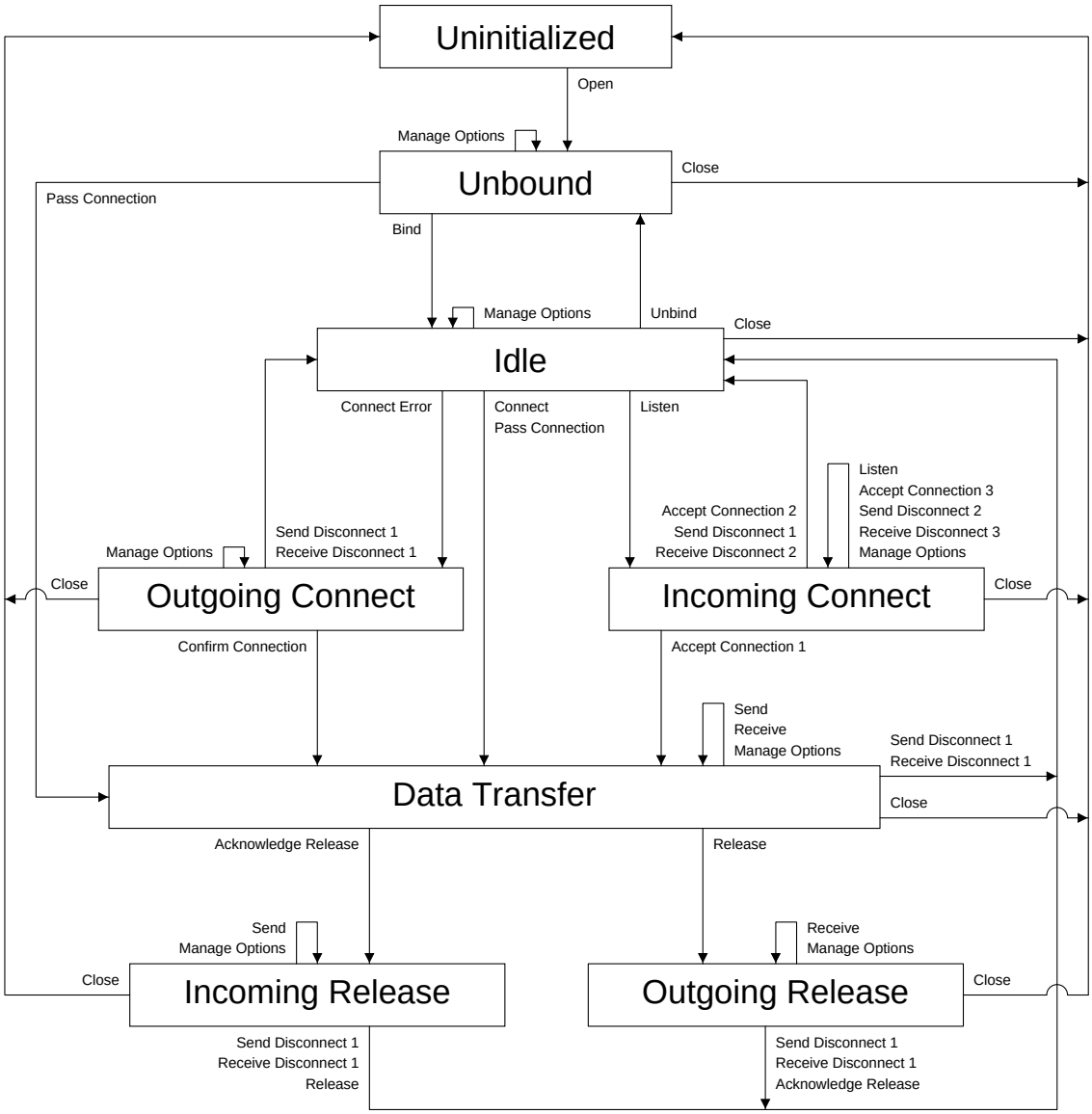

Figure D.4 – Connection-Mode XTI State Diagram

D.2.1.14 Connectionless-Mode Service

Figure D.5 shows the XTI state diagram for connectionless-mode mOSI communications (see also Figure 17.1).

The protocol level for all subsequent options is `mOSI_Connectionless_Mode`.

All options have end-to-end significance (see 17.3). They may be negotiated in all XTI states except Uninitialized. The objects referenced are specified in D.2.1.5.

The following options are defined for connectionless-mode mOSI communications:

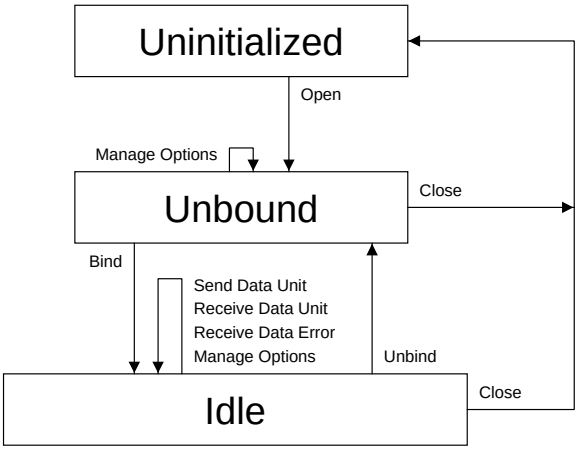

Figure D.5 – Connectionless-Mode XTI State Diagram

Application Context Name

This option shall consist of an object identifier in BER encoded form. It shall identify the application context name. Its format is described in D.2.1.5.

A default value shall be provided for the Application Context Name XTI option option. It denotes the application context name for a generic XTI-mOSI application. The application can propose through this option a value different from the default one. The application can also use this option to check the value returned by the peer application and decide whether the datagram should be kept or discarded.

In connectionless mode, the transfer syntaxes are not negotiated. Their use is determined by the sending application entity and must be acceptable to the receiving application entity.

Presentation Context List

This option is used to propose a presentation context, giving its abstract and transfer syntax. It is a variable sized option. Its format is described in D.2.1.5.

A default value is provided for the Presentation Context List XTI option option. This default value is a list with one element, the generic presentation context (the stream-oriented, unstructured, data transfer service with null encoding). The corresponding abstract and transfer syntaxes are specified in D.2.1.5. Only a single abstract syntax and transfer syntax can be used by connectionless XTI-mOSI. If more than one presentation context is present in the options list for `Send_Data_Unit`, the first shall be used.

D.2.2 Package `POSIX_XTI_ISO`

This package provides the DNI/XTI interface mappings for ISO transport protocols. Unless otherwise specified, all the DNI/XTI calls in package `POSIX_XTI` can be used for this protocol. Only additional information relevant to this protocol is highlighted here.

The functionality described in this subclause is optional. If either the XTI Detailed Network Interface option or the ISO/OSI Protocol option is not supported, the implementation

may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

```
with POSIX,
     POSIX_XTI;
package POSIX_XTI_ISO is
   -- D.2.2.1 ISO Transport Protocols
   ISO_TP_Level :  constant POSIX_XTI.Option_Level := implementation-defined;
   type ISO_XTI_Address is private;
   type ISO_XTI_Address_Pointer is access all ISO_XTI_Address;
   function "+" (Pointer : ISO_XTI_Address_Pointer)
     return POSIX_XTI.XTI_Address_Pointer;
   function "+" (Pointer : POSIX_XTI.XTI_Address_Pointer)
     return ISO_XTI_Address_Pointer;
   function Is_ISO_XTI_Address
      (Pointer : POSIX_XTI.XTI_Address_Pointer)
     return Boolean;
   type ISO_Option is (Enabled,Disabled);
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
     return ISO_Option;
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
       Name        : in     POSIX_XTI.Option_Name;
       To          : in     ISO_Option);
   Residual_Error_Rate : constant POSIX_XTI.Option_Name := implementation-defined;
   Priority            : constant POSIX_XTI.Option_Name := implementation-defined;
   Protection          : constant POSIX_XTI.Option_Name := implementation-defined;
   type Rate is private;
   subtype Error_Rate is Positive
      range implementation-defined;
   Unspecified_Rate : constant Duration := implementation-defined;
   type Priority_Level is private;
   Top     : constant Priority_Level;
   High    : constant Priority_Level;
   Medium  : constant Priority_Level;
   Low     : constant Priority_Level;
   Default : constant Priority_Level;
   type Protection_Level is new POSIX.Option_Set;
   No_Protection        : constant Protection_Level := implementation-defined;
   Passive_Protection   : constant Protection_Level := implementation-defined;
   Active_Protection    : constant Protection_Level := implementation-defined;
   Absolute_Requirement : constant Protection_Level := implementation-defined;
   function Get_Target_Rate (Item : Rate)
     return Duration;
   procedure Set_Target_Rate
      (Item : in out Rate;
       To   : in     Duration);
   function Get_Minimum_Acceptable_Rate (Item : Rate)
     return Duration;
   procedure Set_Minimum_Acceptable_Rate
      (Item : in out Rate;
       To   : in     Duration);
   function Get_Target_Rate (Item : Rate)
     return Error_Rate;
   procedure Set_Target_Rate
      (Item : in out Rate;
       To   : in     Error_Rate);
```

```
function Get_Minimum_Acceptable_Rate (Item : Rate)
  return Error_Rate;
procedure Set_Minimum_Acceptable_Rate
   (Item : in out Rate;
    To   : in     Error_Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Rate;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Priority_Level;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Priority_Level);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Protection_Level;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Protection_Level);
procedure Get_Address
   (Info_Item : in     POSIX_XTI.Connection_Info;
    Address   : in out ISO_XTI_Address);
-- D.2.2.2 Connection-Mode Service
Throughput :
   constant POSIX_XTI.Option_Name := implementation-defined;
Connection_Transit_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
Transfer_Fail_Probability :
   constant POSIX_XTI.Option_Name := implementation-defined;
Establishment_Fail_Probability :
   constant POSIX_XTI.Option_Name := implementation-defined;
Release_Fail_Probability :
   constant POSIX_XTI.Option_Name := implementation-defined;
Establishment_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
Release_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
Connection_Resilience :
   constant POSIX_XTI.Option_Name := implementation-defined;
Expedited_Data :
   constant POSIX_XTI.Option_Name := implementation-defined;
type Requested_Rate is private;
type Throughput_Rate is private;
type Transit_Delay_Rate is private;
function Get_Called_Rate (Item : Requested_Rate)
  return Rate;
procedure Set_Called_Rate
   (Item : in out Requested_Rate;
    To   : in     Rate);
function Get_Calling_Rate (Item : Requested_Rate)
  return Rate;
```

```
procedure Set_Calling_Rate
   (Item : in out Requested_Rate;
    To   : in     Rate);
function Get_Throughput_Maximum (Item : Throughput_Rate)
  return Requested_Rate;
procedure Set_Throughput_Maximum
   (Item : in out Throughput_Rate;
    To   : in     Requested_Rate);
function Get_Throughput_Average (Item : Throughput_Rate)
  return Requested_Rate;
procedure Set_Throughput_Average
   (Item : in out Throughput_Rate;
    To   : in     Requested_Rate);
function Get_Transit_Delay_Maximum (Item : Transit_Delay_Rate)
  return Requested_Rate;
procedure Set_Transit_Delay_Maximum
   (Item : in out Transit_Delay_Rate;
    To   : in     Requested_Rate);
function Get_Transit_Delay_Average (Item : Transit_Delay_Rate)
  return Requested_Rate;
procedure Set_Transit_Delay_Average
   (Item : in out Transit_Delay_Rate;
    To   : in     Requested_Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Throughput_Rate;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Throughput_Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Transit_Delay_Rate;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Transit_Delay_Rate);
TPDU_Length_Maximum :
   constant POSIX_XTI.Option_Name := implementation-defined;
Acknowledge_Time :
   constant POSIX_XTI.Option_Name := implementation-defined;
Reassignment_Time :
   constant POSIX_XTI.Option_Name := implementation-defined;
Preferred_Class :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_1 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_2 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_3 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_4 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Extended_Format :
   constant POSIX_XTI.Option_Name := implementation-defined;
Flow_Control :
   constant POSIX_XTI.Option_Name := implementation-defined;
```

```
   Connection_Checksum :
      constant POSIX_XTI.Option_Name := implementation-defined;
   Network_Expedited_Data :
      constant POSIX_XTI.Option_Name := implementation-defined;
   Network_Receipt_Confirmation :
      constant POSIX_XTI.Option_Name := implementation-defined;
   type ISO_COTS_Option is (Enabled, Disabled, Unspecified);
   type Transport_Class is private;
   Class_0           : constant Transport_Class;
   Class_1           : constant Transport_Class;
   Class_2           : constant Transport_Class;
   Class_3           : constant Transport_Class;
   Class_4           : constant Transport_Class;
   Class_Unspecified : constant Transport_Class;
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
     return ISO_COTS_Option;
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
       Name        : in     POSIX_XTI.Option_Name;
       To          : in     ISO_COTS_Option);
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
     return Transport_Class;
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
       Name        : in     POSIX_XTI.Option_Name;
       To          : in     Transport_Class);
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
     return Duration;
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
       Name        : in     POSIX_XTI.Option_Name;
       To          : in     Duration);
   -- D.2.2.3 Connectionless-Mode Service
   Connectionless_Transit_Delay :
      constant POSIX_XTI.Option_Name := implementation-defined;
   Connectionless_Checksum :
      constant POSIX_XTI.Option_Name := implementation-defined;
private
   implementation-defined
end POSIX_XTI_ISO;
```

D.2.2.1 ISO Transport Protocols

D.2.2.1.1 Synopsis

```
   ISO_TP_Level :  constant POSIX_XTI.Option_Level := implementation-defined;
   type ISO_XTI_Address is private;
   type ISO_XTI_Address_Pointer is access all ISO_XTI_Address;
   function "+" (Pointer : ISO_XTI_Address_Pointer)
     return POSIX_XTI.XTI_Address_Pointer;
   function "+" (Pointer : POSIX_XTI.XTI_Address_Pointer)
     return ISO_XTI_Address_Pointer;
   function Is_ISO_XTI_Address
      (Pointer : POSIX_XTI.XTI_Address_Pointer)
     return Boolean;
```

```
type ISO_Option is (Enabled,Disabled);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return ISO_Option;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     ISO_Option);
Residual_Error_Rate : constant POSIX_XTI.Option_Name := implementation-defined;
Priority            : constant POSIX_XTI.Option_Name := implementation-defined;
Protection          : constant POSIX_XTI.Option_Name := implementation-defined;
type Rate is private;
subtype Error_Rate is Positive
   range implementation-defined;
Unspecified_Rate : constant Duration := implementation-defined;
type Priority_Level is private;
Top     : constant Priority_Level;
High    : constant Priority_Level;
Medium  : constant Priority_Level;
Low     : constant Priority_Level;
Default : constant Priority_Level;
type Protection_Level is new POSIX.Option_Set;
No_Protection        : constant Protection_Level := implementation-defined;
Passive_Protection   : constant Protection_Level := implementation-defined;
Active_Protection    : constant Protection_Level := implementation-defined;
Absolute_Requirement : constant Protection_Level := implementation-defined;
function Get_Target_Rate (Item : Rate)
  return Duration;
procedure Set_Target_Rate
   (Item : in out Rate;
    To   : in     Duration);
function Get_Minimum_Acceptable_Rate (Item : Rate)
  return Duration;
procedure Set_Minimum_Acceptable_Rate
   (Item : in out Rate;
    To   : in     Duration);
function Get_Target_Rate (Item : Rate)
  return Error_Rate;
procedure Set_Target_Rate
   (Item : in out Rate;
    To   : in     Error_Rate);
function Get_Minimum_Acceptable_Rate (Item : Rate)
  return Error_Rate;
procedure Set_Minimum_Acceptable_Rate
   (Item : in out Rate;
    To   : in     Error_Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Rate;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Priority_Level;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Priority_Level);
```

```
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Protection_Level;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Protection_Level);
procedure Get_Address
   (Info_Item : in     POSIX_XTI.Connection_Info;
    Address   : in out ISO_XTI_Address);
```

D.2.2.1.2 Description

This subclause describes the protocol-specific information that is relevant for communications providers that provide the ISO transport service. These include communications providers that provide the ISO transport service over a TCP network.[1)]

In general, this subclause describes the characteristics that the ISO and ISO-overTCP communications providers have in common, with notes indicating where they differ. An ISO-over-TCP communications provider does not provide the connectionless mode.

In the context of the ISO transport protocols, a SDU is a TSDU, and a SEDU is an ETSDU, as defined in ISO/IEC 8072 {3}.

In an ISO environment, the protocol address is the transport address.

The type `ISO_XTI_Address` shall be used to represent an address for this protocol family. The type `ISO_XTI_Address_Pointer` is an access to this protocol-specific address type. The `"+"` operations shall convert a `ISO_XTI_Address_Pointer` to and from the `XTI_Address_Pointer` type for use with the base package operations defined for the `XTI_Address_Pointer` type. The return value of the `"+"` operations designates the same address object as the input parameter. The function `Is_ISO_XTI_Address` shall return `True` if the address object designated by the specified non-null `XTI_Address_Pointer` is a valid `ISO_XTI_Address` and `False` otherwise. The conversion operation to `ISO_XTI_Address_Pointer` shall succeed if and only if the corresponding `Is_ISO_XTI_Address` returns `True`. Otherwise, the results are undefined.

NOTE: The `Null_XTI_Address` constant corresponds to the Ada `null` literal for these operations.

The transport service definition, both in connection mode and connectionless mode, does not permit sending a TSDU of zero octets. Therefore, in connectionless mode, if the `Octets_To_Send` parameter is zero, the `Send_Data_Unit` procedure shall raise `POSIX_Error` with error code `Illegal_Data_Range`. In connection mode, if the `Octets_To_Send` parameter is set to zero, the `Send` procedure shall raise `POSIX_Error` with error code `Illegal_Data_Range` either if the `More_Data` flag is set or if the `More_Data` flag is not set and the preceding `Send` call completed a TSDU or ETSDU (*i.e.*, the sending of a zero byte TSDU or ETSDU was requested).

1) The mapping for ISO-over-TCP that is referred to in this subclause is defined by RFC 1006 {17}.

Options are formatted according to the Protocol_Option object as described in 17.3.2. The overloaded Get_Value function and Set_Option procedure are used to examine and set these attributes in the Protocol_Option object. A communications provider compliant with this specification may support none, all, or any subset of the options defined in this subclause and in D.2.2.2 and D.2.2.3. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode. An ISO-over-TCP provider supports a subset of the options defined in D.2.2.2.

QOS options are defined in the transport service definition in ISO/IEC 8072 {3}. The definitions are not repeated here.

QOS option values are expressed using the Rate object, which has the attributes Target Rate and Minimum Acceptable Rate. "Get_" and "Set_" operations are provided to manipulate these attributes. For certain options, these Rate attributes contain a Duration value representing a timer value (*e.g.*, Establishment Delay XTI option is expressed in milliseconds). For all other QOS options, Rate contains an implementation-defined integral value of type Error_Rate. Error_Rate represents an error ratio calculated from the formula $Error\_Rate = -\log_{10}(ratio)$, where *ratio* is dependent on the parameter, but is always composed of a number of failures divided by a total number of samples. The implementation-defined value may be, for example, the number of TSDUs transferred in error divided by the total number of TSDU transfers (Residual_Error_Rate).

For certain QOS options, an application can indicate that a QOS option value is unspecified by setting the rate attributes to the constant Unspecified_Rate.

An application can indicate whether the request is an absolute requirement or whether a degraded value is acceptable. An absolute requirement is specified via the attribute Minimum Acceptable Rate if that attribute is given a value different from Unspecified_Rate.

The following options are common to both connection-mode and connectionless-mode service:

Protection

This option defines the general level of protection. Its value is an object of type Protection_Level. The flags in the following list are used to specify the required level of protection:

— No_Protection: No protection feature is requested.

— Passive_Protection: Protection against passive monitoring is requested.

— Active_Protection: Protection against modification, replay, addition or deletion is requested.

— Absolute_Requirement: The requested protection level is an absolute requirement.

The operations "+", "-", ">", "<", ">=", "<=", and Empty_Set are available on the type Protection_Level via the derived type semantics of Ada, from the operations available for POSIX.Option_Set. The appropriate operations can be used to create and examine a set containing the required flags.

Both flags Passive_Protection and Active_Protection may be set simultaneously, but are exclusive with No_Protection. If the Active_-Protection or Passive_Protection flags are set, the application may indicate that the requested protections are an absolute requirement by also setting the Absolute_Requirement flag.

Priority

Five priority levels are defined by XTI: Default, Low, Medium, High, and Top. The number of priority levels is not defined by ISO/IEC 8072 {3}. The parameter only has meaning in the context of some management entity or object able to judge relative importance. The value specified for Priority is never an absolute requirement.

Residual Error Rate

This option defines the residual error rate. Its value is an error ratio expressed in a Rate object (see D.2.2.1).

D.2.2.2 Connection-Mode Service

D.2.2.2.1 Synopsis

```
Throughput :
   constant POSIX_XTI.Option_Name := implementation-defined;
Connection_Transit_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
Transfer_Fail_Probability :
   constant POSIX_XTI.Option_Name := implementation-defined;
Establishment_Fail_Probability :
   constant POSIX_XTI.Option_Name := implementation-defined;
Release_Fail_Probability :
   constant POSIX_XTI.Option_Name := implementation-defined;
Establishment_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
Release_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
Connection_Resilience :
   constant POSIX_XTI.Option_Name := implementation-defined;
Expedited_Data :
   constant POSIX_XTI.Option_Name := implementation-defined;
type Requested_Rate is private;
type Throughput_Rate is private;
type Transit_Delay_Rate is private;
function Get_Called_Rate (Item : Requested_Rate)
  return Rate;
procedure Set_Called_Rate
   (Item : in out Requested_Rate;
    To   : in     Rate);
function Get_Calling_Rate (Item : Requested_Rate)
  return Rate;
procedure Set_Calling_Rate
   (Item : in out Requested_Rate;
    To   : in     Rate);
function Get_Throughput_Maximum (Item : Throughput_Rate)
  return Requested_Rate;
procedure Set_Throughput_Maximum
   (Item : in out Throughput_Rate;
    To   : in     Requested_Rate);
```

```
function Get_Throughput_Average (Item : Throughput_Rate)
  return Requested_Rate;
procedure Set_Throughput_Average
   (Item : in out Throughput_Rate;
    To   : in     Requested_Rate);
function Get_Transit_Delay_Maximum (Item : Transit_Delay_Rate)
  return Requested_Rate;
procedure Set_Transit_Delay_Maximum
   (Item : in out Transit_Delay_Rate;
    To   : in     Requested_Rate);
function Get_Transit_Delay_Average (Item : Transit_Delay_Rate)
  return Requested_Rate;
procedure Set_Transit_Delay_Average
   (Item : in out Transit_Delay_Rate;
    To   : in     Requested_Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Throughput_Rate;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Throughput_Rate);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Transit_Delay_Rate;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Transit_Delay_Rate);
TPDU_Length_Maximum :
   constant POSIX_XTI.Option_Name := implementation-defined;
Acknowledge_Time :
   constant POSIX_XTI.Option_Name := implementation-defined;
Reassignment_Time :
   constant POSIX_XTI.Option_Name := implementation-defined;
Preferred_Class :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_1 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_2 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_3 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Alternative_Class_4 :
   constant POSIX_XTI.Option_Name := implementation-defined;
Extended_Format :
   constant POSIX_XTI.Option_Name := implementation-defined;
Flow_Control :
   constant POSIX_XTI.Option_Name := implementation-defined;
Connection_Checksum :
   constant POSIX_XTI.Option_Name := implementation-defined;
Network_Expedited_Data :
   constant POSIX_XTI.Option_Name := implementation-defined;
Network_Receipt_Confirmation :
   constant POSIX_XTI.Option_Name := implementation-defined;
type ISO_COTS_Option is (Enabled, Disabled, Unspecified);
type Transport_Class is private;
Class_0           : constant Transport_Class;
Class_1           : constant Transport_Class;
Class_2           : constant Transport_Class;
```

```
Class_3           : constant Transport_Class;
Class_4           : constant Transport_Class;
Class_Unspecified : constant Transport_Class;
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return ISO_COTS_Option;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     ISO_COTS_Option);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Transport_Class;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Transport_Class);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Duration;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     Duration);
```

D.2.2.2.2 Description

Figure D.6 shows the XTI state diagram for the connection-mode ISO transport service (see also Figure 17.1).

The protocol level of all subsequent options is `ISO_TP_Level`.

All options have end-to-end significance (see 17.3). They may be negotiated in the XTI states Idle and Incoming Connect, and are read-only in all other states except Uninitialized.

The QOS and expedited data options described in this subclause are in addition to the common options listed in D.2.2.1. The attribute elements of the objects in use for the option values are self-explanatory. Only the following details remain to be explained.

— If these options are returned with `Listen`, their values are related to the incoming connection and not to the communications endpoint where `Listen` was issued. To give an example, the value of Protection is the value sent by the calling application, and not the value currently effective for the endpoint (that could be retrieved by `Manage_Options` with the flag `Get_Current_Options` set). The option is not returned at all if the calling application did not specify it. An analogous procedure applies for the other options. See also 17.3.

— If, in a call to `Accept_Connection`, the called application tries to negotiate an option of higher quality than proposed, the option is rejected and the connection establishment fails (see 17.3.2.4).

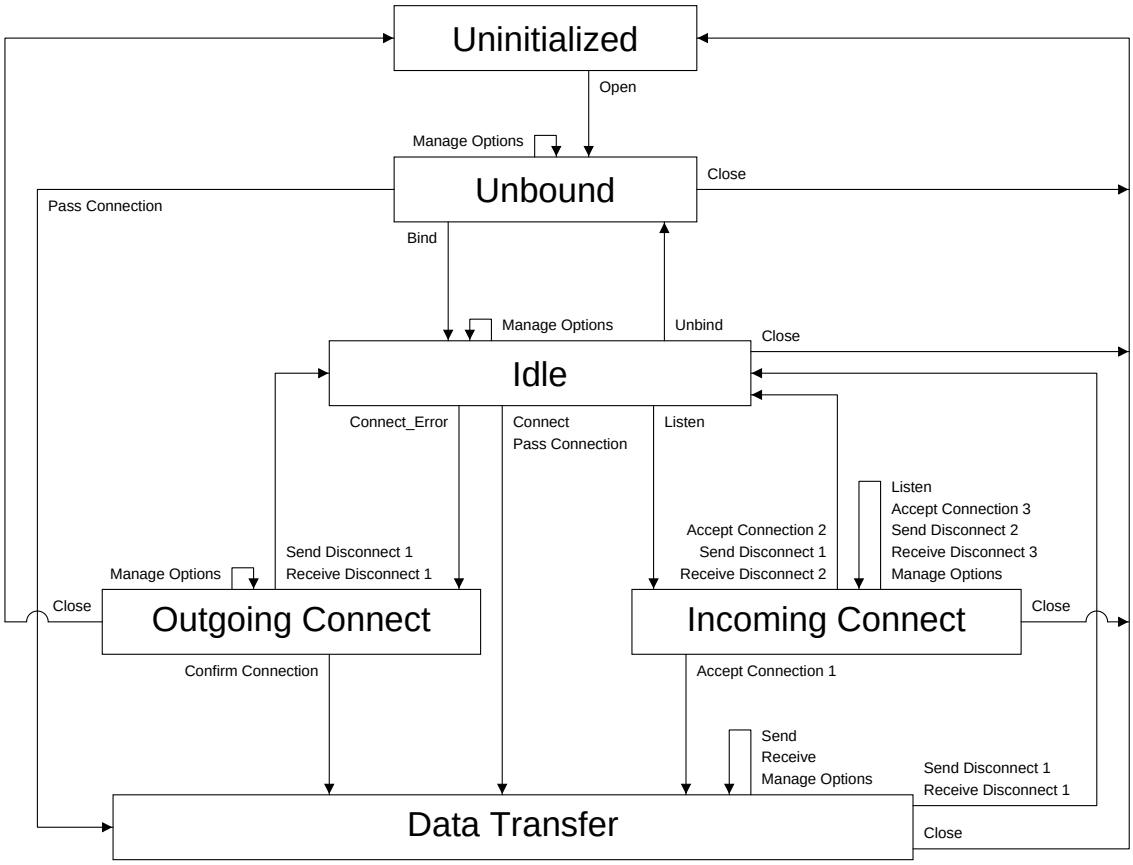

Figure D.6 – OSI Connection-Mode Transport XTI State Diagram

— Values of the QOS options Throughput XTI option, Transit Delay XTI option, Transfer Failure Probability XTI option, Establishment Failure Probability XTI option, Establishment Delay XTI option, Release Failure Probability XTI option, Release Delay XTI option, and Connection Resilience XTI option have a structured format. An application requesting one of these options might leave an attribute of the object unspecified by setting it to `Unspecified_Rate`. The communications provider is then free to select an appropriate value for this attribute. The communications provider may return `Unspecified_Rate` in an attribute of the object to the application to indicate that it has not yet decided on a definite value for this attribute.

The following QOS and epedited data options are formatted according to the `Protocol_Option` object as described in 17.3.2. The overloaded `Get_Value` function and `Set_Option` procedure are used to examine and set these attributes in the `Protocol_Option` object. The description for each option indicates the type of object manipulated by the `Get_Value` and `Set_Option` operations. "`Get_`" and "`Set_`" operations are provided to manipulate the individual attributes of each of these objects.

`Throughput`

The Throughput XTI option, which is in octets per second. The option value

is represented with a Throughput_Rate object. If the Throughput Average attribute is not defined (the Called Rate and Calling Rate attributes are both set to Unspecified_Rate), the communications provider considers that Throughput Average has the same value as Throughput Maximum.

Connection_Transit_Delay

The Connection Transit Delay XTI option. The option value is represented with a Transit_Delay_Rate object. ISO/IEC 8073 {4} does not differentiate between average and maximum transit delay. Communications providers that support this option adopt the values of the maximum delay as input for the CR TPDU.

If the Transit Delay Average attribute is not defined (the Called Rate and Calling Rate attributes are both set to Unspecified_Rate), the communications provider considers that Transit Delay Average has the same value as Transit Delay Maximum.

Transfer_Fail_Probability

The Transfer Failure Probability XTI option. Its value is an error ratio expressed in a Rate object (see D.2.2.1).

Establishment_Fail_Probability

The Establishment Failure Probability XTI option, which is the probability of failure of an attempt to establish a connection. Its value is an error ratio expressed in a Rate object (see D.2.2.1).

Release_Fail_Probability

The Release Failiure Probability XTI option, which is the probability of failure of an attempt to release a connection. Its value is an error ratio expressed in a Rate object (see D.2.2.1).

Establishment_Delay

The Establishment Delay XTI option. Its value is a Duration expressed in a Rate object (see D.2.2.1).

Release_Delay

The Release Delay XTI option. Its value is a Duration expressed in a Rate object (see D.2.2.1).

Connection_Resilience

The Connection Resilience XTI option, which is a ratio expressed with the Rate object. Its value is an error ratio expressed in a Rate object (see D.2.2.1).

Expedited_Data

The Expedited Data XTI option. It is recommended that applications avoid expedited data with an ISO-over-TCP communications provider, since the treatment of expedited data in RFC 1006 {17} does not meet the data reordering requirements specified in ISO/IEC 8072 {3} and may not be supported by the provider.

Unspecified is not a legal value for the Expedited Data XTI option.

The value of the Expedited Data XTI option is never an absolute requirement.

The management options defined in this subclause are in addition to the common options listed in D.2.2.1. These options are parameters of an ISO transport protocol according to ISO/IEC 8073 {4}. They are not included in the ISO transport service definition ISO/IEC 8072 {3}, but are additionally offered by XTI. Communications applications wishing to be truly ISO-compliant should thus not adhere to them. The TPDU Length Maximum XTI option is the only management option supported by an ISO-over-TCP communications provider.

The application should avoid specifying both QOS parameters and management options at the same time.

A request for any of these options shall be considered an absolute requirement.

If these options are returned with `Listen`, their values are related to the incoming connection and not to the communications endpoint where `Listen` was issued. That means `Manage_Options` with the flag `Get_Current_Options` set would usually yield a different result (see 17.3).

For management options that are subject to peer-to-peer negotiation the following holds: If, in a call to `Accept_Connection`, the called application tries to negotiate an option of higher quality than proposed, the option is rejected, and the connection establishment fails (see 17.3.2.4).

A connection-mode communications provider may allow the application to select more than one alternative class. The application may use options (*e.g.*, Alternative Class 1 XTI option and Alternative Class 2 XTI option) to denote the alternatives. A communications provider only supports an implementation dependent limit of alternatives and ignores the rest.

The value `Class_Unspecified` is legal for all these options. It may be set by the application to indicate that the communications provider is free to choose any appropriate value. If returned by the communications provider, it indicates that the communications provider has not yet decided on a specific value.

If a connection has been established, Preferred Class XTI option shall be set to the selected value, and Alternative Class 1 XTI option through Alternative Class 4 XTI option shall be set to `Class_Unspecified`, if these options are supported.

The management options are not independent of one another and not independent of the QOS options defined above. An application must take care not to request conflicting values. If conflicts are detected at negotiation time, the negotiation fails according to the rules for absolute requirements (see 17.3). Conflicts that cannot be detected at negotiation time can lead to unpredictable results in the course of communication. Usually, conflicts are detected at the time the connection is established.

Some relations that must be obeyed are as follows:

— If Expedited Data XTI option is set to `Enabled` and Preferred Class XTI option is set to `Class_2`, Flow_Control XTI option must also be set to `Enabled`.

— If Preferred Class XTI option is set to `Class_0`, Expedited Data XTI option must be set to `Disabled`.

— The value in Preferred Class XTI option must not be lower than the value in Alternative Class 1 XTI option, Alternative Class 2 XTI option, and so on.

— Depending on the chosen QOS options, further value conflicts might occur.

The following management options are formatted according to the `Protocol_Option` object as described in 17.3.2. The overloaded `Get_Value` function and `Set_Option` procedure are used to examine and set these attributes in the `Protocol_Option` object.

`TPDU_Length_Maximum`

The TPDU Length Maximum XTI option, which is the maximum length of a TPDU.

NOTE: Sensible use of the TPDU_Length_Maximum XTI option may require that the application programmer know about system internals. Careless setting of either a lower or a higher value than the implementation dependent default may degrade performance.

Legal values of the TPDU Length Maximum XTI option for ISO communications providers are `Unspecified` and multiples of 128 up to $128 * (232 - 1)$ or the largest multiple of 128 that will fit in an unsigned long (whichever is the smaller).

NOTE: Values other than powers of 2 between 27 and 213 are supported only by communications providers that conform to ISO/IEC 8073 {4}.

Legal values of the TPDU Length Maximum XTI option for an ISO-over-TCP provider are `Unspecified`, any power of 2 between 27 and 211, and 65531.

The action taken by a communications provider is implementation dependent if a value is specified that is not exactly as defined in ISO/IEC 8073 {4} or its addenda.

`Acknowledge_Time`

The Acknowledge Time XTI option.

`Reassignment_Time`

The Reassignment Time XTI option, which is measured in seconds.

`Preferred_Class`

The Preferred Class XTI option. Legal values are `Class_0`, `Class_1`, `Class_2`, `Class_3`, `Class_4`, and `Class_Unspecified`.

`Alternative_Class_1`

The Alternative Class1 XTI option, which is the first alternative class. Legal values are `Class_0`, `Class_1`, `Class_2`, `Class_3`, `Class_4`, and `Class_Unspecified`.

`Alternative_Class_2`

The Alternative Class 2 XTI option, which is the second alternative class. Legal values are `Class_0`, `Class_1`, `Class_2`, `Class_3`, `Class_4`, and `Class_Unspecified`.

`Alternative_Class_3`

The Alternative Class 3 XTI option. Legal values are `Class_0`, `Class_1`, `Class_2`, `Class_3`, `Class_4`, and `Class_Unspecified`.

`Alternative_Class_4`

The Alternative Class 4 XTI option. Legal values are `Class_0`, `Class_1`, `Class_2`, `Class_3`, `Class_4`, and `Class_Unspecified`.

`Extended_Format`

The Extended Format XTI option, which enables extended format. Legal values for this option are `Enabled`, `Disabled`, and `Unspecified`.

`Flow_Control`

The Flow Control XTI option, which enables use of flow control. Legal values for this option are `Enabled`, `Disabled`, and `Unspecified`.

`Connection_Checksum`

The Connection Checksum XTI option, which enables checksum computation. Legal values for this option are `Enabled`, `Disabled`, and `Unspecified`.

`Network_Expedited_Data`

The Network Expedited Data XTI option, which enables use of network expedited data. Legal values for this option are `Enabled`, `Disabled`, and `Unspecified`.

`Network_Receipt_Confirmation`

The Network Receipt Confirmation XTI option, which enables use of network receipt confirmation. Legal values for this option are `Enabled`, `Disabled`, and `Unspecified`.

D.2.2.3 Connectionless-Mode Service

D.2.2.3.1 Synopsis

```
Connectionless_Transit_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
Connectionless_Checksum :
   constant POSIX_XTI.Option_Name := implementation-defined;
```

D.2.2.3.2 Description

Figure D.5 shows the XTI state diagram for the connectionless-mode ISO transport service (see also Figure 17.1).

The protocol level of all subsequent options is `ISO_TP_Level` (as in D.2.2.2).

All options have end-to-end significance. (see 17.3). They may be negotiated in all XTI states but Uninitialized.

The following QOS option is in addition to the common options listed in D.2.2.1. They are all defined in ISO/IEC 8072 {3}. The definitions are not repeated here. None of these options are supported by an ISO-over-TCP communications provider since it does not support connectionless mode.

`Connectionless_Transit_Delay`

The Connectionless Transit Delay XTI option. The options Connectionless Transit Delay XTI option and Connection Transit Delay XTI option are different. Connectionless Transit Delay XTI option specifies the maximum transit delay expected during a datagram transmission. The option value is an error ratio expressed in a `Rate` object (see D.2.2.1), in contrast to the `Transit_Delay_Rate` object used by Connection Transit Delay XTI option. The range of legal option values for each attribute of object rate is the same as that of Connection Transit Delay XTI option.

A request for this option shall be an absolute requirement.

A detailed description of QOS options is found in ISO/IEC 8072 {3}. The attribute elements of the objects in use for the option values are self-explanatory. Only the following details remain to be explained.

— This option is negotiated only between the local application and the local communications provider.

— If this option is returned with `Receive_Data_Unit` its value is related to the received datagram and not to the communications endpoint where `Receive_Data_Unit` was issued. On the other hand, `Manage_Options` with the flag `Get_Current_Options` set returns the values that are currently effective for outgoing datagrams.

— The procedure `Retrieve_Data_Unit_Error` returns the option value of the data unit previously sent that produced the error.

The following management option is in addition to the common options listed in D.2.2.1. It is a parameter of an ISO transport protocol, according to ISO/IEC 8602 {9}. It is not included in the ISO transport service definition in ISO/IEC 8072 {3}, but is an additional option offered by XTI. Communications applications wishing to be truly ISO-compliant should thus not adhere to it.

The application should avoid specifying both QOS parameters and management options at the same time.

A request for this option shall be an absolute requirement.

`Checksum`

The Checksum XTI option enables checksum computation. Legal values for this option are `Enabled`, `Disabled`.

If the Checksum XTI option is returned with `Receive_Data_Unit`, its value shall indicate whether a checksum was present in the received datagram.

NOTE: The advisability of turning off the checksum check is controversial.

The following behavior for XTI functions is specified:ax

`Accept_Connection`

The length of the User Data attribute in parameter `Call` must be in the range 0 to 32. The application may send up to 32 octets of data when accepting the connection.

If `Listening_Endpoint` is not equal to `Responding_Endpoint`, then `Responding_Endpoint` should either be in the Unbound state, or be in the Idle state and be bound to the same address as `Listening_Endpoint` with the `Request_Queue_Length` parameter (of the `Bind` procedure) set to zero. If the protocol address bound to the new accepting endpoint (`Responding_Endpoint`) is not the same as that bound to the listening endpoint (`Listening_Endpoint`), then error `Surrogate_File_Descriptor_Mismatch` may result.

`Bind`

The `Request_Address` parameter represents the local TSAP.

`Close`

If there are no other descriptors in this process or any other process that reference this communication endpoint, `Close` will perform an abortive release on any connection associated with this endpoint.

`Connect`

The Address attribute of the `Send` parameter specifies the remote called TSAP. The returned address set in `Receive` shall have the same value.

The setting of the User Data attribute in `Send` is optional for ISO connections, but, with no data, the length of User Data must be set to 0. The Options and Address attributes of the `Receive` parameter must be set before the call.

`Confirm_Connection`

On return, the Address attribute in `Call` contains the remote calling TSAP. Since, at most, 32 octets of data shall be returned to the application, the size of the buffer associated with the User Data attribute of the `Call` parameter should be 32.

`Get_Address`

The Address attribute returned by `Get_Address` from a `Connection_Info` object shall be a protocol-specific `ISO_XTI_Address` object.

`Get_Info`

The information returned by `Get_Info` reflects the characteristics of the connection or, if no connection is established, the maximum characteristics a connection could take on using the underlying communications provider. In all possible states except Data Transfer, the function `Get_Info` returns in the parameter `Info` the same information as was returned by `Open`. In the Data Transfer state, however, the information returned may differ from that returned by `Open`, depending on

— The transport class negotiated during connection establishment (ISO communications provider only)

— The negotiation of expedited data transfer for this connection

In the Data Transfer state, the Max Size SEDU attribute in the `Info` parameter is set to `Invalid` if no expedited data transfer was negotiated, and to 16 otherwise. The remaining attributes are set according to the characteristics of the transport protocol class in use for this connection, as defined in Table D.7.

`Listen`

The Address attribute of `Call` contains the remote calling TSAP. Since, at most, 32 octets of data shall be returned with the connection indication, the length of the buffer associated with the User Data attribute of the `Call` parameter should be 32.

If the application has set an endpoint queue length greater than one (on the call to `Bind`), the application may queue up several connection indications before responding to any of them. The ISO communications provider may start a timer to be sure of obtaining a response to the connection request in a finite time.

NOTE: If the application queues the connection indications for too long before responding to them, the communications provider initiating the connection will disconnect it.

Table D.7 – `Communications_Provider_Info` Returned by `Get_Info` and `Open`, ISO

Attribute	Connection Class 0	Connection Class 1-4	Connection-less	ISO over TCP
Max Size Protocol Address	*x* (1)	*x* (1)	*x* (1)	*x* (1)
Max Size Protocol Options	*x* (1)	*x* (1)	*x* (1)	*x* (1)
Max Size SDU	*y* (2)	*y* (2)	0 .. 63488	*y* (2)
Max Size SEDU	*invalid* (3)	16 or *invalid* (4)	*invalid* (3)	16 or *invalid* (4)
Max Size Connect Data	*invalid* (3)	32	*invalid* (3)	32 or *invalid* (5)
Max Size Disconnect Data	*invalid* (3)	64	*invalid* (3)	64 or *invalid* (5)
Service Type	Connection_-Mode	Connection_-Mode	Connection-less_Mode	Connection_-Mode
CP Flags	Empty_Set	Empty_Set	Empty_Set	Empty_Set

NOTES:

(1) Either the corresponding function (`Protocol_Addresses_Are_Valid`, `Protocol_Options_Are_Valid`) returns `False` or the value of the attribute is set to an integral number *x* greater than zero.

(2) Either the corresponding function (`SDU_Is_Valid`) returns `True` or the value of the attribute is set to an integral number *y* greater than zero.

(3) The corresponding function (`SEDU_Is_Valid`, `Connection_Data_Is_Valid`, `Disconnect_Data_Is_Valid`) returns `False`.

(4) The corresponding function (`SEDU_Is_Valid`) returns `False` or the value of the attribute is set to the value shown, depending on the negotiation of expedited data transfer.

(5) It is implementation defined whether the corresponding function (`Connect_-Data_Is_Valid`, `Disconnect_Data_Is_Valid`) returns `False` or the value of the attribute is set to the value shown.

`Open`

The procedure `Open` is called as the first step in the initialization of a communications endpoint. This function returns various default characteristics associated with the different classes. According to ISO/IEC 8073 {4}, an OSI communications provider supports one or several out of five different transport protocols, Class 0 through Class 4. The default characteristics returned in the parameter `Info` are those of the highest-numbered protocol class the communications provider is able to support. If, for example, a communications provider supports Class 2 and Class 0, the characteristics returned are those of Class 2. If the communications provider is limited to Class 0, the characteristics returned are those of Class 0. Table D.7 gives the characteristics associated with the different classes.

`Receive`

If expedited data arrive after part of a TSDU has been retrieved, receipt of the remainder of the TSDU shall be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (`More_Data` not set) shall the remainder of the TSDU be available to the application.

`Retrieve_Disconnect_Info`

Since at most 64 octets of data shall be returned to the application, the size of the buffer associated with the `User_Data` parameter should be 64.

`Receive_Data_Unit`

The `Address` parameter specifies the remote TSAP. If the `More_Data` flag is set, an additional `Receive_Data_Unit` call is needed to retrieve the entire TSDU. Only normal data are returned via the `Receive_Data_Unit` call. This function is not supported by an ISO-over-TCP communications provider.

`Retrieve_Data_Unit_Error`

The `Address` parameter contains the remote TSAP.

`Send`

Zero-byte TSDUs are not supported. The `Expedited_Data` flag is not a legal flag unless the Expedited Data XTI option has been negotiated for this connection.

`Send_Disconnect_Request`

Since at most 64 octets of data may be sent with the disconnection request, the length of the User Data attribute in `Call` (or the value of the `Octets_To_Send` parameter) shall have a value less than or equal to 64.

`Send_Data_Unit`

The `Address` parameter specifies the remote TSAP. The ISO connectionless mode transport service does not support the sending of expedited data. This function is not supported by an ISO-over-TCP communications provider.

D.2.3 Package `POSIX_XTI_Internet`

This package provides the DNI/XTI interface mappings for Internet transport protocols. Unless otherwise specified, all the DNI/XTI calls in package `POSIX_XTI` can be used for this protocol. Only additional information relevant to this protocol is highlighted here.

The functionality described in this subclause is optional. If either the XTI Detailed Network Interface option or the Internet Protocol option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`. Otherwise, the behavior shall be as specified in this subclause.

```
with POSIX,
     POSIX_XTI;
package POSIX_XTI_Internet is
    -- D.2.3.1 Internet Transport Protocols
   TCP_Level : constant POSIX_XTI.Option_Level := implementation-defined;
   UDP_Level : constant POSIX_XTI.Option_Level := implementation-defined;
   IP_Level  : constant POSIX_XTI.Option_Level := implementation-defined;
   type XTI_Option is (Enabled, Disabled);
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
     return XTI_Option;
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
```

```
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     XTI_Option);
type Internet_XTI_Address is private;
type Internet_XTI_Address_Pointer is access all Internet_XTI_Address;
function "+" (Pointer : Internet_XTI_Address_Pointer)
  return POSIX_XTI.XTI_Address_Pointer;
function "+" (Pointer : POSIX_XTI.XTI_Address_Pointer)
  return Internet_XTI_Address_Pointer;
function Is_Internet_XTI_Address
   (Pointer : POSIX_XTI.XTI_Address_Pointer)
  return Boolean;
type Internet_Port is implementation-defined-integer;
Unspecified_Internet_Port : constant Internet_Port;
function Get_Internet_Port (Name : Internet_XTI_Address)
  return Internet_Port;
procedure Set_Internet_Port
   (Name : in out Internet_XTI_Address;
    Port : in     Internet_Port);
type Internet_Address is private;
Unspecified_Internet_Address : constant Internet_Address;
Loopback_Internet_Address    : constant Internet_Address;
Broadcast_Internet_Address   : constant Internet_Address;
function Get_Internet_Address (Name : Internet_XTI_Address)
  return Internet_Address;
procedure Set_Internet_Address
   (Name    : in out Internet_XTI_Address;
    Address : in     Internet_Address);
procedure Get_Address
   (Info_Item : in     POSIX_XTI.Connection_Info;
    Address   : in out Internet_XTI_Address);
-- D.2.3.2 Internet Address Support Functions
-- Internet Address Manipulation
function String_To_Internet_Address (Address : POSIX.POSIX_String)
  return Internet_Address;
function Is_Internet_Address (Address : POSIX.POSIX_String)
  return Boolean;
function Internet_Address_To_String (Address : Internet_Address)
  return POSIX.POSIX_String;
-- Network Database Functions
type Network_Info is private;
type Network_Number is range implementation-defined;
type Protocol_Family is range implementation-defined;
Unspecified_Network_Number : constant Network_Number;
type Database_Array is new POSIX.Octet_Array;
type Database_Array_Pointer is access all Database_Array;
function Get_Name (Info_Item : Network_Info)
  return POSIX.POSIX_String;
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Network_Alias (Info_Item : Network_Info);
function Get_Family (Info_Item : Network_Info)
  return Protocol_Family;
function Get_Network_Number (Info_Item : Network_Info)
  return Network_Number;
function Get_Network_Info_By_Address
   (Number  : Network_Number;
```

```
    Family  : Protocol_Family;
    Storage : Database_Array_Pointer)
  return Network_Info;
function Get_Network_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Network_Info;
procedure Open_Network_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Network_Database_Connection;
-- Network Protocol Database Functions
type Protocol_Info is private;
type Protocol_Number is range implementation-defined;
function Get_Name (Info_Item : Protocol_Info)
  return POSIX.POSIX_String;
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Protocol_Alias (Info_Item : Protocol_Info);
function Get_Protocol_Number (Info_Item : Protocol_Info)
  return Protocol_Number;
function Get_Protocol_Info_By_Number
   (Number  : Protocol_Number;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
function Get_Protocol_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
procedure Open_Protocol_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Protocol_Database_Connection;
-- D.2.3.3 Internet Transmission Control Protocol
TCP_Keep_Alive_Interval :
   constant POSIX_XTI.Option_Name := implementation-defined;
TCP_Segment_Size_Maximum :
   constant POSIX_XTI.Option_Name := implementation-defined;
TCP_No_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
type Keep_Alive_Info is private;
type Keep_Alive_Status is private;
subtype Keep_Alive_Time is POSIX.Minutes range 1 .. POSIX.Minutes'Last;
Keep_Alive_On  : constant Keep_Alive_Status;
Keep_Alive_Off : constant Keep_Alive_Status;
Send_Garbage   : constant Keep_Alive_Status;
function Get_Status (Info_Item : Keep_Alive_Info)
  return Keep_Alive_Status;
procedure Set_Status
   (Info_Item : in out Keep_Alive_Info;
    To        : in     Keep_Alive_Status);
procedure Set_Keep_Alive_Interval_Default
   (Info_Item : in out Keep_Alive_Info);
procedure Set_Keep_Alive_Timeout
   (Info_Item : in out Keep_Alive_Info;
    To        : in     Keep_Alive_Time);
function Get_Keep_Alive_Timeout
   (Info_Item : Keep_Alive_Info)
```

```
      return Keep_Alive_Time;
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
      return Keep_Alive_Info;
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
       Name        : in     POSIX_XTI.Option_Name;
       Value       : in     Keep_Alive_Info);
   -- D.2.3.4 Internet User Datagram Protocol
   UDP_Checksum : constant POSIX_XTI.Option_Name := implementation-defined;
   -- D.2.3.5 Internet Protocol
   IP_Options          : constant POSIX_XTI.Option_Name := implementation-defined;
   IP_Type_Of_Service  : constant POSIX_XTI.Option_Name := implementation-defined;
   IP_Time_To_Live     : constant POSIX_XTI.Option_Name := implementation-defined;
   IP_Reuse_Address    : constant POSIX_XTI.Option_Name := implementation-defined;
   IP_Do_Not_Route     : constant POSIX_XTI.Option_Name := implementation-defined;
   IP_Permit_Broadcast : constant POSIX_XTI.Option_Name := implementation-defined;
   type IP_Option_List is new POSIX.Octet_Array;
   procedure Get_Value
      (Option_Item : POSIX_XTI.Protocol_Option;
       IP_Option   : out IP_Option_List;
       Count       : out Natural);
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
       Name        : in     POSIX_XTI.Option_Name;
       To          : in     IP_Option_List);
   type IP_Service_Type is private;
   Normal           : constant IP_Service_Type;
   Low_Delay        : constant IP_Service_Type;
   High_Throughput  : constant IP_Service_Type;
   High_Reliability : constant IP_Service_Type;
   Low_Cost         : constant IP_Service_Type;
   type IP_Precedence_Level is private;
   Routine              : constant IP_Precedence_Level;
   Priority             : constant IP_Precedence_Level;
   Immediate            : constant IP_Precedence_Level;
   Flash                : constant IP_Precedence_Level;
   Flash_Override       : constant IP_Precedence_Level;
   Critic_ECP           : constant IP_Precedence_Level;
   Internetwork_Control : constant IP_Precedence_Level;
   Network_Control      : constant IP_Precedence_Level;
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
      return IP_Service_Type;
   function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
      return IP_Precedence_Level;
   procedure Set_Option
      (Option_Item : in out POSIX_XTI.Protocol_Option;
       Level       : in     POSIX_XTI.Option_Level;
       Name        : in     POSIX_XTI.Option_Name;
       Service     : in     IP_Service_Type;
       Precedence  : in     IP_Precedence_Level);

private
   implementation-defined
end POSIX_XTI_Internet;
```

D.2.3.1 Internet Transport Protocols

D.2.3.1.1 Synopsis

```
TCP_Level : constant POSIX_XTI.Option_Level := implementation-defined;
UDP_Level : constant POSIX_XTI.Option_Level := implementation-defined;
IP_Level  : constant POSIX_XTI.Option_Level := implementation-defined;
type XTI_Option is (Enabled, Disabled);
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return XTI_Option;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     XTI_Option);
type Internet_XTI_Address is private;
type Internet_XTI_Address_Pointer is access all Internet_XTI_Address;
function "+" (Pointer : Internet_XTI_Address_Pointer)
  return POSIX_XTI.XTI_Address_Pointer;
function "+" (Pointer : POSIX_XTI.XTI_Address_Pointer)
  return Internet_XTI_Address_Pointer;
function Is_Internet_XTI_Address
   (Pointer : POSIX_XTI.XTI_Address_Pointer)
  return Boolean;
type Internet_Port is implementation-defined-integer;
Unspecified_Internet_Port : constant Internet_Port;
function Get_Internet_Port (Name : Internet_XTI_Address)
  return Internet_Port;
procedure Set_Internet_Port
   (Name : in out Internet_XTI_Address;
    Port : in     Internet_Port);
type Internet_Address is private;
Unspecified_Internet_Address : constant Internet_Address;
Loopback_Internet_Address    : constant Internet_Address;
Broadcast_Internet_Address   : constant Internet_Address;
function Get_Internet_Address (Name : Internet_XTI_Address)
  return Internet_Address;
procedure Set_Internet_Address
   (Name    : in out Internet_XTI_Address;
    Address : in     Internet_Address);
procedure Get_Address
   (Info_Item : in     POSIX_XTI.Connection_Info;
    Address   : in out Internet_XTI_Address);
```

D.2.3.1.2 Description

This subclause describes the protocol-specific information that is relevant for communications providers that are TCP or UDP communications providers.

The format of Internet addresses is defined in RFC 791 {13}. The representations of Internet addresses, network identifiers, and host identifiers is implementation defined.

The type `Internet_XTI_Address` shall be used to represent an address for this protocol family. The type `Internet_XTI_Address_Pointer` is an access to this protocol-specific address type. The `"+"` operations shall convert a `Internet_XTI_Address_Pointer` to and from the `XTI_Address_Pointer` type for use with the

base package operations defined for the XTI_Address_Pointer type. The return value of the "+" operations designates the same address object as the input parameter. The function Is_Internet_XTI_Address shall return True if the address object designated by the specified non-null XTI_Address_Pointer is a valid Internet_XTI_Address and False otherwise. The conversion operation to Internet_XTI_Address_Pointer shall succeed if and only if the corresponding Is_Internet_XTI_Address returns True. Otherwise, the results are undefined.

NOTE: The Null_XTI_Address constant corresponds to the Ada null literal for these operations.

A value of type Internet_Port forms a component of the XTI address. Given an Internet XTI address, Get_Internet_Port returns the corresponding Internet Port. Set_Internet_Port sets the Internet Port value in an Internet XTI address. This value is set to Unspecified_Internet_Port when an XTI address is created and must be set to a valid port number by the application.

A value of type Internet_Address forms a component of the XTI address. Given an Internet XTI address, Get_Internet_Address returns the corresponding Internet address. Set_Internet_Address sets the Internet address value in an Internet XTI address. This value is set to Unspecified_Internet_Address when an XTI address is created and needs to be set to a valid address by the application.

Options are formatted according to the Protocol_Option object as described in 17.3.2. The overloaded Get_Value function and Set_Option procedure are used to examine and set these attributes in the Protocol_Option object. A communications provider compliant to this specification shall support none, all, or any subset of the options defined in D.2.3.3, D.2.3.4 and D.2.3.5. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode.

The following behavior for XTI functions is specified:

Accept_Connection

Issuing Accept_Connection assigns an already established connection to Responding_Endpoint.

Since application data cannot be exchanged during the connection establishment phase, the application shall set the length of the User Data attribute in Call to 0 before the call, or the overloaded Accept_Connection without the Call parameter should be used. If file descriptor Responding_Endpoint is unbound, then Accept_Connection shall bind it to the same address as Listening_Endpoint. If file descriptor Responding_Endpoint is bound to an address other than that to which Listening_Endpoint is bound, then Accept_Connection shall dissolve its existing binding and shall bind it to the same address as Listening_Endpoint. If file descriptor Responding_Endpoint is bound to the same address as Listening_Endpoint, then Accept_Connection shall leave its binding unchanged.

NOTE: TCP requires the responding address to be the same as the address given in the connect request. This address must be supplied in `Listening_Endpoint` *by the application. Many implementations do not allow two endpoints to be bound to the same address, so that the application cannot bind* `Responding_Endpoint` *to the same address as* `Listening_Endpoint`*. The specified behavior enables an application to bind* `Responding_Endpoint` *to any address, then to call* `Accept_Connection`*, and then to use* `Responding_Endpoint` *to send and receive data over the connection. Thus the specified behavior allows an application to use TCP with such an implementation. It follows that, if the* `Bind` *procedure can only bind one communications endpoint to any particular protocol address, then an application can employ either of the following alternatives for accepting a connection on a different endpoint (*`Responding_Endpoint` *not equal to* `Listening_Endpoint`*):*

(1) It can call `Accept_Connection` *while* `Responding_Endpoint` *is in the Unbound state.*

(2) It can bind to any unused local address and then call `Accept_Connection` *in the Idle state. In this case,* `Accept_Connection` *will change the* `Responding_Endpoint` *address to be the same as that of* `Listening_Endpoint`*.*

For portability, the first alternative should be used.

If options with end-to-end significance (*i.e.*the IP Options XTI option and the IP Type Of Service XTI option) are to be sent with the connection confirmation, the values of these options must be set with `Manage_Options` before the `Connect_Request_Received` event occurs. When the application detects a `Connect_Request_Received`, TCP has already established the connection. Options with end-to-end significance passed with `Accept_Connection` become effective at once, but since the connection is already established, they are transmitted with subsequent IP datagrams sent out in the Data Transfer state.

`Bind`

The `Request_Address` parameter represents the address of the local endpoint, *i.e.*, an address that specifically includes a port identifier.

In connection mode (*i.e.*, TCP), whether the `Bind` procedure can bind more than one communications endpoint to any particular protocol address is implementation defined. If the `Bind` procedure can only bind one communications endpoint to any particular protocol address, and if that endpoint was bound in passive mode (*i.e.*, the endpoint queue length is greater than zero), then other endpoints shall be bound to the protocol of the passive endpoint address via the `Accept_Connection` function only. In other words, if `Listening_Endpoint` refers to the passive endpoint and `Responding_Endpoint` refers to the new endpoint on which the connection is to be accepted, `Responding_Endpoint` shall be bound to the same protocol address as `Listening_Endpoint` after the successful completion of the `Accept_Connection` function.

`Close`

If there are no other descriptors, in this process or any other process, that reference this communication endpoint the `Close` call will perform an orderly connection termination according to the rules of a TCP Close call as specified in RFC 793 {14} and RFC 1122 {18}.

If the Linger On Close If Data Present XTI option is supported and is used to enable the Linger, the linger time will affect the time that the implementation lingers in the execution of `Close`. A linger time of zero specified with

the Linger On Close If Data Present XTI option may cause an abortive release of a TCP connection resulting in lost data.

`Connect`

The Address attribute of the `Connection_Info` object in `Send` specifies the address of the remote endpoint. The returned address set in the `Address` attribute of `Receive` shall have the same value.

Since application data cannot be exchanged during the connection establishment phase, the application shall set the length of the User Data attribute in `Send` to zero before the call. The peer TCP – and not the peer application – confirms the connection.

`Get_Address`

The Address attribute returned by `Get_Address` from a `Connection_Info` object shall be a protocol-specific `Internet_XTI_Address` object.

`Listen`

Upon successful return, `Listen` indicates an existing connection and not a connection indication.

Since application data cannot be exchanged during the connection establishment phase, the length of the User Data attribute of `Call` must be set to zero before the call to `Listen`. The Address attribute of `Call` contains the address of the remote calling endpoint.

`Look`

As soon as a segment with the TCP urgent pointer set enters the TCP receive buffer, the event `Expedited_Data_Received` is indicated. `Expedited_Data_Received` remains set until all data up to the octet pointed to by the TCP urgent pointer have been received. If the urgent pointer is updated, and the application has not yet received the octet previously pointed to by the urgent pointer, the update is invisible to the application.

`Receive`

The `More_Data` flag shall be ignored if normal data are delivered. If an octet in the data stream is pointed to by the TCP urgent pointer, as many octets as possible preceding this marked octet and the marked octet itself shall be denoted as urgent data and shall be received with the `Expedited_Data` flag set. If the buffer supplied by the application is too small to hold all urgent data, the `More_Data` flag shall be set, indicating that urgent data still remain to be read. The number of octets received with the `Expedited_Data` flag set is not necessarily equal to the number of octets sent by the peer application with the `Expedited_Data` flag set.

`Open`

`Open` is called as the first step in the initialization of a communications endpoint. This function returns various default characteristics of the underlying communications protocol by setting attributes in the `Communications_Provider_Info` object.

The values returned by calls to `Open` and `Get_Info` with the indicated communications providers are as defined in Table D.8.

Table D.8 – Communications_Provider_Info Returned by Get_Info and Open, Internet

Attribute	TCP/IP	UDP/IP
Max Size Protocol Address	*x* (1)	*x* (1)
Max Size Protocol Options	*x* (1)	*x* (1)
Max Size SDU	zero	*x* (1)
Max Size SEDU	*infinite* (2)	*invalid* (3)
Max Size Connect Data	*invalid* (3)	*invalid* (3)
Max Size Disconnect Data	*invalid* (3)	*invalid* (3)
Service Type	Connection_Mode_-With_Orderly_Release	Connectionless_Mode
CP Flags	Zero_Length_Service_-Data_Unit_Supported	Zero_Length_Service_-Data_Unit_Supported

NOTES:

(1) Either the corresponding function (Protocol_Addresses_Are_Valid, Protocol_Options_Are_Valid, SDU_Is_Valid) returns False or the value of the attribute is set to an integral number *x* greater than zero.

(2) The corresponding function (SDU_Is_Infinite) returns True.

(3) The corresponding function (SEDU_Is_Valid, Connect_Data_Is_Valid, Disconnect_Data_Is_Valid) returns False.

Confirm_Connection

Since application data cannot be exchanged during the connection establishment phase, the length of the User Data attribute of Call must be set to zero before the call to Confirm_Connection, or the overloaded Confirm_Connection without the Call parameter should be used.

On return, the Address attribute of Call contains the address of the remote calling endpoint.

Retrieve_Disconnect_Info

Since data may not be sent with a disconnection request, the User_Data parameter shall not be meaningful.

Send

The More_Data and Push flags shall be ignored. If Send is called with more than one octet specified and with the Expedited_Data flag set, then the last octet of the buffer shall be the octet that will be pointed to by the TCP urgent pointer of the remote endpoint. If the Expedited_Data flag is set, at least one octet shall be sent.

Data for a Send call with the Expedited_Data flag set shall not pass data sent previously. In other words, the Expedited_Data flag shall not affect the order in which data are transmitted to the peer application.

Send_Disconnect_Request

Since data may not be sent with a disconnection request, the length of the User Data attribute in Call (or the Octets_To_Send parameter) must be set to zero.

NOTE: For Send_Data_Unit the maximum size of a connectionless-mode SDU may vary among implementations.

D.2.3.2 Internet Address Support Functions

D.2.3.2.1 Synopsis

```
-- Internet Address Manipulation
function String_To_Internet_Address (Address : POSIX.POSIX_String)
  return Internet_Address;
function Is_Internet_Address (Address : POSIX.POSIX_String)
  return Boolean;
function Internet_Address_To_String (Address : Internet_Address)
  return POSIX.POSIX_String;
-- Network Database Functions
type Network_Info is private;
type Network_Number is range implementation-defined;
type Protocol_Family is range implementation-defined;
Unspecified_Network_Number : constant Network_Number;
type Database_Array is new POSIX.Octet_Array;
type Database_Array_Pointer is access all Database_Array;
function Get_Name (Info_Item : Network_Info)
  return POSIX.POSIX_String;
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Network_Alias (Info_Item : Network_Info);
function Get_Family (Info_Item : Network_Info)
  return Protocol_Family;
function Get_Network_Number (Info_Item : Network_Info)
  return Network_Number;
function Get_Network_Info_By_Address
   (Number  : Network_Number;
    Family  : Protocol_Family;
    Storage : Database_Array_Pointer)
  return Network_Info;
function Get_Network_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Network_Info;
procedure Open_Network_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Network_Database_Connection;
-- Network Protocol Database Functions
type Protocol_Info is private;
type Protocol_Number is range implementation-defined;
function Get_Name (Info_Item : Protocol_Info)
  return POSIX.POSIX_String;
generic
   with procedure Action
      (Alias_Name : in     POSIX.POSIX_String;
       Quit       : in out Boolean);
procedure For_Every_Protocol_Alias (Info_Item : Protocol_Info);
function Get_Protocol_Number (Info_Item : Protocol_Info)
  return Protocol_Number;
function Get_Protocol_Info_By_Number
   (Number  : Protocol_Number;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
function Get_Protocol_Info_By_Name
   (Name    : POSIX.POSIX_String;
    Storage : Database_Array_Pointer)
  return Protocol_Info;
procedure Open_Protocol_Database_Connection
   (Stay_Open : in Boolean);
procedure Close_Protocol_Database_Connection;
```

D.2.3.2.2 Description

The functionality described in this subclause is optional. If the Network Management option is not supported, the implementation may cause all calls to the explicitly declared operations defined in this subclause to raise `POSIX_Error`*. Otherwise, the behavior shall be as specified in this subclause.*

The functions and procedures in this subclause provide optional Internet support functions. See D.1.3.2 for a description of these facilities.

D.2.3.3 Internet Transmission Control Protocol

D.2.3.3.1 Synopsis

```
TCP_Keep_Alive_Interval :
   constant POSIX_XTI.Option_Name := implementation-defined;
TCP_Segment_Size_Maximum :
   constant POSIX_XTI.Option_Name := implementation-defined;
TCP_No_Delay :
   constant POSIX_XTI.Option_Name := implementation-defined;
type Keep_Alive_Info is private;
type Keep_Alive_Status is private;
subtype Keep_Alive_Time is POSIX.Minutes range 1 .. POSIX.Minutes'Last;
Keep_Alive_On  : constant Keep_Alive_Status;
Keep_Alive_Off : constant Keep_Alive_Status;
Send_Garbage   : constant Keep_Alive_Status;
function Get_Status (Info_Item : Keep_Alive_Info)
  return Keep_Alive_Status;
procedure Set_Status
   (Info_Item : in out Keep_Alive_Info;
    To        : in     Keep_Alive_Status);
procedure Set_Keep_Alive_Interval_Default
   (Info_Item : in out Keep_Alive_Info);
procedure Set_Keep_Alive_Timeout
   (Info_Item : in out Keep_Alive_Info;
    To        : in     Keep_Alive_Time);
function Get_Keep_Alive_Timeout
   (Info_Item : Keep_Alive_Info)
  return Keep_Alive_Time;
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return Keep_Alive_Info;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    Value       : in     Keep_Alive_Info);
```

D.2.3.3.2 Description

The TCP service supports all the states in Figure D.4. (See also Figure 17.1.)

The notion of SDU is not supported by a TCP communications provide. Therefore, the `More_Data` flag shall be ignored when TCP is used.

TCP does not have a notion of expedited data in a sense comparable to ISO expedited data. TCP defines an urgent mechanism by which in-line data are marked for urgent delivery.

The options listed below are defined for the TCP protocol level. These options do not have end-to-end significance. They can be negotiated in all XTI states except Unbound and Uninitialized. They shall be read-only in the Unbound state. See 17.3 for the difference between options that have end-to-end significance and those that do not.

Keep Alive Interval

If the Keep Alive Interval XTI option is set, a keep-alive timer shall be activated to monitor idle connections that might no longer exist. If a connection has been idle since the last keep-alive timeout, a keep-alive packet shall be sent to check whether the connection is still alive or broken.

NOTE: Keep-alive packets are not an explicit feature of TCP, and this practice is not universally accepted. According to RFC 1122 {18}, "a keep-alive mechanism should only be invoked in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a client crashes or aborts a connection during a network failure".

Valid values for this option are:

`Keep_Alive_Off`

Switch the keep-alive timer off.

`Keep_Alive_On`

Activate the keep-alive timer.

`Send_Garbage`

Activate the keep-alive timer and send a fill octet.

An implementation need not support `Send_Garbage` (see RFC 1122 {18}); therefore, a `Set_Option` to `Send_Garbage` may be rejected. If `Send_Garbage` is set and the implementation supports `Send_Garbage`, then the keep-alive packet shall contain one garbage octet.

NOTE: The permission above is for compatibility with erroneous TCP implementations.

If `Send_Garbage` is not set, the implementation shall not include garbage data in keep-alive packets.

NOTE: In most cases, the application should not set `Send_Garbage`.

The conditions under which keep-alive packets are to be sent are described in RFC 1122 {18}. The `Set_Option` procedure determines the frequency, in minutes. The application can request the default value with the `Set_Keep_Alive_Interval_Default` function. The default is implementation dependent, but at least 120 minutes (see RFC 1122 {18}). `Get_Value` returns the current timeout value.

The timeout value shall not be an absolute requirement. The implementation may impose upper and lower limits on this value. Requests that fall short of the lower limit may be negotiated to the lower limit. The default status for the Keep Alive Interval XTI option shall be `Keep_Alive_Off`.

A request to activate the Keep Alive Interval XTI option shall be an absolute requirement.

Segment Size Maximum

The Segment Size Maximum XTI option shall be read-only. `Get_Value` returns the maximum TCP segment size.

No Delay

Under most circumstances, TCP sends data as soon as they are presented. When outstanding data have not yet been acknowledged, small amounts of output may be gathered to be sent in a single packet once an acknowledgment is received. For some clients, such as window systems (see Scheifler-{B20}) that send a stream of mouse events that receive no replies, this packetization may cause significant delays. The No Delay XTI option is used to defeat this algorithm. Legal option values are `Enabled` (do not delay) and `Disabled` (delay).

`Set_Option` sets the No Delay XTI option. `Get_Value` returns the current status. The default status shall be `Disabled`. A request for No Delay XTI option shall be an absolute requirement.

D.2.3.4 Internet User Datagram Protocol

D.2.3.4.1 Synopsis

```
UDP_Checksum : constant POSIX_XTI.Option_Name := implementation-defined;
```

D.2.3.4.2 Description

The UDP service supports all the states in Figure D.5. (See also Figure 17.1.)

UDP has no urgent mechanism. See RFC 793 {14} for more detailed information. See also the discussion of `Receive` in D.2.3.1. Applications can use the event management functions defined in 19.1 (or another suitable interface offered by the implementation) to determine when urgent data have arrived.

The option listed below is defined for the UDP protocol level:

`UDP_Checksum`

The UDP Checksum XTI option, which shall control the disabling or enabling of the UDP checksum computation. Its legal values are `Enabled` and `Disabled`.

This option has end-to-end significance. It can be negotiated in all XTI states except Unbound and Uninitialized. It shall be read-only in the Unbound state. See 17.3 for the difference between options that have end-to-end significance and those that do not. A request for the UDP Checksum XTI option shall be an absolute requirement.

If the UDP Checksum XTI option is returned with `Receive_Data_Unit`, its value shall indicate whether a checksum was present in the received datagram. `Set_Option` sets the UDP Checksum XTI option status. `Get_Value` returns the current status. The default for UDP Checksum XTI option shall be `Enabled`.

NOTE: Numerous cases of undetected errors have been reported when applications chose to turn off checksums for efficiency. Applications should normally set UDP Checksum XTI option to `Enabled`.

D.2.3.5 Internet Protocol

D.2.3.5.1 Synopsis

```
IP_Options          : constant POSIX_XTI.Option_Name := implementation-defined;
IP_Type_Of_Service  : constant POSIX_XTI.Option_Name := implementation-defined;
IP_Time_To_Live     : constant POSIX_XTI.Option_Name := implementation-defined;
IP_Reuse_Address    : constant POSIX_XTI.Option_Name := implementation-defined;
IP_Do_Not_Route     : constant POSIX_XTI.Option_Name := implementation-defined;
IP_Permit_Broadcast : constant POSIX_XTI.Option_Name := implementation-defined;
type IP_Option_List is new POSIX.Octet_Array;
procedure Get_Value
   (Option_Item : POSIX_XTI.Protocol_Option;
    IP_Option   : out IP_Option_List;
    Count       : out Natural);
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    To          : in     IP_Option_List);
type IP_Service_Type is private;
Normal           : constant IP_Service_Type;
Low_Delay        : constant IP_Service_Type;
High_Throughput  : constant IP_Service_Type;
High_Reliability : constant IP_Service_Type;
Low_Cost         : constant IP_Service_Type;
type IP_Precedence_Level is private;
Routine              : constant IP_Precedence_Level;
Priority             : constant IP_Precedence_Level;
Immediate            : constant IP_Precedence_Level;
Flash                : constant IP_Precedence_Level;
Flash_Override       : constant IP_Precedence_Level;
Critic_ECP           : constant IP_Precedence_Level;
Internetwork_Control : constant IP_Precedence_Level;
Network_Control      : constant IP_Precedence_Level;
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return IP_Service_Type;
function Get_Value (Option_Item : POSIX_XTI.Protocol_Option)
  return IP_Precedence_Level;
procedure Set_Option
   (Option_Item : in out POSIX_XTI.Protocol_Option;
    Level       : in     POSIX_XTI.Option_Level;
    Name        : in     POSIX_XTI.Option_Name;
    Service     : in     IP_Service_Type;
    Precedence  : in     IP_Precedence_Level);
```

D.2.3.5.2 Description

The options listed below are defined for the Internet protocol level. A request for any of these options shall be an absolute requirement.

The IP Options XTI option and Type of Service XTI option are both options that have end-to-end significance. No other options have end-to-end significance. See 17.3 for the difference between options that have end-to-end significance and options that do not.

Reuse Address XTI option can be negotiated in all XTI states except Uninitialized. All other options can be negotiated in all other XTI states except Unbound and Uninitialized; they are read-only in the Unbound state.

`IP_Options`

The IP Options XTI option, which is used to set the OPTIONS field of each outgoing IP datagram, and to retrieve the OPTIONS field of each incoming IP datagram. Its value has type `IP_Option_List`, which is a string of octets composed of a number of IP options, whose format matches those defined in the IP specification with one exception: the list of addresses for the source routing options shall include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address shall be extracted from the option list and the size adjusted accordingly before use. The option shall be disabled if it is specified with no value. The implementation may silently truncate the `IP_Option_List` to an implementation-defined limit.

The functions `Connect` (when `POSIX_IO.Non_Blocking` is not set), `Listen`, `Confirm_Connection`, `Receive_Data_Unit`, and `Receive_And_Scatter_Data_Unit` shall return the OPTIONS field, if any, of a received IP datagram. The procedure `Retrieve_Data_Unit_Error` shall return the OPTIONS field of the data unit previously sent that produced the error. The procedure `Manage_Options` with `Get_Current_Options` set shall retrieve the currently effective `IP_Options` value that is sent with outgoing datagrams.

NOTE: Common applications never need the IP Options XTI option. It is mainly used for network debugging and control purposes.

`Type_Of_Service`

The Type of Service XTI option, which is used to set the type-of-service field of an outgoing IP datagram and to retrieve the type-of-service field of an incoming IP datagram. This option is a combination of a precedence value and a type-of-service value as defined in RFC 791 {13}.

The type of service value shall specify the type of service that the network should provide for the IP datagram. Legal type-of-service values are as follows: `Normal`, `Low_Delay`, `High_Throughput`, `High_Reliability`, and `Low_Cost`. The precedence values shall specify datagram precedence, allowing senders to indicate the importance of each datagram as follows: `Routine`, `Priority`, `Immediate`, `Flash`, `Flash_Override`, `Critic_ECP`, `Internetwork_Control`, and `Network_Control`.

NOTE: They are intended for Department of Defense applications.

Applications using `Type_Of_Service`, but not the precedence level, should use the value `Routine` for precedence.

The functions `Connect`, `Listen`, `Confirm_Connection`, and `Receive_Data_Unit` shall return the type-of-service field of any received IP datagram associated with the call. The procedure `Retrieve_Data_Unit_Error` shall return the type-of-service field of the data unit previously sent that produced the error. The procedure `Manage_Options` with `Get_Current_Options` set shall retrieve the currently effective `Type_Of_Service` value that is sent with outgoing datagrams.

Provision of the requested type-of-service is not guaranteed.

NOTE: The requested type-of-service is a hint to the routing algorithm that helps it choose among various paths to a destination. Most hosts and gateways in the Internet currently ignore the type-of-service field.

`Reuse_Address`

The Reuse Address XTI option. Many TCP implementations do not allow the application to bind more than one communications endpoint to addresses with identical port numbers. If the Reuse Address XTI option is set to `Enabled` this restriction shall be relaxed to allowed the application to bind a communications endpoint to an address that has a port number and an underspecified Internet address (wild card address) and to bind further endpoints to addresses that have the same port number and (mutually exclusive) fully specified Internet addresses.

NOTE: A race condition can occur when two processes are both trying to connect from a specific local port to different remote ports. Only one process at a time can have a bound, but unconnected communications endpoint. In particular, if two processes both try to bind to the local port and then create a connection, the `Bind` *call of one process will succeed but the* `Bind` *call of the other process will fail. Therefore, a* `Bind` *call that returns an error should be retried.*

`Permit_Broadcast`

The Permit Broadcast XTI option, which requests permission to send broadcast datagrams (see RFC 791 {13}).

NOTE: The Permit Broadcast XTI option was defined to make sure that broadcasts are not generated by mistake.

`IP_Do_Not_Route`

The IP Do Not Route XTI option, which indicates that the destination address shall refer to a destination on a directly attached network interface and that interface shall be used to deliver all outgoing messages.

NOTE: The main use of the IP Do Not Route XTI option is for testing and development.

`Initial_Time_To_Live`

The Initial Time To Live XTI option, which is used to set the time-to-live field in an outgoing IP datagram. It specifies how long, in seconds, the datagram is allowed to remain in the Internet.[2)] The time-to-live field of an incoming datagram is not returned by any function (since it is not an option with end-to-end significance).

c

2) This is a simplified description. Refer to RFC 1122 {18} for precise details.

Alphabetic Topical Index

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