

INTERNATIONAL
STANDARD

ISO/IEC
14519

ANSI/IEEE
Std 1003.5b

First edition
1999-04-01

**Information technology — POSIX[®] Ada
Language Interfaces — Binding for System
Application Program Interface (API) —
Realtime Extensions**

*Technologies de l'information — Interfaces de langage POSIX[®] Ada —
Boucle pour interface de programme d'application système (API) —
Extensions temps réel*



Reference number
ISO/IEC 14519:1999(E)
ANSI/IEEE
Std 1003.5b, 1996 edition

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: 1987 (Ada) and ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990) (POSIX), as amended by IEEE Std 1003.1b-1993, IEEE Std 1003.1c-1995, and IEEE Std 1003.1i-1995. Terminology and general requirements, process primitives, the process environment, files and directories, input and output primitives, device- and class-specific functions, language-specific services for Ada, system databases, synchronization, memory management, execution scheduling, clocks and timers, and message passing are covered. It also specifies behavior to support the binding that must be provided by the Ada compilation system.

Keywords: Ada, API, application portability, computer language bindings, open systems, operating systems, portable application, POSIX, POSIX language bindings, real-time, thread.

POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1999 by the
Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 1996.
Printed in the United States of America.

ISBN 0-7381-1570-3

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

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

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

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

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of 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 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 all 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.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (508) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Contents

	PAGE
Introduction	xi
Section 1: General	1
1.1 Scope	1
1.2 Normative References	2
1.3 Conformance	3
1.3.1 Implementation Conformance	3
1.3.2 Application Conformance	6
Section 2: Terminology and General Requirements	7
2.1 Editorial Conventions	7
2.2 Definitions	8
2.2.1 Terminology	8
2.2.2 General Terms	10
2.2.3 Abbreviations	26
2.3 General Concepts	27
2.3.1 Process/Active Partition Relationship	27
2.3.2 Task/Thread Relationship	27
2.3.3 Ada Character Differences	27
2.3.4 Posix Signals Are Not Interrupts	28
2.3.5 System Call Exception Errors	28
2.3.6 Extended Security Controls	28
2.3.7 File Access Permissions	28
2.3.8 File Hierarchy	29
2.3.9 Filename Portability	29
2.3.10 File Times Update	29
2.3.11 Pathname Resolution	30
2.3.12 Allocated Storage	31
2.4 Package POSIX	31
2.4.1 Constants and Static Subtypes	36
2.4.2 POSIX Characters	41
2.4.3 POSIX Strings	42
2.4.4 String Lists	43
2.4.5 Option Sets	45
2.4.6 Error Codes and Exceptions	46
2.4.7 System Identification	52
2.4.8 Time Types	53
2.5 Package POSIX_Options	55
2.5.1 Implementation Options	55

2.6	Package POSIX_Limits	58
2.6.1	Implementation Limits	60
2.7	Package Ada_Streams	64
2.8	Package System	65
2.9	Package System_Storage_Elements	66
2.10	Package POSIX_Page_Alignment	67
2.10.1	Description	67
2.10.2	Error Handling	68
2.11	Environment Description	68
2.11.1	Time-Zone Information	69
Section 3: Process Primitives		71
3.1	Package POSIX_Process_Primitives	71
3.1.1	Process Template	73
3.1.2	Process Creation	75
3.1.3	Process Exit	79
3.1.4	Termination Status	80
3.1.5	Wait for Process Termination	82
3.2	Package POSIX_Unsafe_Process_Primitives	84
3.2.1	Process Creation	85
3.2.2	File Execution	87
3.3	Package POSIX_Signals	91
3.3.1	Signal Model	94
3.3.2	Signal Type	98
3.3.3	Standard Signals	98
3.3.4	Default Signal Actions	102
3.3.5	Tasking Safety	103
3.3.6	Interruptibility	103
3.3.7	Signal Sets	105
3.3.8	Block and Unblock Signals	106
3.3.9	Ignore Signals	107
3.3.10	Controlling Generation of Signal for Child Process	108
3.3.11	Examine Pending Signals	109
3.3.12	Signal Event Notification	109
3.3.13	Signal Information	110
3.3.14	Control Signal Queueing	112
3.3.15	Wait for Signal	113
3.3.16	Wait for Signal with Information	115
3.3.17	Signal Entries	116
3.3.18	Send a Signal	118
3.3.19	Queue a Signal	119
3.3.20	Interrupt a Task	120
Section 4: Process Environment		123
4.1	Package POSIX_Process_Identification	123
4.1.1	Process Identification Operations	124
4.1.2	Process Group Identification	125
4.1.3	User Identification	127

4.1.4	User and Group Identification	128
4.2	Package POSIX_Process_Times	130
4.2.1	Process Time Accounting	130
4.3	Package POSIX_Process_Environment	132
4.3.1	Argument List	133
4.3.2	Environment Variables	133
4.3.3	Process Working Directory	137
4.4	Package POSIX_Calendar	138
4.4.1	Time Information	139
4.4.2	Operations on POSIX Times	140
4.5	Package POSIX_Configurable_System_Limits	142
4.5.1	Get Configurable System Options	143
4.5.2	Get Configurable System Limits	145
Section 5:	Files and Directories	149
5.1	Package POSIX_Permissions	149
5.1.1	File Permissions	150
5.1.2	Process Permission Set	151
5.2	Package POSIX_Files	152
5.2.1	Create and Remove Files	153
5.2.2	Inquiries on File Types	156
5.2.3	Modify File Pathnames	156
5.2.4	Directory Iteration	159
5.2.5	Update File Status Information	161
5.2.6	Check File Accessibility	163
5.3	Package POSIX_File_Status	164
5.3.1	Access File Status	165
5.3.2	Access Status Information	166
5.4	Package POSIX_Configurable_File_Limits	168
5.4.1	File Limits	170
5.4.2	File Restrictions	174
5.4.3	Pathname-Specific Options	176
Section 6:	Input and Output Primitives	179
6.1	Package POSIX_IO	179
6.1.1	Open or Create a File	181
6.1.2	I/O Buffer Type	187
6.1.3	Read from a File	187
6.1.4	Write to a File	190
6.1.5	File Position Operations	195
6.1.6	Terminal Operations	196
6.1.7	File Control	196
6.1.8	Update File Status Information	198
6.1.9	Truncate File to A Specified Length	199
6.1.10	Synchronize a File's State	200
6.1.11	Data Synchronization	201
6.2	Package POSIX_File_Locking	202
6.2.1	Lock and Unlock a Region of a File	203

6.3	Package <code>POSIX_Asynchronous_IO</code>	204
6.3.1	AIO Descriptor Type	206
6.3.2	Attributes of AIO Control Blocks	208
6.3.3	Asynchronous Read	211
6.3.4	Asynchronous Write	212
6.3.5	List Directed I/O	214
6.3.6	Retrieve Status of Asynchronous I/O Request	216
6.3.7	Retrieve Bytes Transferred by Asynchronous I/O Request	218
6.3.8	Cancel Asynchronous I/O Request	219
6.3.9	Wait for Asynchronous I/O Request to Complete	220
6.3.10	Asynchronous File and Data Synchronization	221
Section 7: Device- and Class-Specific Functions		225
7.1	General Terminal Interface	225
7.1.1	Interface Characteristics	225
7.2	Package <code>POSIX_Terminal_Functions</code>	232
7.2.1	Terminal Characteristics	234
7.2.2	Input Modes	237
7.2.3	Output Modes	239
7.2.4	Control Modes	239
7.2.5	Local Modes	240
7.2.6	Retrieve and Define Terminal Modes and Bits per Character	242
7.2.7	Special Control Characters	243
7.2.8	Noncanonical Controls	245
7.2.9	Baud Rate Subprograms	245
7.2.10	Line Control Operations	246
7.2.11	Foreground Process Group ID	248
7.2.12	Generate Terminal Pathname	249
Section 8: Language-Specific Services for Ada		251
8.1	Interoperable Ada I/O Services	251
8.1.1	Requirements	251
8.1.2	Additional Interpretation of the POSIX.1 Standard	255
8.2	Package <code>POSIX_Supplement_to_Ada_IO</code>	256
8.2.1	Parse Form values	257
8.2.2	Flush Files	258
Section 9: System Databases		259
9.1	Package <code>POSIX_User_Database</code>	259
9.1.1	Access Contents of a User Database Item	259
9.1.2	Access User Database Items	260
9.2	Package <code>POSIX_Group_Database</code>	261
9.2.1	Access Contents of a Group Database Item	262
9.2.2	Access Elements of the Group Item List of Members	262
9.2.3	Access Group Database Items	263
Section 10: Data Interchange Format		265
Section 11: Synchronization		267

11.1	Package POSIX_Semaphores	267
11.1.1	Semaphore and Semaphore Descriptor Types	268
11.1.2	Initialize an Unnamed Semaphore	269
11.1.3	Finalize an Unnamed Semaphore	270
11.1.4	Create/Open a Named Semaphore	271
11.1.5	Close a Named Semaphore	273
11.1.6	Remove a Named Semaphore	274
11.1.7	Decrement a Semaphore	275
11.1.8	Increment a Semaphore	276
11.1.9	Get the Value of a Semaphore	276
11.2	Package POSIX_Mutexes	277
11.2.1	Mutex Ownership	278
11.2.2	Mutex and Mutex Descriptor Types	279
11.2.3	Mutex Attributes Type	279
11.2.4	Mutex Process Shared Attribute	280
11.2.5	Mutex Locking Policy Attributes	281
11.2.6	Initialize and Finalize a Mutex	283
11.2.7	Change the Ceiling Priority of a Mutex	285
11.2.8	Lock and Unlock a Mutex	286
11.3	Package POSIX_Condition_Variables	287
11.3.1	Condition and Condition Descriptor Types	288
11.3.2	Condition Variable Attributes Type	288
11.3.3	Condition Process Shared Attribute	289
11.3.4	Initialize and Finalize a Condition	290
11.3.5	Broadcast and Signal a Condition	292
11.3.6	Wait on a Condition	293
	Section 12: Memory Management	295
12.1	Package POSIX_Memory_Locking	296
12.1.1	Lock/Unlock a Process's Address Space	296
12.2	Package POSIX_Memory_Range_Locking	298
12.2.1	Lock/Unlock a Range of Process Address Space	298
12.3	Package POSIX_Memory_Mapping	300
12.3.1	Map Process Addresses to a Memory Object	301
12.3.2	Unmap Memory	305
12.3.3	Change Memory Protection	306
12.3.4	Memory Object Synchronization	308
12.4	Package POSIX_Shared_Memory_Objects	309
12.4.1	Open a Shared Memory Object	310
12.4.2	Remove a Shared Memory Object	312
12.5	Package POSIX_Generic_Shared_Memory	313
12.5.1	Open Shared Memory	314
12.5.2	Access Shared Memory	317
12.5.3	Close Shared Memory	317
12.5.4	Remove Shared Memory	318
12.5.5	Lock/Unlock Shared Memory	318
	Section 13: Execution Scheduling	321

13.1	Scheduling Concepts and Terminology	321
13.2	Package <code>POSIX_Process_Scheduling</code>	321
13.2.1	Scheduling Parameters	322
13.2.2	Scheduling Policies	323
13.2.3	Modify Process Scheduling Policy and Parameters	323
13.2.4	Process Yield CPU	325
13.2.5	Get Scheduling Limits	325
13.3	Task Scheduling	326
13.3.1	Dynamic Priorities	326
13.3.2	Task Dispatching Policy Pragma	326
13.3.3	Task Creation Attributes Pragma	327
13.3.4	Task Yield CPU	327
13.4	Synchronization Scheduling	328
Section 14:	Clocks and Timers	329
14.1	Package <code>POSIX_Timers</code>	329
14.1.1	Clock and Timer Types	330
14.1.2	Realtime Clock	330
14.1.3	Timer State and Timer Options	331
14.1.4	Clock Operations	332
14.1.5	Create a Timer	333
14.1.6	Delete a Timer	335
14.1.7	Timer Operations	335
14.2	High Resolution Delay	338
Section 15:	Message Passing	339
15.1	Package <code>POSIX_Message_Queue</code> s	339
15.1.1	Message Queue Attributes	341
15.1.2	Open a Message Queue	343
15.1.3	Close a Message Queue	346
15.1.4	Remove a Message Queue	347
15.1.5	Send a Message to a Message Queue	348
15.1.6	Receive a Message from a Message Queue	349
15.1.7	Generic Message Passing	350
15.1.8	Notify Process that a Message is Available	352
15.1.9	Set/Query Message Queue Attributes	353
Section 16:	Task Management	355
16.1	Package <code>Ada_Task_Identification</code>	355
16.1.1	Description	355
Annex A	(informative) Bibliography	357
Annex B	(informative) Rationale and Notes	359
B.1	General	359
B.1.1	Purpose and Audience	359
B.1.2	Relation to Other POSIX Standards	359
B.1.3	Ada Language	360
B.1.4	Implementation Model	361

B.1.5	Level of Binding	361
B.1.6	Form of Document	362
B.1.7	Global Issues	363
B.1.8	Naming	364
B.1.9	Mapping C Features to Ada	365
B.1.10	Conformance	370
B.2	Terminology and General Requirements	370
B.2.1	Conventions	370
B.2.2	Definitions	370
B.2.3	General Concepts	371
B.2.4	Package POSIX	374
B.2.5	Package POSIX_Options	380
B.2.6	Package POSIX_Limits	381
B.2.7	Package Ada_Streams	381
B.2.8	Package System	381
B.2.9	Package System_Storage_Elements	381
B.2.10	Package POSIX_Page_Alignment	381
B.2.11	Environment Description	382
B.3	Process Primitives	382
B.3.1	Package POSIX_Process_Primitives	382
B.3.2	Process Creation	383
B.3.3	Process Exit	384
B.3.4	Wait for Process Termination	386
B.3.5	Package POSIX_Unsafe_Process_Primitives	386
B.3.6	Package POSIX_Signals	386
B.3.7	Signal Model	387
B.3.8	Signal Masking and Related Concepts	387
B.3.9	Tasking Safety	391
B.3.10	Signal Type	392
B.3.11	Standard Signals	392
B.3.12	Signal Sets	393
B.3.13	Examine and Change Signal Action	393
B.3.14	Signal Queueing	394
B.3.15	Signal Notification Model	395
B.3.16	Examine Pending Signals	396
B.3.17	Send a Signal	396
B.3.18	Binding for <i>sigsuspend</i>	396
B.3.19	Synchronously Accept a Signal	396
B.3.20	Queue a Signal to a Process	397
B.3.21	Send a Signal to a Thread	397
B.3.22	Schedule Alarm	398
B.3.23	Suspend Process Execution	398
B.3.24	Delay Process Execution	398
B.3.25	Task Signal Entries	398
B.3.26	Composability Considerations	399
B.4	Process Environment	400
B.4.1	Organization	400
B.4.2	Package POSIX_Process_Identification	400

B.4.3	Package Process_Times	401
B.4.4	Package POSIX_Process_Environment	402
B.4.5	Package POSIX_Calendar	404
B.4.6	Package POSIX_Configurable_System_Limits	405
B.5	Files and Directories	405
B.5.1	Organization	405
B.5.2	Package POSIX_Permissions	406
B.5.3	Package POSIX_Files	407
B.5.4	Package POSIX_File_Status	408
B.5.5	Package POSIX_Configurable_File_Limits	408
B.5.6	Directory Operations	409
B.6	Input and Output Primitives	409
B.6.1	Package POSIX_IO	409
B.6.2	Package POSIX_File_Locking	414
B.6.3	Package POSIX_Asynchronous_IO	415
B.7	Device- and Class-Specific Functions	417
B.8	Language-Specific Services for Ada	418
B.8.1	General Rationale	418
B.8.2	Limitations on Interoperability	420
B.8.3	Rationale for Form Parameter	421
B.8.4	Unaffected Implementation Dependencies	426
B.8.5	Notes on Specific Topics	426
B.9	System Databases	428
B.9.1	Requirements from POSIX.1	428
B.9.2	Rationale for the Current Design	429
B.10	Data Interchange Format	430
B.11	Synchronization	430
B.11.1	Package POSIX_Semaphores	430
B.11.2	Mutexes & Condition Variables	433
B.12	Memory Management	436
B.12.1	Process Memory Locking	436
B.12.2	Package POSIX_Memory_Mapping	437
B.12.3	Package POSIX_Shared_Memory_Objects	438
B.12.4	Package POSIX_Generic_Shared_Memory	439
B.13	Priority Process Scheduling	441
B.13.1	Scheduling Concepts and Terminology	442
B.13.2	Package POSIX_Process_Scheduling	442
B.13.3	Task Scheduling	443
B.13.4	Synchronization Scheduling	444
B.13.5	Thread Scheduling Pragmas vs. Environment Variables	445
B.14	Clocks and Timers	446
B.14.1	Types and Constants	446
B.14.2	Clock Operations	447
B.14.3	Timer Creation	447
B.14.4	Timer Operations	447
B.14.5	High Resolution Sleep	448
B.15	Message Passing	448
B.15.1	Message Queue Attributes	449

B.15.2	Priorities	449
B.15.3	Generic Message Passing	449
B.15.4	Notification	450
B.16	Task Identification	450
B.17	Thread-Specific Data	450
Annex C (informative)	Ada/C Cross-References	451
C.1	Ada-to-C Cross-Reference	451
C.2	C-to-Ada Cross-Reference	469
Alphabetic Topical Index		489

TABLES

Table2.1	– Typographical Conventions	7
Table2.2	– Constant and Subtype Correspondences	37
Table2.3	– Option Set Comparisons	46
Table2.4	– Static Subtypes and Options	59
Table2.5	– Portable Constants and Limits	63
Table2.6	– Static Subtypes and Limits	64
Table3.1	– Default Actions for Job Control Signals	103
Table4.1	– Functions for System-Wide Options	145
Table4.2	– Configurable System Limits	146
Table6.1	– Standard File Descriptors	182
Table6.2	– Error Codes and AIO Status Values	217
Table7.1	– Terminal_Characteristics Components	235
Table7.2	– Terminal_Modes Values for Input Control	237
Table7.3	– Terminal_Modes Values for Output Control	239
Table7.4	– Terminal_Modes Values for Hardware Control	240
Table7.5	– Terminal_Modes Values for Local Control Modes	241
Table7.6	– Special Control Character Usage	244

International Standard ISO/IEC 14519:1999(E)

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

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

International Standard ISO/IEC 14519 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Annexes A to C of this International Standard are for information only.



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

Introduction

(This introduction is not a part of IEEE Std 1003.5b-1996, IEEE Standard for Information Technology—POSIX Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API) Amendment 1: Realtime Extensions but is included for information only.)

This standard is an amended version of IEEE Std 1003.5-1992. 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: 1990 (IEEE Std 1003.1-1990), as amended by IEEE Std 1003.1b-1993, IEEE Std 1003.1c-1995, and IEEE Std 1003.1i-1995.

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 sufficient to implement an Ada compilation system or a POSIX shell (as defined in IEEE Std 1003.2-1992). 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 16)

The content of the sections parallels that of the correspondingly numbered sections of the base standards (IEEE Std 1003.1-1990, as amended by IEEE Std 1003.1b-1993,

IEEE Std 1003.1c-1995, and IEEE Std 1003.1i-1995), with some changes required to fit the style of IEEE Std 1003.5-1993. There is no Section 10, since there is no Ada binding for that section (Data Interchange Formats) of the base standards.

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 are in progress, and similar efforts can be anticipated in the future.

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

- (1) Language-independent descriptions of the services defined by IEEE-Std 1003.1-1990 and its amendments.
- (2) C and FORTRAN language bindings to (1)
- (3) Shell and utility facilities
- (4) Verification testing methods
- (5) Secure/Trusted system considerations
- (6) Network interface facilities
- (7) System administration
- (8) Graphical user interfaces
- (9) Profiles describing application- or user-specific combinations of open systems standards for: supercomputing, multiprocessor, and batch extensions; transaction processing; realtime systems; and multiuser systems based on historical models
- (10) An overall guide to POSIX-based or related Open Systems standards and profiles

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

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

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

¹A *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 this standard 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 Jespersen
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 Jespersen
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
Bhavesb 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

1 The following persons were members of the 1003.5 balloting group that approved the
2 standard for submission to the IEEE Standards Board:

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

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

1 When the IEEE Standards Board approved this standard on 20 June 1996, it had
2 the following membership:

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

4 **Andrew G. Salem**, *Secretary*

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

14 *Member Emeritus

15 Also included are the following nonvoting IEEE Standards Board liaisons:

16	Satish K. Aggarwal
17	Alan H. Cookson
18	Chester C. Taylor
19	Lisa S. Young
20	<i>IEEE Standards Project Editor</i>

The POSIX.5b standard was prepared by the P1003.5 working group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time this standard 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: Jay Ashford
Andrew Josey
Barry Needham
Stephen R. Walli
Secretary: Nick Stoughton

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

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

1 The following persons were members of the 1003.5b balloting group that approved
2 the standard for submission to the IEEE Standards Board:

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Information technology— POSIX[®] Ada Language Interfaces— Part 1: Binding for System Application Program Interface (API)— Amendment 1: Realtime Extensions

Section 1: General

1.1 Scope

This standard defines a standard Ada language binding to POSIX.1 {2}, as amended by POSIX.1b {3}, POSIX.1c {4}, and POSIX.1i, which are hereinafter called *the base standards*. (See §1.2.) The base standards define C-language interfaces to operating system services.¹ This standard defines interfaces by which the same operating system services may be accessed from the Ada programming language.

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 both the original Ada language standard (ISO/IEC 8652: 1987), and the 1995 revision to that standard (ISO/IEC 8652: 1995).

This standard is comprised of three major components:

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

¹Language-independent definitions of these standards were once under development, but work on those projects has been suspended.

- 1 — Definitions of the specific Ada interfaces to the system services defined by the
2 base POSIX standard, presented in the form of Ada packages.
- 3 — Interpretations of Ada semantics with respect to the base standards.

4 The following areas are outside the scope of this standard:

- 5 (1) User interface (shell) and commands associated with Ada program development.
- 6 (2) Ada bindings to the archive/interchange file formats for *tar* and *cpio*.
- 7 (3) Network protocols.
- 8 (4) Graphics and windowing interfaces.
- 9 (5) Database management system interfaces.
- 10 (6) Object or binary code portability.
- 11 (7) System configuration and resource availability.
- 12 (8) Interfaces to the Ada runtime system.

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

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

20 1.2 Normative References

21 The following standards contain provisions that, through reference in this text, con-
22 stitute provisions of this standard. At the time of publication, the editions indicated
23 were valid. All standards are subject to revision, and parties to agreements based
24 on this standard are encouraged to investigate the possibility of applying the most
25 recent editions of the standards indicated below. Members of IEC and ISO maintain
26 registers of currently valid International Standards.

- 27 {1} ISO/IEC 8652: 1995, *Information technology— Programming languages— Ada*
28 [Revision of first edition (ISO 8652:1987)], 15 February 1995. Hereinafter, this
29 is referred to as Ada RM {1}¹.
- 30 {2} ISO/IEC 9945-1: 1996 (IEEE Std 1003.1-1996), *Information technology— Port-*
31 *able Operating System Interface (POSIX)— Part 1: System Application Program*
32 *Interface (API) [C language]*. Hereinafter, this is referred to as POSIX.1 {2}².
- 33 {3} IEEE Standard 1003.1b-1993, *Information Technology— Portable Operating*
34 *System Interface (POSIX)— Part 1: System Application Program Interface (API)*
35 *[C Language]— Amendment 1: Realtime Extension*. Hereinafter, this is referred
36 to as POSIX.1b {3}².

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

39 ²This standard is available from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway,
40 NJ 08855-1331, USA. Telephone: 1 (800) 678-IEEE or +1 (908) 981-1393 (outside US).

{4} IEEE Standard 1003.1c-1995, *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API)—Amendment 2: Threads Extension [C Language]*. Hereinafter, this is referred to as POSIX.1c {4}².

{5} IEEE Standard 1003.1i-1995, *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Programming Interface (API)—Amendment: Technical Corrigenda to Realtime Extension [C Language]*. Hereinafter, this is referred to as POSIX.1i².

NOTE: Items 2, 3, 4, and 5 are the base standards.

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 herein.
- (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 defined configuration pragma [Ada RM {1} 10.1.5 (8)].

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:

- (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.

²This standard is available from the IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA. Telephone: 1 (800) 678-IEEE or +1 (908) 981-1393 (outside US).

- 1 (f) Adding specific declarations for which permission is explicitly granted in
2 the text of this standard.
- 3 (g) Changing packages with names of the form POSIX_XXX to child packages of
4 the package POSIX with names of the form POSIX.XXX, and adding library-
5 level renaming declarations of the form:
6 **with** POSIX.XXX;
7 **package** POSIX_XXX **renames** POSIX.XXX;
- 8 (h) Where there is a subprogram which has a parameter with a default value
9 expression in the final position, the default value expression may be omitted
10 and an equivalent overloaded definition of the same function name, without
11 the last parameter, may be provided instead, if this results in a gain in
12 efficiency. For example,
13 **procedure** P(X: T:= E);
14 may be replaced by
15 **procedure** P(X: T);
16 **procedure** P;
17 Calling the version of procedure P without the parameter X should be equiv-
18 alent to calling P(X=>E).
- 19 (4) Only the following types may be implemented using Ada access types:
20 — POSIX.POSIX_String_List
21 — POSIX_Asynchronous_IO.AIO_Descriptor
22 — POSIX_Condition_Variables.Attributes
23 — POSIX_Condition_Variables.Condition
24 — POSIX_Condition_Variables.Condition_Descriptor
25 — POSIX_Message_Queues.Message_Queue_Descriptor
26 — POSIX_Mutexes.Attributes
27 — POSIX_Mutexes.Mutex
28 — POSIX_Mutexes.Mutex_Descriptor
29 — POSIX_Process_Environment.Environment
30 — POSIX_Process_Primitives.Process_Template
31 — POSIX_Semaphores.Semaphore
32 — POSIX_Semaphores.Semaphore_Descriptor
33 — Any other type specifically defined as an Ada access type in this standard
34 No other types defined in this standard shall be implemented using Ada access
35 types.
- 36 (5) A POSIX implementation shall not raise **Program_Error** on elaboration of a
37 POSIX package, or on execution of a POSIX subprogram, due to elaboration
38 order dependencies in the POSIX implementation.
- 39 (6) Except as explicitly provided for in this standard, **POSIX.POSIX_Error** or **Stan-**
40 **dard.Storage_Error** are the only exceptions that shall be raised by operations
41 declared in this standard.
- 42 (7) Wherever this standard states a requirement on an “implementation,” without
43 any other explicit qualification, the requirement applies to all conforming imple-
44 mentations.

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 Conforming POSIX.5 Application. 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 providing either a specific reference to the system documentation or providing full syntax and semantics of these features. The conformance document may specify the behavior of the implementation for those features where this 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 Ada RM {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 applications:

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 a Conforming POSIX.5 Application 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

This standard uses the following editorial and typographical conventions. A summary of typographical conventions is shown in Table 2.1.

Table 2.1 – Typographical Conventions

Reference	Example
Ada Identifier	POSIX_String_List
Ada Implementation Defined Element	<i>implementation-defined</i>
Ada Reserved Word	generic
C-Language Data Type	<i>long</i>
C-Language Error Number	[EINTR]
C-Language Function	<i>system()</i>
C-Language Global External	<i>errno</i>
C-Language Header	<sys/stat.h>
Conceptual Attribute	Message Length
Citation	[Ada RM {1} 9.1 (c)]
Cross-Reference: Annex	Annex A
Cross-Reference: Clause	§2.3
Cross-Reference: Other Standard	Ada RM {1}
Cross-Reference: Section	Section 2
Cross-Reference: Subclause	§2.3.4, §2.3.4.5, 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 in “POSIX.1 {2},” represent cross references to the normative reference’s clause (see §1.2). If the number is preceded by a B, it represents a bibliographic entry (see 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]. At the conclusion of this type of definition, the designation of a standard within [brackets] indicates that the text of the definition was copied from that standard, with only editorial changes, if any.

(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 inline; 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 smaller point size and are indented 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.

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

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.

2.2.1.4 obsolescent: An indication that a certain feature may be considered for withdrawal in future revisions of this standard. Use of obsolescent features is deprecated.

NOTE: Obsolescent features are retained for upward compatibility from POSIX.5 {B3}, or to maintain symmetry with certain aspects of the C-language binding to POSIX.1 {2}. Their use in Ada applications is discouraged.

2.2.1.5 shall: An indication of a requirement on the implementation or on Strictly Conforming POSIX.5 Applications, where appropriate.

2.2.1.6 should:

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

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

2.2.1.7 supported: A condition regarding optional functionality.

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.

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

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

2.2.1.9 undefined: An indication that this 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.

2.2.1.10 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.

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 is allowed to complete without being affected by abortion. [Ada RM {1} 9.8 (5)] 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”. [Ada RM {1} 9.8 (15)]

2.2.2.3 absolute pathname: See *pathname resolution* in §2.3.11.

2.2.2.4 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.

See *file access permissions* in §2.3.7.

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

2.2.2.6 address space: The memory locations that can be referenced by a process. [POSIX.1 {2}]

2.2.2.7 appropriate privileges: An implementation defined means of associating privileges with a 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. [POSIX.1 {2}]

2.2.2.8 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.

2.2.2.9 asynchronous I/O operation: An I/O operation that does not of itself cause the task requesting the I/O to be blocked. This implies that the requesting task and the I/O operation may be running concurrently.

2.2.2.10 asynchronous I/O completion: An asynchronous read or write operation is complete when a corresponding synchronous read or write would have completed and any associated status attributes have been updated.

2.2.2.11 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 (see §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.

2.2.2.12 background process: A process that is a member of a background process group. [POSIX.1 {2}]

2.2.2.13 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. [POSIX.1 {2}]

2.2.2.14 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. [POSIX.1 {2}]

2.2.2.15 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.

2.2.2.16 blocked task: An Ada task that is not running or ready to run, as defined in [Ada RM {1} 9 (10)]. 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.

2.2.2.17 byte: A byte is 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 §2.4.1.4. 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.

2.2.2.18 canonical input processing: The processing of terminal input in the form of text lines. (See §7.1.1.6.)

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

NOTE: This is the definition of the term *character* when it is used by itself. It does not apply to qualified phrases containing the word *character*, such as “POSIX character” (see §2.2.2.95), “graphic character” (see §2.2.2.58), “Ada character” (see §2.3.3), and “character special file” (see §2.2.2.20).

2.2.2.20 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. [POSIX.1 {2}]

2.2.2.21 child process: See *POSIX process* in §2.2.2.97.

2.2.2.22 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. [POSIX.1b {3}]

2.2.2.23 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. [Ada RM- {1} 7.6.1 (2)]

2.2.2.24 condition variable: A synchronization object which 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 “signal” operation on the condition variable. In use, condition variables are always associated with mutexes. (See §2.2.2.73.)

2.2.2.25 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. [POSIX.1 {2}]

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

Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal (see §7.1) cause signals to be sent to all processes in the process group associated with the controlling terminal. [POSIX.1 {2}]

2.2.2.27 current working directory: See *working directory* in §2.2.2.152.

- 1 **2.2.2.28 device:** A computer peripheral or an object that appears to the applica-
2 tion as such. [POSIX.1 {2}]
- 3 **2.2.2.29 directory:** A file that contains directory entries. [POSIX.1 {2}]
- 4 No two entries in a directory shall have the same filename.
- 5 **2.2.2.30 directory entry [link]:** A relation that associates a filename with a file.
6 [POSIX.1 {2}]
- 7 Several directory entries can associate different filenames with the same file.
- 8 **2.2.2.31 disarm a timer:** To stop a timer from measuring the passage of time,
9 thereby disabling any future process notifications (until the timer is armed again).
10 [POSIX.1b {3}]
- 11 **2.2.2.32 dot:** The filename consisting of a single dot character (.).
- 12 See *pathname resolution* in §2.3.11. [POSIX.1 {2}]
- 13 **2.2.2.33 dot-dot:** The filename consisting of (..).
- 14 See *pathname resolution* in §2.3.11. [POSIX.1 {2}]
- 15 **2.2.2.34 effective group ID:** An attribute of a process that is used in determin-
16 ing various permissions, including the file access permissions, described in §2.3.7.
17 [POSIX.1 {2}]
- 18 See *group ID* in §2.2.2.59. This value is subject to change during the process lifetime,
19 as described in packages `POSIX_Process_Identification` (see §4.1) and `POSIX_-`
20 `Process_Primitives` (see §3.1).
- 21 **2.2.2.35 effective user ID:** An attribute of a process that is used in determining
22 various permissions, including file access permissions. [POSIX.1 {2}]
- 23 See *user ID* in §2.2.2.150. This value is subject to change during the process lifetime,
24 as described in packages `POSIX_Process_Identification` (see §4.1) and `POSIX_-`
25 `Process_Primitives` (see §3.1).
- 26 **2.2.2.36 empty directory:** A directory that contains, at most, entries for dot and
27 dot-dot. [POSIX.1 {2}]
- 28 **2.2.2.37 empty string [null string]:** A zero-length array whose components are
29 of some character type.
- 30 **2.2.2.38 environment task:** The anonymous task whose execution elaborates
31 the library items of the declarative part of an active partition, and then calls the
32 main subprogram, if there is one. [Ada RM {1} 10.2 (8)]
- 33 **2.2.2.39 epoch:** A base reference time defined as 0 hours, 0 minutes, 0.0 seconds,
34 1 January 1970, Universal Coordinated Time{B2}.

1 **2.2.2.40 erroneous execution:** The term *erroneous execution* is used in this
2 standard as defined in [Ada RM {1}1.1.5].

3 **2.2.2.41 error code of a task:** An attribute of a task, that ordinarily specifies
4 information about the most recent error that caused `POSIX_Error` to be raised.
5 (See §2.4.6.)

6 **2.2.2.42 fifo special file [FIFO]:** A type of file with the property that data writ-
7 ten to such a file is read on a first-in-first-out basis. [POSIX.1 {2}]

8 Other characteristics of FIFOs are described in packages `POSIX_Files` (see §5.2)
9 and `POSIX_IO` (see §6.1).

10 **2.2.2.43 file:** An object that can be written to, or read from, or both.
11 A file has certain attributes, including access permissions and type. File types in-
12 clude regular file, character special file, block special file, FIFO special file, and di-
13 rectory. Other types of files may be defined by the implementation. [POSIX.1 {2}]

14 **2.2.2.44 file description:** See *open file description* in §2.2.2.79.

15 **2.2.2.45 file descriptor:** A per-process unique non-negative integer value used
16 to identify an open file for the purpose of file access. [POSIX.1b {3} 2.2.2.37]

17 **2.2.2.46 file group class:** A property of a file indicating access permissions for a
18 process related to the group identification of the process.

19 A process is in the file group class of a file if the process is not in the file owner class
20 and if the effective group ID or one of the supplementary group IDs of the process
21 matches the group ID associated with the file. Other members of the class may be
22 implementation defined. [POSIX.1 {2}]

23 **2.2.2.47 filename:** A nonempty string that is used to name a file. A filename
24 consists of, at most, `POSIX_Limits.Filename_Maxima` 'Last' components of type
25 `POSIX.POSIX_Character`. The characters composing the name may be selected
26 from the set of all the character values excluding the slash character and the null
27 character. The filenames dot and dot-dot have special meaning; see *pathname res-*
28 *olution* in §2.3.11. A filename is sometimes referred to as a pathname component.
29 [POSIX.1 {2}]

30 **2.2.2.48 file offset:** The byte position in the file where the next I/O operation
31 begins. [POSIX.1 {2}]

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

35 **2.2.2.49 file other class:** A property of a file indicating access permissions for a
36 process related to the user and group information of the process.

37 A process is in the file other class of a file if the process is not in the file owner class
38 or file group class. [POSIX.1 {2}]

1 **2.2.2.50 file owner class:** A property of a file indicating access permissions for a
2 process related to the user identification of the process.

3 A process is in the file owner class of a file if the effective user ID of the process
4 matches the user ID of the file. [POSIX.1 {2}]

5 **2.2.2.51 file permission:** Information about a file that is used, along with other
6 information, to determine if a process has read, write, or execute/search permission
7 to a file. [POSIX.1 {2}]

8 The file permission information is divided into three parts: owner, group, and other.
9 Each part is used with the corresponding file class of processes. These permissions
10 are contained in the *file mode*, as described in the package POSIX_Permissions
11 (see §5.1). The detailed usage of the file permission information in access decisions
12 is described in *file access permissions* in §2.3.7.

13 **2.2.2.52 file serial number:** A per-file-system unique value for a file.

14 File serial numbers are unique throughout a file system. [POSIX.1 {2}]

15 **2.2.2.53 file system:** A collection of files, together with certain of their attributes.
16 [POSIX.1 {2}]

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

21 **2.2.2.54 first open, of a file:** Occurs when a process opens a file, message queue,
22 or shared memory object that is not currently open within any process.

23 **2.2.2.55 foreground process:** A process that is a member of a foreground pro-
24 cess group. [POSIX.1 {2}]

25 **2.2.2.56 foreground process group:** A group of processes that have certain
26 privileges, denied to processes in background process groups, when accessing their
27 controlling terminal.

28 Each session that has established a connection with a controlling terminal has ex-
29 actly one process group of the session as the foreground process group of that con-
30 trolling terminal. See §7.1.1.4. [POSIX.1 {2}]

31 **2.2.2.57 foreground process group ID:** The process group ID of the foreground
32 process group. [POSIX.1 {2}]

33 **2.2.2.58 graphic character:** A graphic character is a sequence of one or more
34 POSIX.POSIX_Characters representing a single graphic symbol.

1 **2.2.2.59 group ID:** A value identifying a group of system users.

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

7 **2.2.2.60 job control:** A facility that allows users to stop (suspend) selectively
8 the execution of processes and continue (resume) their execution at a later time.
9 [POSIX.1 {2}]

10 The user typically employs this facility via the interactive interface jointly supplied
11 by the terminal I/O driver and a command interpreter. Conforming implementations
12 may optionally support job control facilities. The presence of this option is indicated
13 to the application at compile time by the subtype `Job_Control_Support` in pack-
14 age `POSIX_Options` or at run time by the value returned by the function `Job_Con-`
15 `trol_Is_Supported` in package `POSIX_Configurable_System_Limits`; see §2.4
16 and §4.5.

17 **2.2.2.61 last close, of a file:** The act of a process closing a file, message queue,
18 or shared memory object that results in the file, message queue, or shared memory
19 object no longer being open within any process.

20 **2.2.2.62 link:** See *directory entry* in §2.2.2.30.

21 **2.2.2.63 link count:** The number of directory entries that refer to a particular
22 file. [POSIX.1 {2}]

23 **2.2.2.64 login:** The unspecified (by this standard) activity by which a user gains
24 access to the system. [POSIX.1 {2}]

25 Each login shall be associated with exactly one login name. [POSIX.1 {2}]

26 **2.2.2.65 login name:** A user name that is associated with a login. [POSIX.1 {2}]

27 **2.2.2.66 map a range of addresses:** To create an association between a range
28 of a process's address space and a range of physical memory or some memory object,
29 such that a reference to an address in that range of the address space results in a
30 reference to the associated physical memory or memory object. The mapped memory
31 or memory object is not necessarily memory-resident. [POSIX.1b {3}]

32 **2.2.2.67 memory object:** Either a file or shared memory object. When used in
33 conjunction with `Map_Memory`, `Open_And_Map_Shared_Memory` or `Open_Or_Cre-`
34 `ate_And_Map_Shared_Memory`, a memory object will appear in the calling process's
35 address space. [POSIX.1b {3}]

36 **2.2.2.68 memory-resident:** Managed by the implementation in such a way as to
37 provide an upper bound on memory access times. [POSIX.1b {3}]

1 **2.2.2.69 message:** Information that can be transferred among tasks (possibly in
2 different processes) by being added to and removed from a message queue.

3 **2.2.2.70 message queue:** An object to which messages can be added and re-
4 moved. Characteristics and interfaces associated with message queues are defined
5 in Section 15.

6 **2.2.2.71 message queue descriptor:** A per-process unique value used to iden-
7 tify an open message queue.

8 **2.2.2.72 mode:** A collection of attributes that specifies the type of a file and its
9 access permissions.

10 See *file access permissions* in §2.3.7. [POSIX.1 {2}]

11 **2.2.2.73 mutex:** A synchronization object used to allow multiple tasks (possibly
12 in different processes) to serialize their access to shared data or other shared re-
13 sources. The name derives from the capability it provides, namely, mutual exclusion.

14 **2.2.2.74 mutex owner:** The task that has locked a mutex becomes its owner and
15 remains the owner until that same task unlocks the mutex.

16 **2.2.2.75 noncanonical input processing:** The processing of terminal input as
17 uninterpreted *characters* (see Section 7).

18 **2.2.2.76 null character:** The value `POSIX_Character'Value(0)`, if defined.

19 The use of a null character in filenames or environment variable names or values
20 produces undefined results. Implementations shall not return null characters to ap-
21 plications in filenames or environment variable names or values.

22 **2.2.2.77 null string:** See *empty string* in §2.2.2.37.

23 **2.2.2.78 open file:** A file that is currently associated with a file descriptor. [PO-
24 SIX.1 {2}]

25 **2.2.2.79 open file description:** A record of how a process or group of processes
26 is accessing a file.

27 Each file descriptor shall refer to exactly one open file description, but an open file
28 description may be referred to by more than one file descriptor. A file offset, file
29 status (see §6.1.7), and file access modes are attributes of an open file description.
30 [POSIX.1 {2}]

31 **2.2.2.80 orphaned process group:** A process group in which the parent of every
32 member is either itself a member of the group or is not a member of the session of
33 the group. [POSIX.1 {2}]

34 The process group is no longer a member of the session of the process that created it.
35 This can occur when all other members of the session exit.

2.2.2.81 page: The granularity of memory mapping and locking.

A fixed-length contiguous range of a process's address space. Physical memory and memory objects can be mapped into a process's address space 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.

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.

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

NOTE: Unlike POSIX.1 {2}, the page size is given in storage units here rather than bytes. This is 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.

2.2.2.83 parent directory:

(A) 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.

(B) 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. [POSIX.1 {2}]

2.2.2.84 parent process: See *POSIX process* in §2.2.2.97.

2.2.2.85 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. [POSIX.1 {2}]

2.2.2.86 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. [Ada RM {1} 10.2 (2)] An active partition corresponds to a POSIX process.

2.2.2.87 path prefix: A pathname, with an optional ending slash, that refers to a directory. [POSIX.1 {2}]

2.2.2.88 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 described in §2.3.11.

2.2.2.89 pathname component: See *filename* in §2.2.2.47.

2.2.2.90 permission: See *file permission* in §2.2.2.51.

2.2.2.91 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.

2.2.2.92 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.

2.2.2.93 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.

2.2.2.94 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 (/).

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

2.2.2.96 posix I/O: The input/output operations defined by this standard, except for Section 8.

2.2.2.97 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. (See §2.3.1.) A

1 POSIX process is said to perform an action if any of the conceptual threads of control
2 within it performs the action.

3 Many of the *system resources* defined by this standard are shared among all of the
4 threads of control within a process. These include (but are not limited to): the process
5 ID; the parent process ID; process group ID; session membership; real, effective, and
6 saved set-user-ID; real, effective, and saved set-group-ID; supplementary group IDs;
7 current working directory; root directory; file mode creation mask; file descriptors;
8 timers; file locks; mutexes; environment variables; process argument list; possibly, a
9 process signal mask. (See §3.3.1.)

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

15 **2.2.2.98 potentially blocking operation:** An operation that is not allowed
16 within a protected action, because it may be required to block the calling task.
17 [Ada RM {1} 9.5.1 (8)] Certain operations are defined by the Ada language to be po-
18 tentially blocking. In addition, every operation defined by this standard whose effect
19 is defined as blocking the calling task under any circumstance is also a potentially
20 blocking operation.

21 **2.2.2.99 priority:** The general term for an integer-valued attribute of processes,
22 tasks, messages, and asynchronous I/O operations, whose value is used in selecting
23 among entities of the same kind. Numerically higher values represent higher priori-
24 ties, and are given preference for selection over lower priorities.

25 The *priority of an Ada task* is an integer that indicates a degree of urgency and is
26 the basis for resolving competing demands of tasks for resources. Unless otherwise
27 specified, whenever tasks compete for processors or other implementation defined
28 resources, the resources are allocated to the task with the highest priority value.
29 The *base priority* of a task is the priority with which it was created, or to which it
30 was later set by `Dynamic_Priorities.Set_Priority` (see [Ada RM {1} D.5]). At all
31 times, a task also has an *active priority*, which generally reflects its base priority as
32 well as any priority it inherits from other sources. *Priority inheritance* is the process
33 by which the priority of a task or other entity (e.g., a protected object; see [Ada RM-
34 {1} D.3]) is used in the evaluation of another task's active priority. [Ada RM {1} D.1
35 (15)]

36 At any time, the active priority of a task is the maximum of all the priorities the task
37 is inheriting at that instant. [Ada RM {1} D.1 (20)]

38 **2.2.2.100 privilege:** See *appropriate privileges* in §2.2.2.7.

39 **2.2.2.101 process:** See *POSIX process* in §2.2.2.97.

40 **2.2.2.102 process group:** A collection of processes that permits the signaling of
41 related processes.

1 Each process in the system is a member of a process group that is identified by its
2 process group ID. A newly created process joins the process group of its creator. [PO-
3 SIX.1 {2}]

4 **2.2.2.103 process group ID:** A unique value identifying a process group during
5 its lifetime.

6 A process group ID shall not be reused by the system until the process group lifetime
7 ends. [POSIX.1 {2}]

8 **2.2.2.104 process group leader:** The unique process, within a process group,
9 that created the process group.

10 **2.2.2.105 process group lifetime:** A period of time that begins when a process
11 group is created and ends when the last remaining process in the group leaves the
12 group, due either to the end of the process lifetime of the last process or to the last
13 remaining process calling the Set_Process_Group_ID procedure. [POSIX.1 {2}]

14 **2.2.2.106 process ID:** A unique value identifying a process during its lifetime.

15 The process ID is defined in the package POSIX_Process_Identification. (See
16 §4.1.) A process ID shall not be reused by the system until the process lifetime ends.
17 In addition, if there exists a process group where the process ID of the process group
18 leader is equal to that process ID, that process ID shall not be reused by the system
19 until the process group lifetime ends.

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

22 **2.2.2.107 process lifetime:** A period of time that begins when a process is cre-
23 ated and ends when its process ID is returned to the system. [POSIX.1 {2}]

24 After a process is created, it is considered active. Its threads of control and address
25 space exist until it terminates. It then enters an inactive state where certain re-
26 sources may be returned to the system, although some resources, such as the process
27 ID, are still in use. When another process executes a Wait_For_Child_Process
28 procedure for an inactive process, the remaining resources are returned to the sys-
29 tem. The last resource to be returned to the system is the process ID. At this time,
30 the lifetime of the process ends. [POSIX.1 {2}]

31 **2.2.2.108 program:** A set of partitions, which can execute in parallel with one
32 another, possibly in a separate address space, and possibly on a separate computer.
33 [Ada RM {1} 10.2 (1)]

34 **2.2.2.109 read-only file system:** A file system that has implementation defined
35 characteristics restricting modifications. [POSIX.1 {2}]

36 **2.2.2.110 ready task:** A task that is not blocked. (See §2.2.2.16.) The ready
37 tasks include those that are running as well as those that are waiting for a processor.

- 1 NOTE: For consistency, this standard uses the definition of *ready* from Ada RM {1},
2 which includes running tasks among those that are ready. Thus the meaning of *ready*
3 is close to the meaning of *runnable* as used in POSIX.1c {4}, but the two terms differ in
4 that *ready* includes running threads. This difference in terminology requires differences
5 in several descriptions between this Ada binding and the base POSIX standards, to
6 preserve compatible semantics.
- 7 **2.2.2.111 real group ID:** The attribute of a process that, at the time of process
8 creation, identifies the group of the user who created the process.
- 9 See *group ID* in §2.2.2.59. This value is subject to change during the process lifetime,
10 as described in §4.1. [POSIX.1 {2}]
- 11 **2.2.2.112 real user ID:** The attribute of a process that, at the time of process
12 creation, identifies the user who created the process.
- 13 See *user ID* in §2.2.2.150. This value is subject to change during the process lifetime,
14 as described in §4.1. [POSIX.1 {2}]
- 15 **2.2.2.113 referenced shared memory object:** A shared memory object that is
16 open or has one or more mappings defined on it. [POSIX.1b {3}]
- 17 **2.2.2.114 region:** As relates to the address space of a process, a sequence of ad-
18 dresses.
- 19 As relates to a file, a sequence of offsets.
- 20 **2.2.2.115 regular file:** A file that is a randomly accessible sequence of bytes,
21 with no further structure imposed by the system.
- 22 **2.2.2.116 relative pathname:** See *pathname resolution* in §2.3.11.
- 23 **2.2.2.117 reserved signal:** Signals which the application cannot accept, and for
24 which the application cannot modify the signal action or masking, because the sig-
25 nals are reserved for use by the Ada language implementation. The reserved signals
26 defined by this standard are `Signal_Abort`, `Signal_Alarm`, `Signal_Floating_`
27 `Point_Error`, `Signal_Illegal_Instruction`, `Signal_Segmentation_Viola-`
28 `tion`, `Signal_Bus_Error`. If the implementation supports any signals besides those
29 defined by this standard, the implementation may also reserve some of those.
- 30 **2.2.2.118 resolution, time:** The minimum time interval that a clock can mea-
31 sure or whose passage a timer can detect. [POSIX.1b {3}]
- 32 **2.2.2.119 root directory:** A directory, associated with a process, that is used in
33 pathname resolution for pathnames that begin with a slash. [POSIX.1 {2}]
- 34 **2.2.2.120 running task:** The task currently being executed by a processor.
- 35 **2.2.2.121 saved set-group-ID:** When the `Saved_IDs_Support` option is imple-
36 mented, an attribute of a process that allows some flexibility in the assignment of
37 the effective group ID attribute, as described for `Set_Group_ID`.
- 38 See §2.3.7 and Section 4.

2.2.2.122 saved set-user-ID: When the `Saved_IDs_Support` 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`.

See §2.3.7 and Section 4.

2.2.2.123 semaphore: A shareable resource that has a non-negative integer value. When the value is zero, there is a (possibly empty) set of tasks awaiting the availability of the semaphore.

2.2.2.124 semaphore decrement operation: An operation that decrements the value of a semaphore, blocking until this is possible. If, prior to the operation, the semaphore's value 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. (See §11.1.7.)

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

If, prior to the operation, there are any tasks 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. (See §11.1.8.)

2.2.2.126 session: A collection of process groups established for job control purposes. [POSIX.1 {2}]

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. [POSIX.1 {2}]

2.2.2.127 session leader: A process that has created a session. (See §4.1.2.)

2.2.2.128 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. [POSIX.1 {2}]

2.2.2.129 shared memory object: An object that represents memory that can be mapped concurrently into the address space of more than one process. These are named regions of storage which 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.

2.2.2.130 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. [POSIX.1 {2}]

2.2.2.131 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 is included. Support for signal queueing is governed by the Realtime Signals option. Not all signals may support queueing.

2.2.2.132 slash: The literal character “/”.

This character is also known as *solidus* in ISO 8859-1 {B4} [POSIX.1 {2}]

2.2.2.133 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 [Ada RM {1} 13.7 (31)], 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.

2.2.2.134 successfully transferred: Data is successfully transferred for a write operation to a regular file when the system ensures that all data written is readable on any 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. Data is successfully transferred for a read operation when an image of the data on the physical storage medium is available to the requesting task.

2.2.2.135 supplementary group ID: An attribute of a process, used in determining file access permissions.

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. [POSIX.1 {2}]

2.2.2.136 synchronized I/O completion: The state of an I/O operation that has either been successfully transferred or diagnosed as unsuccessful. [POSIX.1b {3}]

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

- (1) For reads, 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 writes, 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. [POSIX.1b {3}]

2.2.2.138 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. [POSIX.1b {3}]

2.2.2.139 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. (See §2.2.2.137 and §2.2.2.138.) [POSIX.1b {3}]

2.2.2.140 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. [POSIX.1b {3}]

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

2.2.2.141 system: An implementation of this standard. [POSIX.1 {2}]

2.2.2.142 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. [POSIX.1b {3}]

2.2.2.143 system process: An object, other than a process executing an application, that is defined by the system and has a process ID. [POSIX.1 {2}]

An implementation shall reserve at least one process ID for system processes.

2.2.2.144 system reboot: An implementation defined sequence of events that may result in the loss of transitory data, i.e., data that is not saved in permanent storage. This includes message queues, shared memory, semaphores, and processes. [POSIX.1b {3}]

2.2.2.145 task: An Ada object with a thread of control, defined by [Ada RM {1} 9] 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.

2.2.2.146 terminal [terminal device]: A character special file that obeys the specifications of §7.1. [POSIX.1 {2}]

2.2.2.147 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.1c {4}.

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

2.2.2.149 timer overrun: A condition that occurs each time a timer for which there is already an expiration signal queued to the process expires. [POSIX.1b {3}]

2.2.2.150 user ID: A value identifying a system user.

The user ID is 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.

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

2.2.2.152 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. [POSIX.1 {2}]

2.2.3 Abbreviations.

For the purposes of this standard, the following abbreviations apply, in addition to the abbreviations given for the normative references in §1.2.

2.2.3.1 Ada83: The original Ada language standard, approved by ANSI in 1983 and by ISO/IEC in 1987, item {B?} in Annex A.

2.2.3.2 Ada95: The 1995 Ada language standard, item {1} in §1.2, used in contrast to Ada 83.

2.2.3.3 POSIX.5: IEEE Standard 1003.5-1992, {B3} in Annex A.

2.2.3.4 POSIX.5b: This standard.

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 §2.2.2.97.)

NOTE: The active partition in Ada95 (ISO/IEC 8652: 1995) generalizes the concept of a main program execution in Ada83 (ISO/IEC 8652: 1987).

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 tasks*), if one of the following applies:

- The implementation takes responsibility for the safety of this optimization. That is, it is able to determine that it will not violate any requirements of 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 §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

The Ada RM {1} defines the type `Character` 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`,

1 which “is a character type whose values correspond to the 65536 code positions of the
2 ISO 10646 Basic Multilingual Plane (BMP).”

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

6 **2.3.4 Posix Signals Are Not Interrupts**

7 The application program interfaces for signals defined by this standard are syn-
8 chronous; that is, a task executes an operation that causes it to block until an oc-
9 currence of the signal is available. The preferred interface is via the `Await_Signal`
10 and `Await_Signal_Or_Timeout` operations (see §3.3.15 and §3.3.16). For compat-
11 ibility with POSIX.5 {B3}, a signal can also be attached to an entry of a task, via
12 the Ada interrupt-entry mechanism, if the `Signal Entries` option is supported. (See
13 §3.3.17.)

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

15 **2.3.5 System Call Exception Errors**

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

19 **2.3.6 Extended Security Controls**

20 The access control (see §2.3.7) and privilege mechanisms (see §2.2.2.7) have been
21 defined to allow implementation defined *extended security controls*. These permit an
22 implementation to provide security mechanisms to implement different security poli-
23 cies than are described in this standard. These mechanisms shall be implemented
24 and mapped onto the POSIX interfaces in such a way as to appear to be an extension
25 to the POSIX security system.

26 **2.3.7 File Access Permissions**

27 The standard file access control mechanism uses the file permission set, as described
28 below. This permission set is determined at file creation by the procedures `POSIX_-`
29 `IO.Open_Or_Create`, `POSIX_Files.Create_FIFO`, or `POSIX_Files.Create_Di-`
30 `rectory` and are changed by `POSIX_Files.Change_Permissions`. The permission
31 set can be read by the function `POSIX_File_Status.Permission_Set_Of`.

32 Implementations may provide *additional* or *alternate* file access control mechanisms,
33 or both. An additional access control mechanism shall only further restrict the access
34 permissions defined by the file permission set. An alternate access control mecha-
35 nism shall

- 36 (1) Specify file permission set for the owner class, group class, and other class of the
37 file, corresponding to the access permissions.
- 38 (2) Be enabled only by explicit user action, on a per-file basis by the file owner or a
39 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.

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

- (1) If a process has the appropriate privilege:
 - (a) If read, write, or directory search permission is requested, access is granted.
 - (b) If execute permission is requested, access is granted if execute permission is granted to at least one user by the file permission 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` (see §2.4) is provided for checking a `POSIX_String` to verify if it is a portable filename. The function `Is_Portable_Pathname` (see §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 has been accessed, file data has 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*.

1 If an implementation of such a function marks for update a time-related attribute
2 not specified by this standard, this shall be documented, except that any changes
3 caused by pathname resolution need not be documented. For the other functions
4 in this standard (those that are not explicitly required to read or write file data or
5 change file status, but that in some implementations happen to do so), the effect is
6 unspecified.

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

14 **2.3.11 Pathname Resolution**

15 Pathname resolution is performed for a process to resolve a pathname to a particular
16 file in a file hierarchy. There may be multiple pathnames that resolve to the same
17 file. (See §5.2.1 and §5.2.3.)

18 Each filename in the pathname is located in the directory specified by its predecessor
19 (for example, in the pathname fragment “a/b” file “b” is located in directory “a”).
20 Pathname resolution fails if this cannot be accomplished. If the pathname begins
21 with a slash, the predecessor of the first filename in the pathname is taken to be the
22 root directory of the process (such pathnames are referred to as absolute pathnames).
23 If the pathname does not begin with a slash, the predecessor of the first filename
24 of the pathname is taken to be the current working directory of the process (such
25 pathnames are referred to as relative pathnames). If the pathname begins with
26 two leading slashes, the predecessor of the first filename in the pathname may be
27 interpreted in an implementation defined manner.

28 The interpretation of a pathname component is dependent on `Filename Limit` and
29 whether the `Filename Truncation` option is supported for the path prefix of the compo-
30 nent. If the length in POSIX characters of a pathname component exceeds `File-`
31 `name_Limit` and the `Filename Truncation` option is not supported for the pathname
32 prefix, the implementation shall raise the exception `POSIX_Error` and set the er-
33 ror code `Filename_Too_Long`. Otherwise, the implementation shall ignore all the
34 characters after the initial substring whose length in POSIX characters is the value
35 returned by `Filename_Limit` for the pathname component. (See Section 5.)

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

40 A pathname consisting of a single slash (`/`) resolves to the root directory of the pro-
41 cess. 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. This allows 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:

- 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;
package POSIX is
  -- 2.4.1 Constants and Static Subtypes
  -- 2.4.1.1 Version Identification
  POSIX_Version:      constant := 1995_XX;
  POSIX_Ada_Version:  constant := 1995_YY;
  -- 2.4.1.2 Optional Facilities
  subtype Job_Control_Support is Boolean
    range implementation-defined;
  subtype Saved_IDS_Support is Boolean
    range implementation-defined;
  subtype Change_Owner_Restriction is Boolean
    range implementation-defined;
  subtype Filename_Truncation is Boolean
    range implementation-defined;
  -- 2.4.1.4 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;
  -- 2.4.1.3 System Limits
  Portable_Groups_Maximum: constant Natural := 0;
  subtype Groups_Maxima is Natural
    range implementation-defined;
  Portable_Argument_List_Maximum: constant Natural := 4096;
  subtype Argument_List_Maxima is Natural
    range implementation-defined;

```

```

1     Portable_Child_Processes_Maximum: constant Natural := 6;           -- obsolescent
2     subtype Child_Processes_Maxima is Natural                        -- obsolescent
3         range implementation-defined;                                -- obsolescent
4     Portable_Open_Files_Maximum: constant Natural := 16;           -- obsolescent
5     subtype Open_Files_Maxima is Natural                            -- obsolescent
6         range implementation-defined;                                -- obsolescent
7     Portable_Stream_Maximum: constant Natural := 8;                -- obsolescent
8     subtype Stream_Maxima is Natural                                -- obsolescent
9         range implementation-defined;                                -- obsolescent
10    Portable_Time_Zone_String_Maximum: constant Natural := 3;       -- obsolescent
11    subtype Time_Zone_String_Maxima is Natural                      -- obsolescent
12        range implementation-defined;                                -- obsolescent
13    -- Pathname Variable Values                                       -- obsolescent
14    Portable_Link_Limit_Maximum: constant Natural := 8;            -- obsolescent
15    subtype Link_Limit_Maxima is Natural                            -- obsolescent
16        range implementation-defined;                                -- obsolescent
17    Portable_Input_Line_Limit_Maximum: constant IO_Count := 255;    -- obsolescent
18    subtype Input_Line_Limit_Maxima is IO_Count                    -- obsolescent
19        range implementation-defined;                                -- obsolescent
20    Portable_Input_Queue_Limit_Maximum: constant IO_Count := 255;  -- obsolescent
21    subtype Input_Queue_Limit_Maxima is IO_Count                  -- obsolescent
22        range implementation-defined;                                -- obsolescent
23    Portable_Filename_Limit_Maximum: constant Natural := 14;       -- obsolescent
24    subtype Filename_Limit_Maxima is Natural                      -- obsolescent
25        range implementation-defined;                                -- obsolescent
26    Portable_Pathname_Limit_Maximum: constant Natural := 255;      -- obsolescent
27    subtype Pathname_Limit_Maxima is Natural                      -- obsolescent
28        range implementation-defined;                                -- obsolescent
29    Portable_Pipe_Limit_Maximum: constant IO_Count := 512;         -- obsolescent
30    subtype Pipe_Limit_Maxima is IO_Count                         -- obsolescent
31        range implementation-defined;                                -- obsolescent
32    -- 2.4.1.5 Blocking Behavior Values
33    type Blocking_Behavior is (Tasks, Program, Special);
34    subtype Text_IO_Blocking_Behavior is Blocking_Behavior
35        range implementation-defined;
36    IO_Blocking_Behavior: constant Blocking_Behavior := implementation-defined;
37    File_Lock_Blocking_Behavior:
38        constant Blocking_Behavior := implementation-defined;
39    Wait_For_Child_Blocking_Behavior:
40        constant Blocking_Behavior := implementation-defined;
41    subtype Realtime_Blocking_Behavior is Blocking_Behavior
42        range implementation-defined;
43    -- 2.4.1.6 Signal Masking for Interruptible Operations
44    type Signal_Masking is (No_Signals, RTS_Signals, All_Signals);
45    -- 2.4.2 POSIX Characters
46    type POSIX_Character is
47    (
48        -- ' ' , '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8' , '9' ,
49        -- 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , 'G' , 'H' , 'I' , 'J' , 'K' , 'L' , 'M' ,
50        -- 'N' , 'O' , 'P' , 'Q' , 'R' , 'S' , 'T' , 'U' , 'V' , 'W' , 'X' , 'Y' , 'Z' ,
51        -- 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' ,
52        -- 'n' , 'o' , 'p' , 'q' , 'r' , 's' , 't' , 'u' , 'v' , 'w' , 'x' , 'y' , 'z' ,
53        -- ' , ' - ' _ ' / ' ' ' " ' # ' & ' ' ' ( ' , ' ) '
54        -- ' * ' , ' + ' , ' , ' : ' , ' ; ' , ' < ' , ' = ' , ' > ' , ' | '
55        other characters are implementation defined);
56    -- 2.4.3 POSIX Strings
57    type POSIX_String is array (Positive range <>) of POSIX_Character;
58    function To_POSIX_String (Str: String) return POSIX_String;

```

```

1  function To_POSIX_String (Str: Wide_String) return POSIX_String;
2  function To_String (Str: POSIX_String) return String;
3  function To_Wide_String (Str: POSIX_String) return Wide_String;
4  function To_Stream_Element_Array (Buffer: POSIX_String)
5      return Ada_Streams.Stream_Element_Array;
6  function To_POSIX_String (Buffer: Ada_Streams.Stream_Element_Array)
7      return POSIX_String;
8  subtype Filename is POSIX_String;
9  subtype Pathname is POSIX_String;
10 function Is_Filename (Str: POSIX_String) return Boolean;
11 function Is_Pathname (Str: POSIX_String) return Boolean;
12 function Is_Portable_Filename (Str: POSIX_String) return Boolean;
13 function Is_Portable_Pathname (Str: POSIX_String) return Boolean;
14 -- 2.4.4 String Lists
15 type POSIX_String_List is limited private;
16 Empty_String_List: constant POSIX_String_List;
17 procedure Make_Empty (List: in out POSIX_String_List);
18 procedure Append
19     (List: in out POSIX_String_List;
20      Str: in POSIX_String);
21 generic
22     with procedure Action
23         (Item: in POSIX_String;
24          Quit: in out Boolean);
25 procedure For_Every_Item (List: in POSIX_String_List);
26 function Length (List: POSIX_String_List) return Natural;
27 function Value
28     (List: POSIX_String_List;
29      Index: Positive)
30     return POSIX_String;
31 -- 2.4.5 Option Sets
32 type Option_Set is private;
33 function Empty_Set return Option_Set;
34 function "+" (L, R: Option_Set) return Option_Set;
35 function "-" (L, R: Option_Set) return Option_Set;
36 function "<" (Left, Right: Option_Set) return Boolean;
37 function "<=" (Left, Right: Option_Set) return Boolean;
38 function ">" (Left, Right: Option_Set) return Boolean;
39 function ">=" (Left, Right: Option_Set) return Boolean;
40 Option_1: constant Option_Set;
41 Option_2: constant Option_Set;
42 Option_3: constant Option_Set;
43 Option_4: constant Option_Set;
44 Option_5: constant Option_Set;
45 Option_6: constant Option_Set;
46 Option_7: constant Option_Set;
47 Option_8: constant Option_Set;
48 Option_9: constant Option_Set;
49 Option_10: constant Option_Set;
50 Option_11: constant Option_Set;
51 Option_12: constant Option_Set;
52 Option_13: constant Option_Set;
53 Option_14: constant Option_Set;
54 Option_15: constant Option_Set;
55 Option_16: constant Option_Set;
56 Option_17: constant Option_Set;
57 Option_18: constant Option_Set;

```



```

1      Option_19: constant Option_Set;
2      Option_20: constant Option_Set;
3      Option_21: constant Option_Set;
4      Option_22: constant Option_Set;
5      Option_23: constant Option_Set;
6      Option_24: constant Option_Set;
7      Option_25: constant Option_Set;
8      Option_26: constant Option_Set;
9      Option_27: constant Option_Set;
10     Option_28: constant Option_Set;
11     Option_29: constant Option_Set;
12     Option_30: constant Option_Set;
13     Option_31: constant Option_Set;
14     -- 2.4.6 Error Codes and Exceptions
15     POSIX_Error: exception;
16     type Error_Code is range implementation-defined;
17     function Get_Error_Code return Error_Code;
18     procedure Set_Error_Code (Error: in Error_Code);
19     function Is_POSIX_Error (Error: Error_Code) return Boolean;
20     function Image (Error: Error_Code) return String;
21     No_Error: constant Error_Code:= 0;
22     E2BIG,
23     Argument_List_Too_Long:      constant Error_Code:= impl-def-static-expression;
24     EFAULT,
25     Bad_Address:                 constant Error_Code:= impl-def-static-expression;
26     EBADF,
27     Bad_File_Descriptor:         constant Error_Code:= impl-def-static-expression;
28     EBADMSG,
29     Bad_Message:                constant Error_Code:= impl-def-static-expression;
30     EPIPE,
31     Broken_Pipe:                 constant Error_Code:= impl-def-static-expression;
32     ENOTEMPTY,
33     Directory_Not_Empty:         constant Error_Code:= impl-def-static-expression;
34     ENOEXEC,
35     Exec_Format_Error:           constant Error_Code:= impl-def-static-expression;
36     EEXIST,
37     File_Exists:                 constant Error_Code:= impl-def-static-expression;
38     EFBIG,
39     File_Too_Large:              constant Error_Code:= impl-def-static-expression;
40     ENAMETOOLONG,
41     Filename_Too_Long:           constant Error_Code:= impl-def-static-expression;
42     EXDEV,
43     Improper_Link:               constant Error_Code:= impl-def-static-expression;
44     ENOTTY,
45     Inappropriate_IO_Control_Operation:
46                                     constant Error_Code:= impl-def-static-expression;
47     EIO,
48     Input_Output_Error:          constant Error_Code:= impl-def-static-expression;
49     EINTR,
50     Interrupted_Operation:       constant Error_Code:= impl-def-static-expression;
51     EINVAL,
52     Invalid_Argument:            constant Error_Code:= impl-def-static-expression;
53     ESPIPE,
54     Invalid_Seek:                constant Error_Code:= impl-def-static-expression;
55     EISDIR,
56     Is_A_Directory:              constant Error_Code:= impl-def-static-expression;
57     EMSGSIZE,
58     Message_Too_Long:            constant Error_Code:= impl-def-static-expression;

```

```

1      ECHILD,
2      No_Child_Process:      constant Error_Code:= impl-def-static-expression;
3      ENOLCK,
4      No_Locks_Available:    constant Error_Code:= impl-def-static-expression;
5      ENOSPC,
6      No_Space_Left_On_Device: constant Error_Code:= impl-def-static-expression;
7      ENODEV,
8      No_Such_Operation_On_Device:
9                               constant Error_Code:= impl-def-static-expression;
10     ENXIO,
11     No_Such_Device_Or_Address: constant Error_Code:= impl-def-static-expression;
12     ENOENT,
13     No_Such_File_Or_Directory: constant Error_Code:= impl-def-static-expression;
14     ESRCH,
15     No_Such_Process:        constant Error_Code:= impl-def-static-expression;
16     ENOTDIR,
17     Not_A_Directory:        constant Error_Code:= impl-def-static-expression;
18     ENOMEM,
19     Not_Enough_Space:       constant Error_Code:= impl-def-static-expression;
20     ECANCELED,
21     Operation_Canceled:     constant Error_Code:= impl-def-static-expression;
22     EINPROGRESS,
23     Operation_In_Progress:   constant Error_Code:= impl-def-static-expression;
24     ENOSYS,
25     Operation_Not_Implemented: constant Error_Code:= impl-def-static-expression;
26     EPERM,
27     Operation_Not_Permitted: constant Error_Code:= impl-def-static-expression;
28     ENOTSUP,
29     Operation_Not_Supported: constant Error_Code:= impl-def-static-expression;
30     EACCES,
31     Permission_Denied:      constant Error_Code:= impl-def-static-expression;
32     EROFS,
33     Read_Only_File_System:   constant Error_Code:= impl-def-static-expression;
34     EBUSY,
35     Resource_Busy:          constant Error_Code:= impl-def-static-expression;
36     EDEADLK,
37     Resource_Deadlock_Avoided: constant Error_Code:= impl-def-static-expression;
38     EAGAIN,
39     Resource_Temporarily_Unavailable:
40                               constant Error_Code:= impl-def-static-expression;
41     ETIMEDOUT,
42     Timed_Out:              constant Error_Code:= impl-def-static-expression;
43     EMLINK,
44     Too_Many_Links:         constant Error_Code:= impl-def-static-expression;
45     EMFILE,
46     Too_Many_Open_Files:    constant Error_Code:= impl-def-static-expression;
47     ENFILE,
48     Too_Many_Open_Files_In_System:
49                               constant Error_Code:= impl-def-static-expression;
50     -- 2.4.7 System Identification
51     function System_Name return POSIX_String;
52     function Node_Name return POSIX_String;
53     function Release return POSIX_String;
54     function Version return POSIX_String;
55     function Machine return POSIX_String;
56     -- 2.4.8 Time Types
57     type Seconds is range implementation-defined;
58     -- must include at least  $-(2^{31} - 1) .. (2^{31} - 1)$ 

```



```

1      type Nanoseconds_Base is range implementation-defined;
2      -- must include at least  $-(2^{31} - 1) .. (2^{31} - 1)$ 
3      subtype Nanoseconds is Nanoseconds_Base range 0..(10**9)-1;
4      type Timespec is private;
5      function Get_Seconds (Time: Timespec) return Seconds;
6      procedure Set_Seconds
7          (Time: in out Timespec;
8           S: in Seconds);
9      function Get_Nanoseconds (Time: Timespec) return Nanoseconds;
10     procedure Set_Nanoseconds
11         (Time: in out Timespec;
12          NS: in Nanoseconds);
13     procedure Split
14         (Time: in Timespec;
15          S: out Seconds;
16          NS: out Nanoseconds);
17     function To_Timespec
18         (S: Seconds;
19          NS: Nanoseconds)
20         return Timespec;
21     function "+" (Left, Right: Timespec) return Timespec;
22     function "+" (Left: Timespec; Right: Nanoseconds) return Timespec;
23     function "-" (Right: Timespec) return Timespec;
24     function "-" (Left, Right: Timespec) return Timespec;
25     function "-" (Left: Timespec; Right: Nanoseconds) return Timespec;
26     function "*" (Left: Timespec; Right: Integer) return Timespec;
27     function "*" (Left: Integer; Right: Timespec) return Timespec;
28     function "/" (Left: Timespec; Right: Integer) return Timespec;
29     function "/" (Left, Right: Timespec) return Integer;
30     function "<" (Left, Right: Timespec) return Boolean;
31     function "<=" (Left, Right: Timespec) return Boolean;
32     function ">" (Left, Right: Timespec) return Boolean;
33     function ">=" (Left, Right: Timespec) return Boolean;
34     function To_Duration (Time: Timespec) return Duration;
35     function To_Timespec (D: Duration) return Timespec;
36 private
37     implementation-defined
38 end POSIX;

```

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

41 2.4.1 Constants and Static Subtypes

42 2.4.1.1 Version Identification

43 2.4.1.1.1 Synopsis

```

44     POSIX_Version:      constant:= 1995_XX;
45     POSIX_Ada_Version:  constant:= 1995_YY;

```

46 2.4.1.1.2 Description

47 The constant POSIX_Version shall provide the year and month of the POSIX stan-
48 dard on which this standard is based. The constant POSIX_Ada_Version shall
49 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 {B3}. 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 System Limits

The implementation limit constants and subtypes declared in POSIX are obsolescent. They are only provided for compatibility with POSIX.5 {B3}. 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

2.4.1.4 Bytes and I/O Counts

2.4.1.4.1 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;

```

2.4.1.4.2 Description

The constant `Byte_Size` is the size, in bits, of one byte of data. (See §2.2.2.17.) 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 “byte” is used here in the same sense as in POSIX.1 {2} and POSIX.1b {3}.

NOTE: A byte is likely to be eight bits, but this 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.

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;  
subtype Realtime_Blocking_Behavior is Blocking_Behavior  
    range implementation-defined;
```

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 so cannot prevent any other task from executing.

- 1 **Program**
2 The blocking behavior is per process. If a task executes a blocking operation
3 and the operation blocks, the task continues to hold all the execution re-
4 sources, and so prevents all the tasks in the POSIX process from executing.
- 5 **Special**
6 The blocking behavior is neither per task nor per process. If a task executes
7 a blocking operation and the operation blocks, the task may continue to hold
8 some execution resource which may prevent some, but not necessarily all, of
9 the other tasks in the process from executing. The specific conditions under
10 which tasks other than the calling task are prevented from executing are
11 implementation defined.
- 12 NOTE: With any form of blocking behavior other than per task, there is danger of ac-
13 cidental deadlock, in which a blocked task is waiting for an action by another task, but
14 is implicitly preventing the other task from executing by holding needed execution re-
15 sources.
- 16 If the underlying system supports multithreaded processes as defined in POSIX.1c-
17 {4}, the blocking behavior of all operations defined by this standard shall be Tasks.
- 18 Otherwise, different classes of POSIX operations may have different blocking behav-
19 iors, subject to the following restrictions:
- 20 — Per-task blocking behavior is always required for:
21 — All potentially blocking operations defined in the package `POSIX_Mutexes`.
22 — All potentially blocking operations defined in the package `POSIX_Condition_Variables`.
23
24 — For all other operations defined by this standard, blocking behavior shall also
25 be per task, except where this is not feasible using the facilities of the underly-
26 ing system. The conformance document for each implementation shall list any
27 exceptions to this rule, and the justification for the exception.
- 28 The constants and subtypes below shall indicate the actual blocking behavior of the
29 particular implementation, for those operations where exceptions to the per task
30 blocking requirement are permitted. They can be used by an application to tailor its
31 execution, based on the blocking behavior of specific operations.
- 32 **Text_IO_Blocking_Behavior**
33 If the range of the subtype is `Tasks .. Tasks`, the blocking behavior of Ada
34 I/O operations shall be per task. If the range of the subtype is `Program ..`
35 **Program**, the blocking behavior of Ada I/O operations shall be per process.
36 If the range of the subtype is `Tasks .. Program`, the blocking behavior of
37 Ada I/O operations shall be selectable via the `FORM` parameter. (See §8.1.)
38 If the range of the subtype includes `Special`, the blocking behavior of Ada
39 I/O operations is implementation defined.
- 40 **IO_Blocking_Behavior**
41 Specifies the blocking behavior of POSIX I/O operations that may block. (See
42 §8.1, §6.1 and §6.3.)
- 43 **File_Lock_Blocking_Behavior**
44 Specifies the blocking behavior of the file locking operations. (See §6.2.)

1 **Wait_For_Child_Blocking_Behavior**
 2 Specifies the blocking behavior of the wait for child termination operations.
 3 (See §3.1.)

4 **Realtime_Blocking_Behavior**
 5 Specifies the blocking behavior of the following “realtime” operations defined
 6 by this standard:
 7 — POSIX_Semaphores.Wait
 8 — POSIX_Signals.Await_Signal
 9 — POSIX_Signals.Await_Signal_Or_Timeout
 10 — POSIX_Shared_Memory_Objects.Open_Shared_Memory
 11 — POSIX_Shared_Memory_Objects.Open_Or_Create_Shared_Memory
 12 — POSIX_Generic_Shared_Memory.Open_And_Map_Shared_Memory
 13 — POSIX_Generic_Shared_Memory.Open_Or_Create_And_Map_
 14 Shared_Memory
 15 — POSIX_Message_Queues.Open
 16 — POSIX_Message_Queues.Open_Or_Create
 17 — POSIX_Message_Queues.Send
 18 — POSIX_Message_Queues.Receive
 19 — Generic_Message_Queues.Send
 20 — Generic_Message_Queues.Receive
 21 — POSIX_Asynchronous_IO.List_IO_Wait
 22 — POSIX_Asynchronous_IO.Await_IO_Or_Timeout
 23 — POSIX_Asynchronous_IO.Await_IO

24 NOTE: Realtime_Blocking_Behavior is a range, rather than a simple constant. This
 25 allows for the possibility that an implementation may have different blocking behavior
 26 for different operations in the list above. However, the recommended implementation is
 27 that the blocking behavior of all these operations be per-task, in which case the range
 28 will be Tasks..Tasks.

29 NOTE: It is recommended that the implementation defined expressions and ranges for
 30 the constants and subtypes described in this subclause be static, so as to permit compile-
 31 time optimization.

32 **2.4.1.6 Signal Masking for Interruptible Operations**

33 **2.4.1.6.1 Synopsis**

34 **type** Signal_Masking **is** (No_Signals, RTS_Signals, All_Signals);

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

37 **No_Signals**

38 No additional signals are masked during the POSIX operation.

39 **RTS_Signals**

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

42 **All_Signals**

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

2.4.2 POSIX Characters

2.4.2.1 Synopsis

```

3      type POSIX_Character is
4      (
5          -- ' ' , '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8' , '9' ,
6          -- 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , 'G' , 'H' , 'I' , 'J' , 'K' , 'L' , 'M' ,
7          -- 'N' , 'O' , 'P' , 'Q' , 'R' , 'S' , 'T' , 'U' , 'V' , 'W' , 'X' , 'Y' , 'Z' ,
8          -- 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' , 'l' , 'm' ,
9          -- 'n' , 'o' , 'p' , 'q' , 'r' , 's' , 't' , 'u' , 'v' , 'w' , 'x' , 'y' , 'z' ,
10         -- '.' , '-' , '_' , '/' , ':' , ';' , '<' , '=' , '>' , '|' ,
11         -- '*' , '+' , ',' , '!' , '~' , '^' , '&' , '&' , '&' , '&' , '&' , '&' , '&' , '&' ,
12         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_Characters` or concatenations of `POSIX.POSIX_Characters`.

The implementation shall provide correspondences between the standard Ada type `Character` 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 Ada RM {1}.
- (2) A conforming application shall make no assumption as to 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`, such that characters that have 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 (_).

- 1 — Format effectors: horizontal tabulation, vertical tabulation, carriage re-
- 2 turn, line feed, form feed.
- 3 (4) All remaining characters of the type `Standard.Character` shall be mapped
- 4 to some subset of the remaining characters within `POSIX_Character`. This
- 5 standard does not require invertibility of this mapping.
- 6 (5) Any remaining characters of the type `POSIX_Character` would have no coun-
- 7 terpart in `Standard.Character`.

8 2.4.3 **POSIX Strings**

9 2.4.3.1 **Synopsis**

```

10       type POSIX_String is array (Positive range <>) of POSIX_Character;
11       function To_POSIX_String (Str: String) return POSIX_String;

12       function To_POSIX_String (Str: Wide_String) return POSIX_String;
13       function To_String (Str: POSIX_String) return String;
14       function To_Wide_String (Str: POSIX_String) return Wide_String;
15       function To_Stream_Element_Array (Buffer: POSIX_String)
16       return Ada_Streams.Stream_Element_Array;
17       function To_POSIX_String (Buffer: Ada_Streams.Stream_Element_Array)
18       return POSIX_String;
19       subtype Filename is POSIX_String;
20       subtype Pathname is POSIX_String;
21       function Is_Filename (Str: POSIX_String) return Boolean;
22       function Is_Pathname (Str: POSIX_String) return Boolean;
23       function Is_Portable_Filename (Str: POSIX_String) return Boolean;
24       function Is_Portable_Pathname (Str: POSIX_String) return Boolean;

```

25 2.4.3.2 **Description**

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

30 The function `To_POSIX_String` taking arguments of type `Stream_Element_`
31 `Array` shall convert an `Ada_Streams.Stream_Element_Array` value to a `POSIX_`
32 `String` value. The function `To_Stream_Element_Array` shall convert a `POSIX_`
33 `String` value to a `Ada_Streams.Stream_Element_Array` value. These operations
34 shall obey the semantics of instantiations of `Unchecked_Conversion`, as defined in
35 [Ada RM {1} 13.9 J.1].

36 The functions `To_POSIX_String` taking arguments of types `String` and `Wide_`
37 `String`, and the functions `To_String`, and `To_Wide_String` are used to convert
38 between the standard Ada string types and the `POSIX_String` type. All characters
39 in the core subset, defined above, shall be *convertible* and their conversion shall be
40 *invertible*. The results of converting the other characters in the standard Ada char-
41 acter set is implementation defined. A character in one character set may translate
42 into a sequence of characters in the other character set. An implementation may
43 convert the string sequences back into the single character when doing the inverse
44 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` (see §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`, 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_String`s. The value `Empty_String_List` represents a `POSIX_String_List` that contains no `POSIX_String`s 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 `POSIX_String` values in the `POSIX_String_List`. This includes any null strings.

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`.

1 2.4.5 Option Sets

2 2.4.5.1 Synopsis

```

3     type Option_Set is private;
4     function Empty_Set return Option_Set;
5     function "+" (L, R: Option_Set) return Option_Set;
6     function "-" (L, R: Option_Set) return Option_Set;
7     function "<" (Left, Right: Option_Set) return Boolean;
8     function "<=" (Left, Right: Option_Set) return Boolean;
9     function ">" (Left, Right: Option_Set) return Boolean;
10    function ">=" (Left, Right: Option_Set) return Boolean;
11    Option_1: constant Option_Set;
12    Option_2: constant Option_Set;
13    Option_3: constant Option_Set;
14    Option_4: constant Option_Set;
15    Option_5: constant Option_Set;
16    Option_6: constant Option_Set;
17    Option_7: constant Option_Set;
18    Option_8: constant Option_Set;
19    Option_9: constant Option_Set;
20    Option_10: constant Option_Set;
21    Option_11: constant Option_Set;
22    Option_12: constant Option_Set;
23    Option_13: constant Option_Set;
24    Option_14: constant Option_Set;
25    Option_15: constant Option_Set;
26    Option_16: constant Option_Set;
27    Option_17: constant Option_Set;
28    Option_18: constant Option_Set;
29    Option_19: constant Option_Set;
30    Option_20: constant Option_Set;
31    Option_21: constant Option_Set;
32    Option_22: constant Option_Set;
33    Option_23: constant Option_Set;
34    Option_24: constant Option_Set;
35    Option_25: constant Option_Set;
36    Option_26: constant Option_Set;
37    Option_27: constant Option_Set;
38    Option_28: constant Option_Set;
39    Option_29: constant Option_Set;
40    Option_30: constant Option_Set;
41    Option_31: constant Option_Set;

```

42 2.4.5.2 Description

43 The type `Option_Set` shall be used to represent sets of options. All objects of this
44 type shall have as their initial value the value returned by `Empty_Set`. The function
45 `Empty_Set` shall return a value of type `Option_Set` with no options in it. The
46 operation `"+"` shall return an option set containing exactly the set of options that
47 are members of one or both of the two operand sets, i.e., the “union” operation on
48 option sets. The binary operation `"-"` shall return an option set where options set in
49 the right-hand operand set are not members of the result set, and all other options
50 in the left-hand operand set are members of the result set.

51 An implementation is permitted to add declarations to the `POSIX` package for addi-
52 tional 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 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;

```

```

1      EXDEV,
2      Improper_Link:          constant Error_Code:= impl-def-static-expression;
3      ENOTTY,
4      Inappropriate_IO_Control_Operation:
5                               constant Error_Code:= impl-def-static-expression;
6      EIO,
7      Input_Output_Error:     constant Error_Code:= impl-def-static-expression;
8      EINTR,
9      Interrupted_Operation:  constant Error_Code:= impl-def-static-expression;
10     EINVAL,
11     Invalid_Argument:       constant Error_Code:= impl-def-static-expression;
12     ESPIPE,
13     Invalid_Seek:           constant Error_Code:= impl-def-static-expression;
14     EISDIR,
15     Is_A_Directory:         constant Error_Code:= impl-def-static-expression;
16     EMSGSIZE,
17     Message_Too_Long:       constant Error_Code:= impl-def-static-expression;
18     ECHILD,
19     No_Child_Process:        constant Error_Code:= impl-def-static-expression;
20     ENOLCK,
21     No_Locks_Available:     constant Error_Code:= impl-def-static-expression;
22     ENOSPC,
23     No_Space_Left_On_Device: constant Error_Code:= impl-def-static-expression;
24     ENODEV,
25     No_Such_Operation_On_Device:
26                               constant Error_Code:= impl-def-static-expression;
27     ENXIO,
28     No_Such_Device_Or_Address: constant Error_Code:= impl-def-static-expression;
29     ENOENT,
30     No_Such_File_Or_Directory: constant Error_Code:= impl-def-static-expression;
31     ESRCH,
32     No_Such_Process:         constant Error_Code:= impl-def-static-expression;
33     ENOTDIR,
34     Not_A_Directory:         constant Error_Code:= impl-def-static-expression;
35     ENOMEM,
36     Not_Enough_Space:        constant Error_Code:= impl-def-static-expression;
37     ECANCELED,
38     Operation_Canceled:      constant Error_Code:= impl-def-static-expression;
39     EINPROGRESS,
40     Operation_In_Progress:   constant Error_Code:= impl-def-static-expression;
41     ENOSYS,
42     Operation_Not_Implemented: constant Error_Code:= impl-def-static-expression;
43     EPERM,
44     Operation_Not_Permitted: constant Error_Code:= impl-def-static-expression;
45     ENOTSUP,
46     Operation_Not_Supported: constant Error_Code:= impl-def-static-expression;
47     EACCES,
48     Permission_Denied:       constant Error_Code:= impl-def-static-expression;
49     EROFS,
50     Read_Only_File_System:   constant Error_Code:= impl-def-static-expression;
51     EBUSY,
52     Resource_Busy:           constant Error_Code:= impl-def-static-expression;
53     EDEADLK,
54     Resource_Deadlock_Avoided: constant Error_Code:= impl-def-static-expression;
55     EAGAIN,
56     Resource_Temporarily_Unavailable:
57                               constant Error_Code:= impl-def-static-expression;
58     ETIMEDOUT,
59     Timed_Out:               constant Error_Code:= impl-def-static-expression;
60     EMLINK,
61     Too_Many_Links:          constant Error_Code:= impl-def-static-expression;
62     EMFILE,
63     Too_Many_Open_Files:     constant Error_Code:= impl-def-static-expression;
64     ENFILE,
65     Too_Many_Open_Files_In_System:
66                               constant Error_Code:= impl-def-static-expression;

```

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 for the calling task to the value specified.

The function `Is_POSIX_Error` shall return `True` if the error code specified 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 identifiers 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 supported by the implementation.

If the implementation supports package `Ada.Exceptions`, as defined in [Ada RM-11.4.1], and if the function `Ada.Exceptions.Exception_Message` is called for an occurrence of `POSIX_Error` raised with a given error code 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.

The following descriptions identify the possible *errors* that can cause the exception `POSIX_Error` to be raised, and the Error Code 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.

Argument_List_Too_Long

The sum of the number of `POSIX_Characters` 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, 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.

Directory_Not_Empty

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

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.

- 1 **Improper_Link**
2 A link to a file on another file system was attempted.
- 3 **Inappropriate_IO_Control_Operation**
4 A control function was attempted for a file or special file for which the oper-
5 ation was inappropriate.
- 6 **Input_Output_Error**
7 Some physical input or output error occurred. This error may be reported on
8 a subsequent operation on the same file descriptor. Any other error-causing
9 operation on the same file descriptor may cause the **Input_Output_Error**
10 to be lost.
- 11 **Interrupted_Operation**
12 An asynchronous signal was caught by the task during the execution of
13 an interruptible function. If the signal handler returns normally, then the
14 interrupted function call may raise the **POSIX_Error** exception with this
15 error code.
- 16 **Invalid_Argument**
17 Some invalid argument was supplied.
- 18 **Invalid_Seek**
19 A Seek operation was issued on a pipe or FIFO.
- 20 **Is_A_Directory**
21 An attempt was made to open a directory with write mode specified.
- 22 **Message_Too_Long**
23 Inappropriate message buffer length.
- 24 **No_Child_Process**
25 A call to a **Wait_For_Child_Process** procedure was executed by a pro-
26 cess that had no existing child processes or whose status had not yet been
27 reported.
- 28 **No_Locks_Available**
29 A system-imposed limit on the number of simultaneous file and record locks
30 was reached, and no more were available at that time.
- 31 **No_Space_Left_On_Device**
32 During a write on a regular file, or when extending a directory, there was no
33 free space left on the device.
- 34 **No_Such_Operation_On_Device**
35 An attempt was made to apply an inappropriate operation on a device; for
36 example, trying to read a write-only device such as a printer.
- 37 **No_Such_Device_Or_Address**
38 Input or output on a special file referred to a device that did not exist or
39 made a request beyond the limits of the device. This error may also occur
40 when, for example, a tape drive is not online or a removable disk is not
41 loaded on a drive.

- 1 **No_Such_File_Or_Directory**
2 A component of a specified pathname did not exist, or the pathname was an
3 empty string.
- 4 **No_Such_Process**
5 No process could be found corresponding to that specified by the given pro-
6 cess ID.
- 7 **Not_A_Directory**
8 A component of the specified pathname existed, but it was not a directory
9 when a directory was expected.
- 10 **Not_Enough_Space**
11 The new process image required more memory than was allowed by the
12 hardware or by system-imposed memory management constraints.
- 13 **Operation_Canceled**
14 The associated asynchronous operation was canceled before completion.
- 15 **Operation_In_Progress**
16 An asynchronous operation has not yet completed.
- 17 **Operation_Not_Implemented**
18 An attempt was made to use a subprogram that is not available in this im-
19 plementation.
- 20 **Operation_Not_Permitted**
21 An attempt was made to perform an operation limited to processes with
22 appropriate privileges or to the owner of a file or other resource.
- 23 **Operation_Not_Supported**
24 The implementation does not support this feature of the standard.
- 25 **Permission_Denied**
26 An attempt was made to access a file in a way forbidden by its file access
27 permissions.
- 28 **Read_Only_File_System**
29 An attempt was made to modify a file or directory on a file system that was
30 read-only at that time.
- 31 **Resource_Busy**
32 An attempt was made to use a system resource that was not available at the
33 time because it was being used by a process in a manner that would have
34 conflicted with the request being made by this process.
- 35 **Resource_Deadlock_Avoided**
36 An attempt was made to lock a system resource that would have resulted in
37 a deadlock situation.
- 38 **Resource_Temporarily_Unavailable**
39 This is a temporary condition, and later calls to the same routine may com-
40 plete normally.

1 **Timed_Out**

2 The operation timed out, without completing normally.

3 NOTE: Most operations with time-outs return error code `Resource_Tem-`
4 porarily_Unavailable.

5 **Too_Many_Links**

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

9 **Too_Many_Open_Files**

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

12 **Too_Many_Open_Files_In_System**

13 Too many files are currently open in the system. The system reached its pre-
14 defined limit for simultaneously open files and temporarily could not accept
15 requests to open another one.

16 **2.4.6.3 Error Handling**

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

19 **2.4.7 System Identification**

20 **2.4.7.1 Synopsis**

21 **function** `System_Name` **return** `POSIX_String`;
22 **function** `Node_Name` **return** `POSIX_String`;
23 **function** `Release` **return** `POSIX_String`;
24 **function** `Version` **return** `POSIX_String`;
25 **function** `Machine` **return** `POSIX_String`;

26 **2.4.7.2 Description**

27 The function `System_Name` shall return the name of this implementation of the op-
28 erating system. The function `Node_Name` shall return the name of this node within
29 an implementation-specified communications network. The function `Release` shall
30 return the current release level of this implementation. The function `Version` shall
31 return the current version level of this release. The function `Machine` shall return
32 the name of the hardware type on which the system is running.

33 **2.4.7.3 Error Handling**

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

2.4.8 Time Types

2.4.8.1 Synopsis

```

3  type Seconds is range implementation-defined;
4  -- must include at least  $-(2^{31} - 1) .. (2^{31} - 1)$ 
5  type Nanoseconds_Base is range implementation-defined;
6  -- must include at least  $-(2^{31} - 1) .. (2^{31} - 1)$ 
7  subtype Nanoseconds is Nanoseconds_Base range 0..(10**9)-1;
8  type Timespec is private;
9  function Get_Seconds (Time: Timespec) return Seconds;
10 procedure Set_Seconds
11     (Time: in out Timespec;
12      S: in Seconds);
13 function Get_Nanoseconds (Time: Timespec) return Nanoseconds;
14 procedure Set_Nanoseconds
15     (Time: in out Timespec;
16      NS: in Nanoseconds);
17 procedure Split
18     (Time: in Timespec;
19      S: out Seconds;
20      NS: out Nanoseconds);
21 function To_Timespec
22     (S: Seconds;
23      NS: Nanoseconds)
24     return Timespec;
25 function "+" (Left, Right: Timespec) return Timespec;
26 function "+" (Left: Timespec; Right: Nanoseconds) return Timespec;
27 function "-" (Right: Timespec) return Timespec;
28 function "-" (Left, Right: Timespec) return Timespec;
29 function "-" (Left: Timespec; Right: Nanoseconds) return Timespec;
30 function "*" (Left: Timespec; Right: Integer) return Timespec;
31 function "*" (Left: Integer; Right: Timespec) return Timespec;
32 function "/" (Left: Timespec; Right: Integer) return Timespec;
33 function "/" (Left, Right: Timespec) return Integer;
34 function "<" (Left, Right: Timespec) return Boolean;
35 function "<=" (Left, Right: Timespec) return Boolean;
36 function ">" (Left, Right: Timespec) return Boolean;
37 function ">=" (Left, Right: Timespec) return Boolean;
38 function To_Duration (Time: Timespec) return Duration;
39 function To_Timespec (D: Duration) return Timespec;

```

2.4.8.2 Description

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

Nanoseconds is a subtype that is interpreted as a non-negative 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 arithmetic must be performed 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 $-(2^{31}-1) \dots (2^{31}-1)$ 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 in such a way 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 $0 \dots 10^9 - 1$. 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`.

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, that is:

- If $T = \text{To\_Timespec}(S\_1, NS\_1)$, 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` must 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 which 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 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 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 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 Synchronized_IO_Support      is Boolean range implementation-defined;
  subtype Timers_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, so as to take advantage of optional features that are supported by a particular implementation.

The names and general descriptions of the optional facilities are listed below. 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.

- 1 Asynchronous I/O
- 2 The implementation supports asynchronous input and output operations.
- 3 This option is pathname-specific.
- 4 Change Owner Restriction
- 5 The procedure `Change_Owner_And_Group` is restricted to changing the
- 6 group ID of a file only to the effective group ID of the process or to one of
- 7 its supplementary group IDs, and changing the owner ID is restricted to
- 8 processes with the use privilege. If this option is not supported, no such
- 9 restriction applies. This option is pathname-specific.
- 10 Filename Truncation
- 11 Filenames are truncated to the maximum allowed length in POSIX char-
- 12 acters, `Filename Maximum`. If this option is not supported, and a filename is
- 13 longer than allowed, an error shall result, as defined by the subprograms
- 14 that use filenames and pathnames. This option is pathname-specific.
- 15 File Synchronization
- 16 The implementation supports file synchronization operations.
- 17 Job Control
- 18 The implementation supports job control signals.
- 19 Memory Mapped Files
- 20 The implementation supports the mapping of files to the address space of a
- 21 process.
- 22 Memory Locking
- 23 The implementation supports locking all or part of a process's address space
- 24 to be continually resident in memory.
- 25 Memory Range Locking
- 26 The implementation supports locking regions of a process's address space so
- 27 that they are continually resident in memory. If this option is supported,
- 28 the Memory Locking option shall be supported also.
- 29 Memory Protection
- 30 The implementation supports protecting regions of a process's address space
- 31 from read, write, or execute access. If this option is supported, then at least
- 32 one of the Memory Mapped Files option or the Shared Memory Objects option
- 33 shall be supported also.
- 34 Message Queues
- 35 The implementation supports message queues.
- 36 Mutex Priority Ceiling
- 37 The implementation temporarily raises the priority of a task that holds a
- 38 mutex to the specified ceiling priority of the mutex.
- 39 NOTE: The utility of this option depends on support for the `Mutexes` option.
- 40 Mutex Priority Inheritance
- 41 The implementation temporarily raises the priority of a task that holds a
- 42 mutex to the highest priority of the tasks that are blocked waiting for the
- 43 mutex.
- 44 NOTE: The utility of this option depends on support for the `Mutexes` option.

- 1 **Mutexes**
- 2 The implementation supports mutex and condition variable synchronization
- 3 objects.
- 4 **Prioritized I/O**
- 5 The implementation supports prioritized input and output. This option is
- 6 pathname-specific.
- 7 NOTE: The utility of this option depends on support for the Asynchronous I/O
- 8 option and for the Priority Process Scheduling option.
- 9 **Priority Process Scheduling**
- 10 The implementation supports the process scheduling interfaces defined by
- 11 this standard.
- 12 **Priority Task Scheduling**
- 13 The implementation supports the extended task scheduling interfaces de-
- 14 fined by this standard.
- 15 NOTE: A prerequisite for support of this option is support for the priority model
- 16 defined in [Ada RM {1} D.1] and the pragmas and package interfaces defined
- 17 in [Ada RM {1} D.2-D.5]. (See §13.3.)
- 18 **Process Shared**
- 19 The implementation supports sharing of mutexes and condition variables
- 20 between processes, in a shared memory object or memory-mapped file.
- 21 NOTE: The utility of this option depends on support for the Mutexes option.
- 22 **Realtime Signals**
- 23 The implementation supports the realtime signals, for which queueing, sig-
- 24 nal information, and notification in priority order are defined.
- 25 **Saved IDs**
- 26 The implementation supports a *saved set-user-ID* and a *saved set-group-ID*
- 27 for each process. If this option is not supported, there is neither a saved
- 28 set-user-ID nor a saved set-group-ID.
- 29 **Semaphores**
- 30 The implementation supports counting semaphores.
- 31 **Shared Memory Objects**
- 32 The implementation supports shared memory objects.
- 33 **Signal Entries**
- 34 The implementation supports binding task entries to signals. (See §3.3.17.)
- 35 The range of the static subtype for this option shall be either `True..True` or
- 36 `False..False`.
- 37 **Synchronized I/O**
- 38 The implementation supports I/O synchronization, which permits a process
- 39 to force I/O operations to synchronized I/O data integrity completion or syn-
- 40 chronized I/O file integrity completion. If this option is supported, then the
- 41 File Synchronization option shall be supported also. This option is pathname-
- 42 specific.
- 43 **Timers**
- 44 The implementation supports clocks and timers.

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.

NOTE: As indicated by the phrase *static subtype* in the heading of this item and the corresponding table, these subtypes are all required to 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.
- (4) If the range is `True..True`, the option is always supported. The corresponding function shall always return `True`.

NOTE: There is one kind of implementation option that 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` (see §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 which can be used to discover the capacity limits that are imposed by an implementation of this standard.

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
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
Prioritized_IO_Support	Prioritized I/O
Priority_Process_Scheduling_Support	Priority Process Scheduling
Priority_Task_Scheduling_Support	Priority Task Scheduling
Shared_Memory_Objects_Support	Shared Memory Objects
Realtime_Signals_Support	Realtime Signals
Saved_IDs_Support	Saved IDs
Semaphores_Support	Semaphores
Process_Shared_Support	Process Shared
Signal_Entries_Support	Signal Entries
Synchronized_IO_Support	Synchronized I/O
Timers_Support	Timers

```

with POSIX;
package POSIX_Limits is
  -- Portable System Limits
  Portable_Argument_List_Maximum: Natural
    renames POSIX.Portable_Argument_List_Maximum;
  Portable_Asynchrounous_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_Open_Message_Queues_Maximum: constant Natural:= 8;

```

```

1      Portable_Pathname_Maximum: Natural
2      renames POSIX.Portable_Pathname_Limit_Maximum;
3      Portable_Pipe_Length_Maximum: POSIX.IO_Count
4      renames POSIX.Portable_Pipe_Limit_Maximum;
5      Portable_Queued_Signals_Maximum: constant Natural:= 32;
6      Portable_Realtime_Signals_Maximum: constant Natural:= 8;
7      Portable_Semaphores_Maximum: constant Natural:= 256;
8      Portable_Semaphores_Value_Maximum: constant Natural:= 32_767;
9      Portable_Streams_Maximum: Natural
10     renames POSIX.Portable_Stream_Maximum;
11     Portable_Timer_Overruns_Maximum: constant Natural:= 32;
12     Portable_Timers_Maximum: constant Natural:= 32;
13     Portable_Time_Zone_String_Maximum: Natural
14     renames POSIX.Portable_Time_Zone_String_Maximum;
15     -- Configurable Limits
16     subtype Argument_List_Maxima is POSIX.Argument_List_Maxima;
17     subtype Asynchronous_IO_Maxima is Natural range implementation-defined;
18     subtype Asynchronous_IO_Priority_Delta_Maxima is Natural
19     range implementation-defined;
20     subtype Child_Processes_Maxima is POSIX.Child_Processes_Maxima;
21     subtype Filename_Maxima is POSIX.Filename_Limit_Maxima;
22     subtype Groups_Maxima is POSIX.Groups_Maxima;
23     subtype Input_Line_Maxima is POSIX.Input_Line_Limit_Maxima;
24     subtype Input_Queue_Maxima is POSIX.Input_Queue_Limit_Maxima;
25     subtype Links_Maxima is POSIX.Link_Limit_Maxima;
26     subtype List_IO_Maxima is Natural range implementation-defined;
27     subtype Open_Message_Queues_Maxima is Natural range implementation-defined;
28     subtype Message_Priority_Maxima is Natural range implementation-defined;
29     subtype Open_Files_Maxima is POSIX.Open_Files_Maxima;
30     subtype Page_Size_Range is Natural range 1..implementation-defined;
31     subtype Pathname_Maxima is POSIX.Pathname_Limit_Maxima;
32     subtype Pipe_Length_Maxima is POSIX.Pipe_Limit_Maxima;
33     subtype Queued_Signals_Maxima is Natural range implementation-defined;
34     subtype Realtime_Signals_Maxima is Natural range implementation-defined;
35     subtype Semaphores_Maxima is Natural range implementation-defined;
36     subtype Semaphores_Value_Maxima is Natural range implementation-defined;
37     subtype Streams_Maxima is POSIX.Stream_Maxima;
38     subtype Timer_Overruns_Maxima is Natural range implementation-defined;
39     subtype Timers_Maxima is Natural
40     range implementation-defined;
41     subtype Time_Zone_String_Maxima is POSIX.Time_Zone_String_Maxima;
42 end POSIX_Limits;

```

43 2.6.1 Implementation Limits

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

48 The names and general descriptions of the limits are listed below. These names are
49 used in the rest of this standard to specify the functionality that depends on each
50 limit. Generally, the value of a limit is system-wide. For some limits, related to
51 files, the value is *pathname-specific*. This means that the value of the limit may be
52 different for some files than for others, depending on the pathname of the file, and
53 the time that the file is accessed.

- 1 Argument List Maximum
- 2 The maximum total length of the argument list, environment data, and
- 3 any associated overhead (in bytes) that can be passed to a child when it is
- 4 started by the `POSIX_Process_Primitives` procedures `Start_Process`
- 5 and `Start_Process_Search`, and the `POSIX_Unsafe_Process_Primitives`
- 6 procedures `Exec` and `Exec_Search`.
- 7 Asynchronous I/O Maximum
- 8 The maximum number of outstanding asynchronous I/O operations.
- 9 Asynchronous I/O Priority Delta Maximum
- 10 The maximum amount by which a task can decrease the priority of an asyn-
- 11 chronous I/O operation below the process priority. (See §6.3.1).
- 12 Child Processes Maximum
- 13 The maximum number of simultaneous processes per real user ID.
- 14 Clock Resolution Minimum
- 15 The minimum resolution of the system realtime clock. (See `Clock_Real-`
- 16 `time` in §14.1.2.)
- 17 Filename Maximum
- 18 The maximum length of a filename, measured in POSIX characters. This
- 19 limit is pathname-specific.
- 20 Groups Maximum
- 21 The maximum number of simultaneous supplementary group IDs per pro-
- 22 cess. Zero means that this feature is not supported.
- 23 Input Line Maximum
- 24 The maximum length of an input line, measured in bytes. This limit is
- 25 pathname-specific.
- 26 Input Queue Maximum
- 27 The maximum length of a terminal input queue, measured in bytes. This
- 28 limit is pathname-specific.
- 29 Links Maximum
- 30 The maximum value of the link count of a file. This limit is pathname-
- 31 specific.
- 32 List I/O Maximum
- 33 The maximum number of I/O operations that can be specified in a list I/O
- 34 call.
- 35 Message Priority Maximum
- 36 The maximum number of message priority levels supported.
- 37 Open Files Maximum
- 38 The maximum number of files a process can have open at one time.
- 39 Open Message Queues Maximum
- 40 The maximum number of message queues that a process can have open at
- 41 one time.

- 1 Page Size
- 2 Granularity, measured in storage units, of memory mapping and process
- 3 memory locking.
- 4 NOTE: This is measured in storage units, rather than bytes, for consistency
- 5 with the address arithmetic defined in `System.Storage_Elements` in case
- 6 the system storage unit is not one byte.
- 7 Pathname Maximum
- 8 The maximum length of a pathname, in POSIX characters. This limit is
- 9 pathname-specific.
- 10 Pipe Length Maximum
- 11 The number of bytes that can be written atomically when writing to a pipe
- 12 or FIFO. This limit is pathname-specific.
- 13 Queued Signals Maximum
- 14 The maximum number of queued signals a single process can send and have
- 15 pending at the receiver(s) at any one time.
- 16 Realtime Signals Maximum
- 17 The maximum number of realtime signal numbers reserved for application
- 18 use.
- 19 Semaphores Maximum
- 20 The maximum number of semaphores that a process can have.
- 21 Semaphores Value Maximum
- 22 The maximum value a semaphore can have.
- 23 Streams Maximum
- 24 For an implementation that also supports the C-language binding, the max-
- 25 imum number of C-language streams a process can have open at one time.
- 26 Timer Overruns Maximum
- 27 The maximum number of timer expiration overruns.
- 28 Timers Maximum
- 29 The maximum number of timers a process can have.
- 30 Time Zone String Maximum
- 31 The maximum length of the **TZ** environment variable, measured in POSIX
- 32 characters.
- 33 Information about limits is provided to an application in the following forms:
- 34 Portable constant
- 35 These numeric constants, beginning with `Portable_`, define worst-case
- 36 bounds of values for a conforming implementation. If the limit is an upper
- 37 bound (e.g., `Argument List Maximum`), the portable constant for the limit is the
- 38 smallest value allowed by this standard for the limit. If the limit is a lower
- 39 bound (e.g., `Clock Resolution Minimum`), the portable constant is the largest
- 40 value allowed by this standard for the limit.
- 41 These portable constants are declared in the package `POSIX_Limits`. The
- 42 correspondence between portable constants and system limits is given in
- 43 Table 2.5.

For compatibility with POSIX.5 {B3}, 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
<i>none</i>	Asynchronous I/O Priority Delta Maximum
<i>none</i>	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_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_Open_Message_Queues_Maximum	Open Message Queues Maximum
Portable_Message_Priority_Maximum	Message Priority Maximum
Portable_Open_Files_Maximum	Open Files 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_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

Static subtype

These subtypes, ending with `..._Maximum`, 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 maximums, 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 {B3}, redundant definitions of some of the subtypes are retained in the package `POSIX`. However, those declarations in the package `POSIX` are obsolescent.

NOTE: As indicated by the phrase *static subtype* in the heading of this item and the corresponding table, these subtypes are all required to be static.

- 1 Configurable system variable
2 The functions declared in the package `POSIX_Configurable_System_`
3 `Limits` and described in §4.5, shall return the actual system-wide limit
4 imposed by the implementation.
- 5 Configurable pathname variable
6 The functions declared in the package `POSIX_Configurable_File_Lim-`
7 `its` and described in §5.4 shall return the pathname dependent limit im-
8 posed by the implementation.

9 **Table 2.6 – Static Subtypes and Limits**

10	Subtype	Limit
11	<code>Argument_List_Maxima</code>	Argument List Maximum
12	<code>Asynchronous_IO_Maxima</code>	Asynchronous I/O Maximum
13	<code>Asynchronous_IO_Priority_Delta_Maxima</code>	Asynchronous I/O Priority Delta Maximum
14	<code>Child_Processes_Maxima</code>	Child Processes Maximum
15	<code>none</code>	Clock Resolution Minimum
16	<code>Filename_Maxima</code>	Filename Maximum
17	<code>Groups_Maxima</code>	Groups Maximum
18	<code>Input_Line_Maxima</code>	Input Line Maximum
19	<code>Input_Queue_Maxima</code>	Input Queue Maximum
20	<code>Links_Maxima</code>	Links Maximum
21	<code>List_IO_Maxima</code>	List I/O Maximum
22	<code>Message_Priority_Maxima</code>	Message Priority Maximum
23	<code>Open_Files_Maxima</code>	Open Files Maximum
24	<code>Open_Message_Queues_Maxima</code>	Open Message Queues Maximum
25	<code>Page_Size_Range</code>	Page Size
26	<code>Pathname_Maxima</code>	Pathname Maximum
27	<code>Pipe_Length_Maxima</code>	Pipe Length Maximum
28	<code>Queued_Signals_Maxima</code>	Queued Signals Maximum
29	<code>Realtime_Signals_Maxima</code>	Realtime Signals Maximum
30	<code>Semaphores_Maxima</code>	Semaphores Maximum
31	<code>Semaphores_Value_Maxima</code>	Semaphores Value Maximum
32	<code>Streams_Maxima</code>	Streams Maximum
33	<code>Timer_OVERRUNS_Maxima</code>	Timer Overruns Maximum
34	<code>Timers_Maxima</code>	Timers Maximum
35	<code>Time_Zone_String_Maxima</code>	Time Zone String Maximum

36 2.7 Package `Ada_Streams`

37 **with** `Ada.Streams`;
38 **package** `Ada_Streams` **renames** `Ada.Streams`;

39 During transition from Ada83 (ISO/IEC 8652: 1987) to Ada95 (ISO/IEC 8652: 1995)
40 the implementation is permitted to replace this renaming declaration with the fol-
41 lowing package specification.

42 **package** `Ada_Streams` **is**
43 **-- pragma** `Pure(Streams)`;
44 **-- type** `Root_Stream_Type` **is abstract tagged limited private**;

```

1      type Stream_Element is -- mod
2          impl-def-unsigned-integer-type;
3      type Stream_Element_Offset is range implementation-defined;
4      subtype Stream_Element_Count is
5          Stream_Element_Offset range 0..Stream_Element_Offset'Last;
6      type Stream_Element_Array is
7          array (Stream_Element_Offset range <>) of Stream_Element;
8      -- procedure Read (
9      --     Stream: in out Root_Stream_Type;
10     --     Item: out Stream_Element_Array;
11     --     Last: out Stream_Element_Offset) is abstract;
12     -- procedure Write (
13     --     Stream: in out Root_Stream_Type;
14     --     Item: out Stream_Element_Array) is abstract;
15 private
16     implementation-defined
17 end Ada_Streams;

```

NOTE: Places where details of the Ada95 (ISO/IEC 8652: 1995) package specification have been omitted, because they use features not supported by Ada83 (ISO/IEC 8652: 1987), are indicated by comments.

The semantics of the types and operations defined in this package shall satisfy the requirements of [Ada RM {1} 13.13.1].

2.8 Package System

This standard requires the Ada implementation to include the following declarations in the package System.

```

26     Null_Address: constant Address := implementation-defined;
27     function "<" (Left, Right: Address) return Boolean;
28     function "<=" (Left, Right: Address) return Boolean;
29     function ">" (Left, Right: Address) return Boolean;
30     function ">=" (Left, Right: Address) return Boolean;
31     function Image (Addr: Address) return String;
32     function Value (Str: String) return Address;
33     Word_Size: constant := implementation-defined;

```

NOTE: All these declarations are required by Ada95 (ISO/IEC 8652: 1995), and are permitted by Ada83 (ISO/IEC 8652: 1987). They are included here to permit this standard to be upward compatible. The detailed semantics are intentionally left unspecified, to avoid conflicts with Ada95 (ISO/IEC 8652: 1995).

The type System.Address shall be non-limited. 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).

1 2.9 Package System_Storage_Elements

```
2 with System.Storage_Elements;
3 package System_Storage_Elements renames System.Storage_Elements;
```

4 During transition from Ada83 (ISO/IEC 8652: 1987) to Ada95 (ISO/IEC 8652: 1995)
5 the implementation is permitted to replace this renaming declaration with the fol-
6 lowing package specification.

```
7 with System;
8 package System_Storage_Elements is
9     -- pragma Preelaborate (System.Storage_Units);
10    -- Storage Elements and Address Arithmetic
11    type Storage_Offset is range implementation-defined;
12    subtype Storage_Count is Storage_Offset range 0..Storage_Offset_Last;
13    type Storage_Element is implementation-defined;
14    -- required to be an unsigned type whose size is one storage unit
15    type Storage_Array is array
16        (Storage_Offset range <>) of Storage_Element;
17    -- required to have a component size of one storage unit
18    -- required to have components aligned on addressable boundaries
19    -- components required to be aliased
20    function "+"
21        (Left: System.Address;
22         Right: Storage_Offset)
23        return System.Address;
24    function "+"
25        (Left: Storage_Offset;
26         Right: System.Address)
27        return System.Address;
28    function "-"
29        (Left: System.Address;
30         Right: Storage_Offset)
31        return System.Address;
32    function "-"
33        (Left, Right: System.Address)
34        return Storage_Offset;
35    function "mod"
36        (Left: System.Address;
37         Right: Storage_Offset)
38        return Storage_Offset;
39    -- Conversions to/from Integers
40    type Integer_Address is range implementation-defined;
41    function To_Address (Value: Integer_Address) return System.Address;
42    function To_Integer (Value: System.Address) return Integer_Address;
43 end System_Storage_Elements;
```

44 NOTE: Places where details of the Ada95 (ISO/IEC 8652: 1995) package specification
45 have been omitted, because they use features not supported by Ada83 (ISO/IEC 8652:
46 1987), are indicated by comments.

47 The semantics of the types and operations defined in this package shall satisfy the
48 requirements of [Ada RM {1} 13.7.1].

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`.
This 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))`. That is, 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`.
This 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 page aligned) 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 '='. There is no meaning 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 this), 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 {2}. They have no predefined meaning in this standard.

Environment variable *names* 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 named **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_Characters** 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 {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 “-”, the time zone shall be east of the Prime Meridian; otherwise, it shall be west (which may be indicated by an optional preceding “+”).

1 *rule* Indicates when to change to and back from summer time. The *rule* has the
2 form
3 *date/time,date/time*
4 where the first *date* describes when the change from standard to summer
5 time occurs and the second *date* describes when the change back happens.
6 Each *time* field describes when, in current local time, the change to the other
7 time is made.
8 The format of *date* shall be one of the following:
9 *Jn* The Julian day n ($1 \leq n \leq 365$). Leap days shall not be counted.
10 That is, in all years—including leap years—February 28 is day 59
11 and March 1 is day 60. It is impossible to refer to the occasional
12 February 29 explicitly.
13 *n* The zero-based Julian day ($0 \leq n \leq 365$). Leap days shall be
14 counted, and it is possible to refer to February 29.
15 *Mm.n.d* The d th day ($0 \leq d \leq 6$) of week n of month m of the year
16 ($1 \leq n \leq 5$, $1 \leq m \leq 12$, where week 5 means “the last d
17 day in month m ,” which may occur in either the fourth or the fifth
18 week). Week 1 is the first week in which the d th day occurs. Day
19 zero is Sunday.
20 The *time* has the same format as *offset* except that no leading sign (“-” or
21 “+”) 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.

Section 10.2 (29) of the Ada RM {1} requires the Ada implementation to allow public parameterless library procedures to be main subprograms. 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);
```

```

1  procedure Set_File_Action_To_Close
2      (Template: in out Process_Template;
3        File: in POSIX_IO.File_Descriptor);
4  procedure Set_File_Action_To_Duplicate
5      (Template: in out Process_Template;
6        File: in POSIX_IO.File_Descriptor;
7        From_File: in POSIX_IO.File_Descriptor);
8  -- 3.1.2 Process Creation
9  procedure Start_Process
10     (Child: out POSIX_Process_Identification.Process_ID;
11      Pathname: in POSIX.Pathname;
12      Template: in Process_Template;
13      Arg_List: in POSIX.POSIX_String_List:= POSIX.Empty_String_List);
14 procedure Start_Process
15     (Child: out POSIX_Process_Identification.Process_ID;
16      Pathname: in POSIX.Pathname;
17      Template: in Process_Template;
18      Env_List: in POSIX_Process_Environment.Environment;
19      Arg_List: in POSIX.POSIX_String_List:= POSIX.Empty_String_List);
20 procedure Start_Process_Search
21     (Child: out POSIX_Process_Identification.Process_ID;
22      Filename: in POSIX.Filename;
23      Template: in Process_Template;
24      Arg_List: in POSIX.POSIX_String_List:= POSIX.Empty_String_List);
25 procedure Start_Process_Search
26     (Child: out POSIX_Process_Identification.Process_ID;
27      Filename: in POSIX.Filename;
28      Template: in Process_Template;
29      Env_List: in POSIX_Process_Environment.Environment;
30      Arg_List: in POSIX.POSIX_String_List:= POSIX.Empty_String_List);
31 -- 3.1.3 Process Exit
32 type Exit_Status is range 0 .. 2**8-1;
33 Normal_Exit: constant Exit_Status:= 0;
34 Failed_Creation_Exit: constant Exit_Status:= 41;
35 Unhandled_Exception_Exit: constant Exit_Status:= 42;
36 procedure Exit_Process (Status: in Exit_Status:= Normal_Exit);
37 -- 3.1.4 Termination Status
38 type Termination_Status is private;
39 type Termination_Cause is
40     (Exited, Terminated_By_Signal, Stopped_By_Signal);
41 function Status_Available (Status: Termination_Status)
42     return Boolean;
43 function Process_ID_Of (Status: Termination_Status)
44     return POSIX_Process_Identification.Process_ID;
45 function Termination_Cause_Of (Status: Termination_Status)
46     return Termination_Cause;
47 function Exit_Status_Of (Status: Termination_Status)
48     return Exit_Status;
49 function Termination_Signal_Of (Status: Termination_Status)
50     return POSIX_Signals.Signal;
51 function Stopping_Signal_Of (Status: Termination_Status)
52     return POSIX_Signals.Signal;
53 -- 3.1.5 Wait for Process Termination
54 procedure Wait_For_Child_Process
55     (Status: out Termination_Status;
56      Block: in Boolean:= True;
57      Trace_Stopped: in Boolean:= True;
58      Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);

```

```

1      procedure Wait_For_Child_Process
2      (Status: out Termination_Status;
3       Child: in POSIX_Process_Identification.Process_ID;
4       Block: in Boolean:= True;
5       Trace_Stopped: in Boolean:= True;
6       Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
7      procedure Wait_For_Child_Process
8      (Status: out Termination_Status;
9       Group: in POSIX_Process_Identification.Process_Group_ID;
10     Block: in Boolean:= True;
11     Trace_Stopped: in Boolean:= True;
12     Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
13 private
14     implementation-defined
15 end POSIX_Process_Primitives;

```

16 3.1.1 Process Template

17 3.1.1.1 Synopsis

```

18 type Process_Template is limited private;
19 procedure Open_Template (Template: in out Process_Template);
20 procedure Close_Template (Template: in out Process_Template);
21 procedure Set_Keep_Effective_IDs (Template: in out Process_Template);
22 procedure Set_Signal_Mask
23 (Template: in out Process_Template;
24  Mask: in POSIX_Signals.Signal_Set);
25 procedure Set_Creation_Signal_Masking
26 (Template: in out Process_Template;
27  Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
28 procedure Set_File_Action_To_Open
29 (Template: in out Process_Template;
30  File: in POSIX_IO.File_Descriptor;
31  Name: in POSIX.Pathname;
32  Mode: in POSIX_IO.File_Mode:= POSIX_IO.Read_Only;
33  Options: in POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set);
34 procedure Set_File_Action_To_Close
35 (Template: in out Process_Template;
36  File: in POSIX_IO.File_Descriptor);
37 procedure Set_File_Action_To_Duplicate
38 (Template: in out Process_Template;
39  File: in POSIX_IO.File_Descriptor;
40  From_File: in POSIX_IO.File_Descriptor);

```

41 3.1.1.2 Description

42 An object of type `Process_Template` is used to represent the actions that are to take
43 place when creating another process with `Start_Process` and `Start_Process_`
44 `Search`. The user initializes a process template with `Open_Template`, modifies it
45 using the various procedures with names of the form `Set_...`, and then passes it as
46 a parameter to `Start_Process` or `Start_Process_Search`. When the user is done
47 with the template, the user closes the template, and causes any storage allocated for
48 it to be reclaimed, by calling `Close_Template`.

49 `Open_Template` shall open the process template specified by the parameter `Tem-`
50 `plate`, if it is not already open, and initialize it to the following content:

- 1 — The effective user and group IDs shall be obtained from the real user and group
2 IDs of the calling process, respectively.
- 3 — The Signal Mask attribute shall contain no signals.
- 4 — The set of file descriptor actions shall be empty, which means no file descriptors
5 will be changed when starting a process, except that file descriptors with the
6 mode `Close_On_Exec` will be closed. (See type `POSIX_IO.File_Mode`, in §6.1.1.)
- 7 `Close_Template` shall release any dynamically allocated storage that may be asso-
8 ciated with the template and shall render the template unusable. Any subsequent
9 calls using the template as a parameter will fail. This call is typically made by the
10 application just before exiting the scope of the template. Failure by an application to
11 call `Close_Template` to release templates may eventually result in storage exhaus-
12 tion.
- 13 `Set_Keep_Effective_IDs` shall cause the effective user and group IDs of the new
14 process to be the same as those of the calling process.
- 15 `Set_Signal_Mask` shall set the Signal Mask attribute of the specified template to the
16 value of the parameter `Mask`.
- 17 `Set_Creation_Signal_Masking` shall set the `Masked_Signals` parameter that
18 will be used for all the interruptible operations involved in creation and initialization
19 of the new process to the value given by `Masked_Signals`. (See §3.3.6 and §2.4.1.6.)
20 The interruptible operations involved in process creation include the open, close, and
21 duplicate operations.
- 22 `Set_File_Action_To_Close` shall cause the file descriptor named by parameter
23 `File` to be closed when the new process is started.
- 24 `Set_File_Action_To_Open` shall cause the file descriptor named by parameter
25 `File` to be opened when the new process is started, using the values of the other
26 parameters as parameters to `POSIX_IO.Open`. If the given file descriptor was already
27 open, it shall be closed before the new file is opened.
- 28 `Set_File_Action_To_Duplicate` shall cause the file descriptor named by param-
29 eter `File` to be duplicated from the file descriptor named by `From_File` when the
30 new process is started.
- 31 The effective order of events in processing the process template shall be as follows:
 - 32 (1) The new process shall be created.
 - 33 (2) The signal mask and the effective user and group IDs shall be set.
 - 34 (3) The file actions specified by the `Set_File_Action_to_xxxx` operations shall be
35 taken, in the order in which the procedures were called.
 - 36 (4) The new process image shall replace the old one, and execution of that process
37 image shall be started.
- 38 For example, the code sequence below has the effect of setting both standard output
39 and standard error to reference a (newly opened) temporary file:

```

1      Set_File_Action_To_Open
2          (Template=> T, File=> Standard_Output,
3           Name=> "/tmp/name",...);
4      Set_File_Action_To_Duplicate
5          (Template=> T, File=> Standard_Error,
6           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 occur, 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

```

26      procedure Start_Process
27          (Child:      out POSIX_Process_Identification.Process_ID;
28           Pathname:   in  POSIX.Pathname;
29           Template:   in  Process_Template;
30           Arg_List:   in  POSIX.POSIX_String_List:= POSIX.Empty_String_List);
31      procedure Start_Process
32          (Child:      out POSIX_Process_Identification.Process_ID;
33           Pathname:   in  POSIX.Pathname;
34           Template:   in  Process_Template;
35           Env_List:   in  POSIX_Process_Environment.Environment;
36           Arg_List:   in  POSIX.POSIX_String_List:= POSIX.Empty_String_List);
37      procedure Start_Process_Search
38          (Child:      out POSIX_Process_Identification.Process_ID;
39           Filename:   in  POSIX.Filename;
40           Template:   in  Process_Template;
41           Arg_List:   in  POSIX.POSIX_String_List:= POSIX.Empty_String_List);
42      procedure Start_Process_Search
43          (Child:      out POSIX_Process_Identification.Process_ID;
44           Filename:   in  POSIX.Filename;
45           Template:   in  Process_Template;
46           Env_List:   in  POSIX_Process_Environment.Environment;
47           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, and 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.

- 1 — The argument list of the child process shall be the list specified by the parameter
- 2 Arg_List. By convention, the first component of Arg_List should be the
- 3 filename of the file specified by the parameter Pathname or Filename. (See
- 4 §4.3.1 for a description of accessing arguments.)
- 5 — The environment shall be specified by the value of the parameter Env_List for
- 6 the forms of the procedures where this parameter is present. Otherwise, the
- 7 environment of the new process shall be the current environment of the calling
- 8 process. (See §2.11 and §4.3.2.)
- 9 — The effective user and group IDs of the child process shall be determined by
- 10 the information in the template and the permissions of the new process file, as
- 11 explained further below.
- 12 — If the Semaphores option is supported, the child process shall not inherit any
- 13 semaphores that are open in the parent process.
- 14 — If the Memory Locking option is supported, the child process shall not inherit any
- 15 address space memory locks established by the parent process. Any locks estab-
- 16 lished by the parent process shall not be affected.
- 17 — If the Memory Mapped Files option is supported, the child process shall not inherit
- 18 any memory mappings created in the parent process.
- 19 — If the Priority Process Scheduling option is supported, for the FIFO_Within_Pri-
- 20 orities and Round_Robin_Within_Priorities scheduling policies, the child
- 21 process shall inherit the policy and priority settings of the parent process. For
- 22 other scheduling policies, the policy and priority settings of the child are imple-
- 23 mentation defined.
- 24 — If the Timers option is supported, the child process shall not inherit any timers
- 25 created by the parent.
- 26 — If the Message Queues option is supported, the child process shall not inherit any
- 27 of the parent's message queue descriptors.
- 28 — If the Asynchronous I/O option is supported, no outstanding asynchronous input
- 29 or asynchronous output operations shall be inherited by the child process.
- 30 — If the Mutexes option is supported, the state of mutexes and condition variables in
- 31 the child process's copy of the parent process's address space (if any) is undefined.

32 Normally, if the template has not been set to keep effective IDs, the effective user and

33 group IDs of the child process are set to the real user and group IDs of the creator. If

34 the template has been set to keep effective IDs, they are set to the effective user and

35 group IDs of the creator.

36 The effect of the template on the effective user and group IDs may be overridden if

37 the new process image file has permission Set_User_ID or Set_Group_ID. (See

38 §5.1.1.) If the new process image file has permission Set_User_ID, the effective user

39 ID of the child process shall be set to the owner ID of the new process image file.

40 Similarly, if the new process image file has permission Set_Group_ID (see §5.1.1),

41 the effective group ID of the child process shall be set to the group ID of the new

42 process image file. The real user ID, real group ID, and supplementary group IDs

43 of the child process remain the same as those of the calling process. If the Saved IDs

44 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 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 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 occur 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 occur, 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 active partition's execution 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. The sending of a `Signal_Hangup` signal as described below indirectly terminates child processes in some circumstances. Child processes of a terminated process shall be assigned a new parent process ID, corresponding to an implementation defined system 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, and if the exit of the process causes a process group to become orphaned, and if any member of the newly

1 orphaned process group is stopped, then a `Signal_Hangup` signal followed by a
2 `Signal_Continue` signal shall be sent to each process in the newly orphaned
3 process group.

4 — If the Semaphores option is supported, all open named semaphores in the calling
5 process shall be closed, as if by appropriate calls to `POSIX_Semaphores.Close`.

6 — If the Memory Locking option is supported, any memory locks established by the
7 process via calls to `POSIX_Memory_Locking.Lock_All` or `POSIX_Memory_`
8 `Range_Locking.Lock_Range` shall be removed. If locked pages in the calling
9 process's address space are also mapped into other processes' address spaces and
10 are locked by those processes, the locks established by the other processes shall
11 be unaffected by this process's call to `Exit_Process`.

12 — If the Memory Mapped Files option is supported, memory mappings created in the
13 process are unmapped before the process is destroyed.

14 — If the Message Queues option is supported, all open message queue descriptors
15 in the calling process shall be closed, as if by appropriate calls to `POSIX_`
16 `Message_Queues.Close`.

17 — If the Asynchronous I/O option is supported, any outstanding cancelable asyn-
18 chronous I/O operations may be canceled. Those asynchronous I/O operations
19 that are not canceled shall complete as if the `Exit_Process` operation had not
20 yet occurred, but any associated signal notifications shall be suppressed. The
21 `Exit_Process` operation itself may or may not block awaiting such I/O comple-
22 tion. Whether any I/O is canceled, and which I/O may be canceled upon `Exit_`
23 `Process`, is implementation defined.

24 When an Ada active partition exits without explicitly calling `Exit_Process`, the
25 Ada language implementation shall provide the exit status value.

26 The exit status value of an Ada active partition shall be `Unhandled_Exception_`
27 `Exit` if completion occurs because of an unhandled exception.

28 For a child process that is terminated before the beginning of execution of the pro-
29 gram or active partition in the new process image file, as described in §3.1.2, the
30 exit status value shall be `Failed_Creation_Exit`. By convention, a POSIX Ada
31 application should not use the two predefined exit status values for other meanings.
32 However, the POSIX Ada interface shall permit these values to be used with `Exit_`
33 `Process`.

34 **3.1.3.3 Error Handling**

35 No exceptions shall be raised by `Exit_Process`.

36 **3.1.4 Termination Status**

37 **3.1.4.1 Synopsis**

```
38 type Termination_Status is private;
39 type Termination_Cause is
40   (Exited, Terminated_By_Signal, Stopped_By_Signal);
41 function Status_Available (Status: Termination_Status)
42   return Boolean;
```

```

1      function Process_ID_Of (Status: Termination_Status)
2          return POSIX_Process_Identification.Process_ID;
3      function Termination_Cause_Of (Status: Termination_Status)
4          return Termination_Cause;
5      function Exit_Status_Of (Status: Termination_Status)
6          return Exit_Status;
7      function Termination_Signal_Of (Status: Termination_Status)
8          return POSIX_Signals.Signal;
9      function Stopping_Signal_Of (Status: Termination_Status)
10         return POSIX_Signals.Signal;

```

11 3.1.4.2 Description

12 The type `Termination_Status` is used to provide information about the condition
13 that caused termination of a process. If the implementation supports the Job Control
14 option, the type `Termination_Status` is also used to provide information about the
15 condition that caused a process to be stopped. Information may be extracted from a
16 value of type `Termination_Status` by means of query functions.

17 For each value of type `Termination_Status`, it shall be possible to determine
18 whether status is available from that value. All objects of type `Termination_`
19 `Status` shall have a default initial value from which status is not available.

20 Each value of the type `Termination_Status` from which status is available shall
21 have an associated value of `Process_ID`, which specifies the process for which status
22 is available, and an associated value of `Termination_Cause`, which specifies one of
23 the reasons an Ada process may terminate or stop. The possible termination causes
24 are

25 Exited

26 Some task in the process called `Exit_Process` (see §3.1.3), or the process
27 was implicitly terminated by the action of the Ada language implementa-
28 tion in response to normal active partition completion, propagation of an
29 unhandled exception, or receipt of a signal caught by the Ada language im-
30 plementation. In all these cases the process is said to have “exited.”

31 NOTE: The Ada language requires that the termination of the environment
32 task awaits the termination of all tasks created as part of the execution of the
33 partition [Ada RM {1} 10.2 (25)], except when a task calls `Exit_Process`.

34 Terminated_By_Signal

35 The process was terminated by the action of the operating system due to
36 the receipt of a signal that was not caught by the application or the runtime
37 system of the Ada language implementation.

38 Stopped_By_Signal

39 The process was stopped by a signal.

40 `Status_Available` shall return `True` if-and-only-if status is available from the
41 `Status` parameter.

42 `Process_ID_Of` shall return the process ID of the `Status` parameter, if status is
43 available from it. Otherwise, this operation shall raise `POSIX_Error`.

44 `Termination_Cause_Of` shall return the termination cause of the `Status` parame-
45 ter, 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 occur, 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);

```

```
1      procedure Wait_For_Child_Process
2      (Status: out Termination_Status;
3       Group: in POSIX_Process_Identification.Process_Group_ID;
4       Block: in Boolean:= True;
5       Trace_Stopped: in Boolean:= True;
6       Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
```

7 **3.1.5.2 Description**

8 **Wait_For_Child_Process** allows the calling process to enquire about the status of
9 one of its child processes. Status shall be available for a process if the process has
10 terminated or is stopped and its status has not been reported by **Wait_For_Child_**
11 **Process** since it terminated or was last stopped.

12 A process that is stopped may be signaled to continue. If a child process is stopped
13 and then is signaled to continue before its parent attempts to obtain status for it,
14 the parent will not find status available until the process is stopped again or ter-
15 minates. If a process is signaled to continue after status has been obtained, status
16 may be obtained for that process again if it is stopped again or terminates. If status
17 information is available for two or more processes, the order in which their status is
18 reported is unspecified.

19 An implementation may define additional circumstances under which status is avail-
20 able for a child process. This shall not occur unless the calling process or one of its
21 child processes explicitly makes use of a nonstandard extension. In these cases, the
22 interpretation of the reported status shall be implementation defined.

23 The status of a child process has not yet been reported if either

- 24 (1) It has not terminated and is not stopped.
- 25 (2) It has terminated or is stopped and has status available.

26 **Wait_For_Child_Process** shall permit the calling process to obtain the status
27 of one out of a set of child processes. This set is specified by the parameters. The
28 parameter **Traced_Stopped** specifies whether stopped child processes are to be in-
29 cluded if the implementation supports the Job Control option. If the implementation
30 supports the Job Control option, stopped child processes shall be included in the set
31 only if **Traced_Stopped** is True. The parameter **Masked_Signals** specifies which
32 additional signals shall be blocked from delivery during the operation. (See §2.4.1.6.)
33 If more than one task waits for a particular child, only one task shall return with
34 the child's status; what occurs for the other tasks is unspecified.

35 For the form of **Wait_For_Child_Process** without a parameter **Child** or **Group**,
36 the set of processes for which status is requested contains all the child processes
37 of the calling process. For the form of **Wait_For_Child_Process** with parameter
38 **Child**, this set contains only the child process with the specified process ID. For the
39 form of **Wait_For_Child_Process** with parameter **Group**, the set contains all the
40 child processes that are members of the specified process group.

41 When this set contains a child process whose status has not yet been reported, for
42 which status is available, the status of the child shall be returned in the parame-
43 ter **Status**. The process ID of the child process shall be available by applying the
44 function **Process_ID_Of** to the value returned in the parameter **Status**.

1 If the set of processes for which status is requested contains no child processes, PO-
2 SIX_Error shall be raised with error code No_Child_Process.

3 If the set of processes for which status is requested contains child processes whose
4 statuses have not yet been reported, but status is not available for any of these,
5 the operation shall either block until status becomes available or return without
6 reporting status, as determined by the value of the parameter Block. If Block
7 is False, the procedure shall return a value for the parameter Status such that
8 Status_Available shall return False.

9 NOTE: The other query functions raise POSIX_Error for such a value of Status.

10 If Block is True, the procedure shall block the calling process until status becomes
11 available for a child process in the set specified by the other parameters.

12 NOTE: In the Ada multitasking environment, waiting for status of a child process may
13 affect the entire process, not just the calling task. Whether the entire process or just the
14 calling task is blocked is specified by the constant POSIX.Wait_For_Child_Blocking_
15 Behavior. (See §2.4.1.5.)

16 3.1.5.3 Error Handling

17 If any of the following conditions occur, the exception POSIX_Error shall be raised,
18 with the corresponding error code:

19 No_Child_Process

20 Wait_For_Child_Process was called with parameters that specify an
21 empty set of child processes.

22 Interrupted_Operation

23 The operation was interrupted by a signal.

24 3.2 Package POSIX_Unsafe_Process_Primitives

```
25 with POSIX,  
26     POSIX_Process_Environment,  
27     POSIX_Process_Identification;  
28 package POSIX_Unsafe_Process_Primitives is  
29     -- 3.2.1 Process Creation  
30     function Fork return POSIX_Process_Identification.Process_ID;  
31     -- 3.2.2 File Execution  
32     procedure Exec  
33         (Pathname: in POSIX.Pathname;  
34          Arg_List: in POSIX.POSIX_String_List := POSIX.Empty_String_List;  
35          Env_List: in POSIX_Process_Environment.Environment);  
36     procedure Exec  
37         (Pathname: in POSIX.Pathname;  
38          Arg_List: in POSIX.POSIX_String_List := POSIX.Empty_String_List);  
39     procedure Exec_Search  
40         (Filename: in POSIX.Filename;  
41          Arg_List: in POSIX.POSIX_String_List := POSIX.Empty_String_List;  
42          Env_List: in POSIX_Process_Environment.Environment);  
43     procedure Exec_Search  
44         (Filename: in POSIX.Filename;  
45          Arg_List: in POSIX.POSIX_String_List := POSIX.Empty_String_List);  
46 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 nonterminated 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, and system time of the descendants) shall be set to zero. (See §4.2.1 for a description of process time accounting.)

- 1 — File locks previously set by the parent shall not be inherited by the child. (See
2 §6.2.1 for a description of file locking.)
- 3 — The set of signals pending for the child process shall be initialized to the empty
4 set. (See §3.3.7 for a description of signal sets.)
- 5 — If the parent process has more than one unterminated task, the set of untermi-
6 nated tasks in the child process is unspecified.
- 7 — If the Semaphores option is supported, any semaphores that are open in the par-
8 ent process shall also be open in the child process.
- 9 — If the Memory Locking option is supported, the child process shall not inherit any
10 address space memory locks established by the parent process via calls to PO-
11 SIX_Memory_Locking.Lock_All or POSIX_Memory_Range_Locking.Lock_-
12 Range.
- 13 — If the Memory Mapped Files option is supported, memory mappings created in
14 the parent are retained in the child process. Private mappings (see POSIX_-
15 Memory_Mapping.Map_Private) inherited from the parent shall also be private
16 mappings in the child. Any modifications to the data in these mappings made by
17 the parent prior to calling Fork shall be visible to the child. Any modifications
18 to the data in private mappings made by the parent after Fork returns shall be
19 visible only to the parent. Modifications to the data in private mappings made
20 by the child shall be visible only to the child.
- 21 — If the Priority Process Scheduling option is supported, the child process shall inherit
22 the policy and priority settings of the parent process during a Fork operation,
23 for the the FIFO_Within_Priorities_Round_Robin_Within_Priorities
24 scheduling policies. For other scheduling policies, the policy and priority settings
25 on Fork are implementation defined.
- 26 — If the Timers option is supported, the child process shall not inherit any timers
27 created by the parent.
- 28 — If the Message Queues option is supported, the child process has its own copy of
29 the parent's message queue descriptors. Each of the child's message queue de-
30 scriptors refers to the same open message queue description as the correspond-
31 ing message queue descriptor of the parent.
- 32 — If the Asynchronous I/O option is supported, no asynchronous input or asynchron-
33 ous output operations are inherited by the child process.
- 34 All other process attributes defined by this standard shall be the same in the parent
35 and the child processes. The inheritance of attributes not defined by this standard is
36 implementation defined.
- 37 Upon successful completion, Fork shall return the value Null_Process_ID to the
38 child process and shall return the Process_ID of the child process to the parent.
39 Both processes shall continue to execute from the call to Fork. Both parent and
40 child processes shall be capable of executing independently before either terminates.
- 41 If Fork raises an exception, no process shall be created.

42 3.2.1.3 Error Handling

- 43 If any of the following conditions occur, the exception POSIX_Error shall be raised,
44 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: This is similar to the treatment of a process that 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.

1 The parameter `Arg_List` shall become the new argument list of the calling process.
2 By convention, the first component of `Arg_List` should be the filename of the file
3 specified by the parameter `Pathname` or `Filename`. (See §4.3.1.)

4 If the parameter `Environment` is present, it shall become the new environment of
5 the calling process. Otherwise, the current environment of the calling process shall
6 be maintained. (See §2.11 and §4.3.2.)

7 The number of POSIX characters available for the combined argument and envi-
8 ronment lists may be limited by the implementation to `Argument_List_Maximum`. (In-
9 formation on this limit is provided by `Argument_List_Maximum` in the package
10 `POSIX_Configurable_System_Limits`, described in §4.5.)

11 File descriptors open in the calling process image shall remain open in the new pro-
12 cess image, except that those with mode `Close_on_Exec` (see §6.1.7) shall be closed.
13 For those file descriptors that remain open, all attributes of the open file description,
14 including the file locks, shall remain unchanged by this operation. (See §6.2.1.)

15 Any active directory iterators shall be terminated, and any associated implementa-
16 tion file structures shall be closed.

17 The operation shall leave the signal actions in the calling process unchanged, except
18 that any bindings of signals to task entries shall be broken, and the actions for these
19 signals shall revert to the defaults.

20 NOTE: This implies that signals that are set to be ignored by the calling process shall
21 also be ignored in the new process image.

22 The set of signals blocked from delivery for the environment task of the process shall
23 be that of the calling task.

24 If the new process image file has permission `Set_User_ID`, the 2.2.2.35 of the process
25 shall be set to the owner ID of the new process image file. (See §4.1.3.) Similarly, if
26 the new process image file has permission `Set_Group_ID`, the 2.2.2.34 of the process
27 is set to the 2.2.2.59 of the new process image file. The real 2.2.2.150, real group
28 ID, and 2.2.2.135s of the process remain the same as those of the calling process. If
29 the `Saved IDs` option is supported, the effective user ID and effective group ID of the
30 process shall be saved (as the 2.2.2.122 and the 2.2.2.121) for use by the `Set_User_-`
31 `ID` and `Set_Group_ID` operations. (See §4.1.4.)

32 The process also shall retain the following attributes from the calling process:

- 33 — Process ID
- 34 — Parent process ID
- 35 — Process group ID
- 36 — Session membership
- 37 — Real user ID
- 38 — Real group ID
- 39 — Supplementary group IDs
- 40 — Time left until an alarm clock signal (see the following paragraphs)

- 1 — Current working directory
- 2 — Root directory
- 3 — File mode creation mask (see §5.1.1)
- 4 — Pending signals (see §3.3)
- 5 — Process times (see §4.2.1)

6 NOTE: The time left until an alarm clock signal is not something that would ordinar-
7 ily be observable to an Ada application, since this signal is a reserved signal. (See
8 §2.2.2.117.) However, this may be observable by the new process image if it was imple-
9 mented in another programming language.

10 If the Semaphores option is supported, any named semaphores that are open in the
11 calling process shall be closed as if by appropriate calls to `POSIX_Semaphores.-`
12 `Close`.

13 If the Memory Locking option is supported, memory locks established by the calling
14 process via calls to `POSIX_Memory_Locking.Lock_All` or `POSIX_Memory_Range_-`
15 `Locking.Lock_Range` shall be removed. If locked pages in the calling process's
16 address space are also mapped into other processes' address spaces and are locked
17 by those processes, the locks established by the other processes shall be unaffected
18 by this process's call to `Exec` or `Exec_Search`. If `Exec` or `Exec_Search` fails, the
19 effect on memory locks is unspecified.

20 If the Memory Mapped Files option is supported, memory mappings created in the pro-
21 cess are unmapped before the address space is rebuilt for the new process image.

22 If the Priority Process Scheduling option is supported, for the `FIFO_Within_Priori-`
23 `ties` and `Round_Robin_Within_Priorities` scheduling policies, the policy and
24 priority settings shall not be changed by a call to `Exec` or `Exec_Search`. For other
25 scheduling policies, the policy and priority settings on `Exec` or `Exec_Search` are
26 implementation defined.

27 If the Timers option is supported, any timers created by the calling process shall be
28 deleted before replacing the current process image with the new process image.

29 If the Message Queues option is supported, all open message queue descriptors in the
30 calling process shall be closed, as described in `POSIX_Message_Queues.Close`.

31 If the Asynchronous I/O option is supported, any outstanding asynchronous I/O oper-
32 ations may be canceled. Those asynchronous I/O operations that are not canceled
33 shall complete as if the `Exec` or `Exec_Search` had not yet occurred, but any as-
34 sociated signal notifications shall be suppressed. It is unspecified whether `Exec` or
35 `Exec_Search` itself blocks, awaiting such I/O completion. In no event, however, shall
36 the new process image created by `Exec` or `Exec_Search` be affected by the presence
37 of outstanding asynchronous I/O operations at the time `Exec` or `Exec_Search` is
38 called. Whether any I/O is canceled, and which I/O may be canceled upon `Exec` or
39 `Exec_Search`, is implementation defined.

40 All process attributes defined by this standard and not specified in this clause shall
41 remain unchanged by the operations `Exec` and `Exec_Search`. The inheritance of
42 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 occur, 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.

`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.

1 **3.3 Package POSIX_Signals**

2 This package defines the type `Signal` and operations for sending, accepting, and
3 catching signals. Signals are events that may affect the execution of a process.

```
4  with POSIX,
5      POSIX_Process_Identification,
6      Ada_Task_Identification,
7      System;
8  package POSIX_Signals is
9      -- 3.3.2 Signal Type
10     type Signal is implementation-defined-integer;;
11     function Image (Sig: Signal) return String;
12     function Value (Str: String) return Signal;
13     -- 3.3.3 Standard Signals
14     Signal_Null,
15     SIGNULL: constant Signal:= 0;
16     Signal_Abort,
17     SIGABRT: constant Signal:= implementation-defined;
18     Signal_Alarm,
19     SIGALRM: constant Signal:= implementation-defined;
20     Signal_Bus_Error,
21     SIGBUS: constant Signal:= implementation-defined;
22     Signal_Floating_Point_Error,
23     SIGFPE: constant Signal:= implementation-defined;
24     Signal_Hangup,
25     SIGHUP: constant Signal:= implementation-defined;
26     Signal_Illegal_Instruction,
27     SIGILL: constant Signal:= implementation-defined;
28     Signal_Interrupt,
29     SIGINT: constant Signal:= implementation-defined;
30     Signal_Kill,
31     SIGKILL: constant Signal:= implementation-defined;
32     Signal_Pipe_Write,
33     SIGPIPE: constant Signal:= implementation-defined;
34     Signal_Quit,
35     SIGQUIT: constant Signal:= implementation-defined;
36     Signal_Segmentation_Violation,
37     SIGSEGV: constant Signal:= implementation-defined;
38     Signal_Terminate,
39     SIGTERM: constant Signal:= implementation-defined;
40     Signal_User_1,
41     SIGUSR1: constant Signal:= implementation-defined;
42     Signal_User_2,
43     SIGUSR2: constant Signal:= implementation-defined;
44     Signal_Child,
45     SIGCHLD: constant Signal:= implementation-defined;
46     Signal_Continue,
47     SIGCONT: constant Signal:= implementation-defined;
48     Signal_Stop,
49     SIGSTOP: constant Signal:= implementation-defined;
50     Signal_Terminal_Stop,
51     SIGTSTP: constant Signal:= implementation-defined;
52     Signal_Terminal_Input,
53     SIGTTIN: constant Signal:= implementation-defined;
54     Signal_Terminal_Output,
55     SIGTTOU: constant Signal:= implementation-defined;
56     subtype Realtime_Signal is Signal range implementation-defined;
```



```

1      -- 3.3.7 Signal Sets
2      type Signal_Set is private;
3      procedure Add_Signal
4          (Set: in out Signal_Set;
5           Sig: in Signal);
6      procedure Add_All_Signals (Set: in out Signal_Set);
7      procedure Delete_Signal
8          (Set: in out Signal_Set;
9           Sig: in Signal);
10     procedure Delete_All_Signals (Set: in out Signal_Set);
11     function Is_Member
12         (Set: Signal_Set;
13          Sig: Signal)
14         return Boolean;
15     -- 3.3.8 Block and Unblock Signals
16     procedure Set_Blocked_Signals
17         (New_Mask: in Signal_Set;
18          Old_Mask: out Signal_Set);
19     procedure Block_Signals
20         (Mask_to_Add: in Signal_Set;
21          Old_Mask: out Signal_Set);
22     procedure Unblock_Signals
23         (Mask_to_Subtract: in Signal_Set;
24          Old_Mask: out Signal_Set);
25     function Blocked_Signals return Signal_Set;
26     -- 3.3.9 Ignore Signals
27     procedure Ignore_Signal (Sig: in Signal);
28     procedure Unignore_Signal (Sig: in Signal);
29     procedure Restore_Default_Action (Sig: in Signal)
30         renames Unignore_Signal;
31     function Is_Ignored (Sig: Signal) return Boolean;
32     -- 3.3.10 Controlling Generation of Signal for Child Process
33     procedure Set_Stopped_Child_Signal (Enable: in Boolean:= True);
34     function Stopped_Child_Signal_Enabled return Boolean;
35     -- 3.3.11 Examine Pending Signals
36     function Pending_Signals return Signal_Set;
37     -- 3.3.12 Signal Event Notification
38     type Signal_Event is private;
39     type Signal_Data is private;
40     type Notification is range implementation-defined;
41     No_Notification: constant Notification:= implementation-defined;
42     Signal_Notification: constant Notification:= implementation-defined;
43     function Get_Signal (Event: Signal_Event) return Signal;
44     procedure Set_Signal
45         (Event: in out Signal_Event;
46          Sig: in Signal);
47     function Get_Notification (Event: Signal_Event) return Notification;
48     procedure Set_Notification
49         (Event: in out Signal_Event;
50          Notify: in Notification);
51     function Get_Data (Event: Signal_Event) return Signal_Data;
52     procedure Set_Data
53         (Event: in out Signal_Event;
54          Data: in Signal_Data);
55     -- 3.3.13 Signal Information
56     type Signal_Source is range implementation-defined;
57     From_Send_Signal: constant Signal_Source:= implementation-defined;

```

```

1      From_Queue_Signal: constant Signal_Source:= implementation-defined;
2      From_Timer:        constant Signal_Source:= implementation-defined;
3      From_Async_IO:      constant Signal_Source:= implementation-defined;
4      From_Message_Queue: constant Signal_Source:= implementation-defined;
5      type Signal_Info is private;
6      function Get_Signal (Info: Signal_Info) return Signal;
7      procedure Set_Signal
8          (Info: in out Signal_Info;
9           Sig: in Signal);
10     function Get_Source (Info: Signal_Info) return Signal_Source;
11     procedure Set_Source
12         (Info: in out Signal_Info;
13          Source: in Signal_Source);
14     function Has_Data (Source: Signal_Source) return Boolean;
15     function Get_Data (Info: Signal_Info) return Signal_Data;
16     procedure Set_Data
17         (Info: in out Signal_Info;
18          Data: in Signal_Data);
19     -- 3.3.14 Control Signal Queueing
20     procedure Enable_Queueing (Sig: in Signal);
21     procedure Disable_Queueing (Sig: in Signal);
22     -- 3.3.15 Wait for Signal
23     function Await_Signal (Set: Signal_Set) return Signal;
24     function Await_Signal_Or_Timeout
25         (Set: in Signal_Set;
26          Timeout: POSIX.Timespec)
27         return Signal;
28     -- 3.3.16 Wait for Signal with Information
29     function Await_Signal (Set: Signal_Set) return Signal_Info;
30     function Await_Signal_Or_Timeout
31         (Set: in Signal_Set;
32          Timeout: POSIX.Timespec)
33         return Signal_Info;
34     -- 3.3.17 Signal Entries -- obsolescent
35     Signal_Abort_Ref: constant System.Address:= implementation-defined;
36     -- Signal_Floating_Point_Error intentionally omitted.
37     Signal_Hangup_Ref: constant System.Address:= implementation-defined;
38     -- Signal_Illegal_Instruction intentionally omitted.
39     Signal_Interrupt_Ref: constant System.Address:= implementation-defined;
40     -- Signal_Kill intentionally omitted.
41     Signal_Pipe_Write_Ref: constant System.Address:= implementation-defined;
42     Signal_Quit_Ref: constant System.Address:= implementation-defined;
43     Signal_Terminate_Ref: constant System.Address:= implementation-defined;
44     Signal_User_1_Ref: constant System.Address:= implementation-defined;
45     Signal_User_2_Ref: constant System.Address:= implementation-defined;
46     Signal_Child_Ref: constant System.Address:= implementation-defined;
47     Signal_Continue_Ref: constant System.Address:= implementation-defined;
48     -- Signal_Stop intentionally omitted.
49     Signal_Terminal_Stop_Ref: constant System.Address:= implementation-defined;
50     Signal_Terminal_Input_Ref: constant System.Address:= implementation-defined;
51     Signal_Terminal_Output_Ref: constant System.Address:= implementation-defined;
52     function Signal_Reference (Sig: Signal) return System.Address;
53     -- 3.3.18 Send a Signal
54     procedure Send_Signal
55         (Process: in POSIX_Process_Identification.Process_ID;
56          Sig: in Signal);
57     procedure Send_Signal
58         (Group: in POSIX_Process_Identification.Process_Group_ID;

```

```

1      Sig:  in Signal);
2      procedure Send_Signal (Sig: in Signal);
3      -- 3.3.19 Queue a Signal
4      procedure Queue_Signal
5        (Process: in POSIX_Process_Identification.Process_ID;
6         Sig: in Signal;
7         Data: in Signal_Data);
8      -- 3.3.20 Interrupt a Task
9      procedure Interrupt_Task (T: in Ada_Task_Identification.Task_ID);
10     private
11       implementation-defined
12     end POSIX_Signals;
```

13 3.3.1 Signal Model

14 A signal is an abstraction of an event. An occurrence of a signal is said to be a
 15 *generated* when the event that causes the signal occurs. Examples of such events
 16 include detection of hardware faults, timer expiration, and terminal activity, as well
 17 as the invocation of the `Send_Signal` operation. A signal may be generated for a
 18 process. A signal may also be generated for a particular task within a process. In
 19 some cases, the same event generates signals for multiple processes.

20 If signals can be generated for tasks, at the time a signal is generated, a determina-
 21 tion shall be made whether the signal has been generated for the process or for a spe-
 22 cific task within the process. Signals that are generated by some action attributable
 23 to a particular task, such as a hardware fault, shall be generated for the task that
 24 caused the signal to be generated. Signals that are generated in association with a
 25 process ID, a process group ID, or an asynchronous event such as terminal activity
 26 shall be generated for the process.

27 A signal may be *accepted* synchronously by a task within a process. Alternatively,
 28 it may be *delivered* asynchronously to a task within a process. The latter will only
 29 be visible to the task indirectly, through a service performed by the Ada runtime
 30 system such as the `Interrupt_Task` operation described in §3.3.20, and through
 31 the exception `POSIX_Error` being raised with error code `Interrupted_Operation`
 32 if the task is executing a call to the interruptible operations defined by this standard
 33 and described in §3.3.6. These two different kinds of signal notification are explained
 34 below.

35 A task is said to *accept a signal* when the signal is selected and returned by one of
 36 the *signal-awaiting operations* defined by this standard, which cause a task to block
 37 and wait for the arrival of a signal. These operations are as follows:

- 38 — `POSIX_Signals.Await_Signal`
- 39 — `POSIX_Signals.Await_Signal_Or_Timeout`

40 See §3.3.15 and §3.3.16 for more details.

41 If the `Signal Entries` option is supported, a task is also said to *accept* a signal when it
 42 executes a rendezvous via an **accept** statement for a task entry that has been bound
 43 to the signal.

- 1 NOTE: Throughout this section, the term “accept” is used to mean both of the signal
2 notification mechanisms defined above, and is used in contrast to the term “deliver”
3 defined below.
- 4 The system is said to *deliver a signal* occurrence when it performs an action in
5 response to the signal occurrence, other than holding the signal occurrence pending
6 for a task to accept it. For each process and signal, there shall be a corresponding
7 *signal action* that shall be taken by the system when it delivers the signal.
- 8 The possible signal actions include the following:
- 9 Ignore the signal.
- 10 Only certain signals can be *ignored*. An attempt to call `Ignore_Signal`
11 or `Unignore_Signal` for a signal that is not permitted to be ignored shall
12 cause `POSIX_Error` to be raised with error code `Invalid_Argument`.
13 If the signal action for a signal is to ignore it, delivery of the signal shall
14 have no effect on the process.
15 Setting the signal action to ignore for a signal that is pending shall cause
16 any pending occurrences of the signal to be discarded, whether or not the
17 signal is blocked. Any queued values pending shall be discarded and the
18 resources used to queue them shall be released and made available to queue
19 other signals. If a process sets the action for `Signal_Child` to ignore, the
20 behavior is unspecified.
- 21 Catch the signal.
- 22 Other languages allow that certain signals may be caught within the process
23 to which they are delivered, via a signal handler subprogram installed by
24 the application (see POSIX.1 {2}). This mechanism is not directly available
25 to Ada applications under this standard. However, handler subprograms
26 may be installed by the Ada language implementation, for the reserved
27 signals (see §2.2.2.117), and possibly for signals attached to task entries.
28 In mixed-language programs, they may also be installed via interfaces not
29 defined by this standard; the effect on an Ada active partition of calling
30 an Ada tasking operation or transferring control from a handler installed
31 by such means (whether via C-language `longjmp()`, propagation of an Ada
32 exception, `goto`, or whatever) is undefined.
- 33 Take the default action.
- 34 Every signal has a default signal action. (See §3.3.4.) The default action is
35 taken if all of the following conditions are satisfied:
- 36 — There is no task currently waiting for the signal.
37 — There is no entry bound to the signal.
38 — The signal action has not been set to ignore the signal.
- 39 The effect of changing the signal action for a signal that is currently awaited by a
40 task is unspecified.
- 41 NOTE: It is specified in POSIX.1b {3} and POSIX.1c {4} that when a task calls one
42 of the signal-awaiting operations to wait for a set of signals the effect on the actions is
43 unspecified. Thus, it is unsafe to allow a signal to be unblocked for delivery to any task
44 after any task has performed a signal-awaiting operation on that signal in the process,
45 until some operation has been executed to restore the signal action to a well-defined
46 state. (See §3.3.15 of this standard.)

1 During the time between the generation of a signal and its acceptance or delivery,
2 the signal is said to be *pending*. If the occurrence was generated for the process it
3 is pending for the process; if it was generated for a specific task it is pending for
4 that task. Ordinarily, this interval cannot be detected by an application. However,
5 a signal can be *blocked* from delivery to a task. If a signal occurrence is generated
6 for a task and the signal action associated with the signal is anything other than to
7 ignore the signal, the signal occurrence shall remain pending until it is unblocked
8 for delivery to that task, it is accepted by a call to a signal-awaiting operation by that
9 task, or the action is set to ignore the signal. Signals generated for the process shall
10 be accepted by or delivered to exactly one of the tasks within the process that is ready
11 to accept the signal or for which the signal is not blocked from delivery. If there are
12 no tasks that are ready to accept the signal and the signal is blocked from delivery
13 to all tasks within the process, the signal occurrence shall remain pending on the
14 process until a task is ready to accept the signal, the signal is no longer blocked from
15 delivery to some task, or the action associated with the signal is set to ignore the
16 signal. If the action associated with a blocked signal is to ignore the signal, and if
17 that signal is generated for the process or task it is unspecified whether the signal is
18 discarded immediately upon generation or remains pending.

19 Generation of a signal is guaranteed to cause a subsequent attempt at delivery, but
20 there is no guarantee that an attempt at delivery will be made for every occurrence
21 of the signal. In the case where the signal is generated again while a previous occur-
22 rence is still pending, the system need not queue multiple occurrences of a signal.

23 If the Realtime Signals option is supported, *queueing* may be enabled for a signal (see
24 §3.3.14). When queueing is enabled for a signal, operations that generate occurrences
25 of the signal are guaranteed to queue the signal if they return normally, and to fail
26 if there are insufficient resources to queue the signal.

27 The set of signals that is blocked from delivery to a task is called the *signal mask* of
28 the task. It is unspecified whether the signal mask of a task is an attribute of the
29 individual task or an attribute of the process to which the task belongs.

30 For the environment task, the initial signal mask is that specified for the process in
31 §3.1.2 and §3.2.1.

32 NOTE: It is anticipated that the paragraph above may be modified by a future revision
33 of this standard, to require that the realtime signals always be initially masked for a
34 process that is an Ada active partition. See §B.3.8 for further details.

35 For all other tasks, the initial signal mask shall include all the signals that are not
36 reserved signals (see §2.2.2.117) and are not bound to entries of the task.

37 An application can explicitly modify the signal mask via calls to `Block_Signals`,
38 `Unblock_Signals`, and `Set_Blocked_Signals`.

39 The implementation of the Ada language or this standard may temporarily block
40 signals during the execution of operations defined by this standard, and during calls
41 to the Ada runtime system, as necessary.

42 The implementation is also allowed to block or unblock any of the reserved signals
43 (see §2.2.2.117) and any of the signals bound by the application to task entries, at
44 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, asynchronous I/O 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 which 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 which 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.

1 **3.3.2 Signal Type**

2 **3.3.2.1 Synopsis**

```
3        type Signal is implementation-defined-integer;;  
4        function Image (Sig: Signal) return String;  
5        function Value (Str: String) return Signal;
```

6 **3.3.2.2 Description**

7 The type `Signal` shall be used to represent signals. The mapping from signals to
8 values of type `Signal` is implementation defined, subject to the requirement that it
9 be one-to-one.

10 The values of the type `Signal` shall represent valid signals in the implementation.

11 `Image` shall return a string identifying the signal specified by the parameter `Sig`. If
12 `Sig` is the value of one of the signals defined by this standard, the value returned by
13 `Image` shall be the identifier of the corresponding long-name constant, in uppercase.
14 Otherwise, the value returned by `Image` is implementation defined, subject to the
15 requirement that it shall be distinct for each signal supported by the implementation.

16 `Value` shall return the value of type `Signal` corresponding to the parameter `Str`.
17 If `Str` matches either the short name or the long name of an identifier for a signal
18 supported by the implementation, ignoring leading and trailing blanks and adjusting
19 the case of letters as necessary, `Value` shall return the corresponding signal value.
20 Otherwise, if `Str` does not match the image of any signal supported by the imple-
21 mentation, `Value` shall raise `Constraint_Error`. In any case, for every signal `S`
22 supported by the implementation, `Value(Image(S))=S`.

23 **3.3.2.3 Error Handling**

24 `Constraint_Error` shall be raised by `Value` if `Str` does not match the image of any
25 of the signals supported by the implementation.

26 No exceptions shall be raised by `Image`.

27 **3.3.3 Standard Signals**

28 **3.3.3.1 Synopsis**

```
29    Signal_Null,  
30    SIGNULL: constant Signal:= 0;  
31    Signal_Abort,  
32    SIGABRT: constant Signal:= implementation-defined;  
33    Signal_Alarm,  
34    SIGALRM: constant Signal:= implementation-defined;  
35    Signal_Bus_Error,  
36    SIGBUS: constant Signal:= implementation-defined;  
37    Signal_Floating_Point_Error,  
38    SIGFPE: constant Signal:= implementation-defined;  
39    Signal_Hangup,  
40    SIGHUP: constant Signal:= implementation-defined;
```



```
1      Signal_Illegal_Instruction,  
2      SIGILL: constant Signal:= implementation-defined;  
3      Signal_Interrupt,  
4      SIGINT: constant Signal:= implementation-defined;  
5      Signal_Kill,  
6      SIGKILL: constant Signal:= implementation-defined;  
7      Signal_Pipe_Write,  
8      SIGPIPE: constant Signal:= implementation-defined;  
9      Signal_Quit,  
10     SIGQUIT: constant Signal:= implementation-defined;  
11     Signal_Segmentation_Violation,  
12     SIGSEGV: constant Signal:= implementation-defined;  
13     Signal_Terminate,  
14     SIGTERM: constant Signal:= implementation-defined;  
15     Signal_User_1,  
16     SIGUSR1: constant Signal:= implementation-defined;  
17     Signal_User_2,  
18     SIGUSR2: constant Signal:= implementation-defined;  
19     Signal_Child,  
20     SIGCHLD: constant Signal:= implementation-defined;  
21     Signal_Continue,  
22     SIGCONT: constant Signal:= implementation-defined;  
23     Signal_Stop,  
24     SIGSTOP: constant Signal:= implementation-defined;  
25     Signal_Terminal_Stop,  
26     SIGTSTP: constant Signal:= implementation-defined;  
27     Signal_Terminal_Input,  
28     SIGTTIN: constant Signal:= implementation-defined;  
29     Signal_Terminal_Output,  
30     SIGTTOU: constant Signal:= implementation-defined;  
31     subtype Realtime_Signal is Signal range implementation-defined;
```

32 The *signals defined by this standard* are `Signal_Null`, the required signals, the job
33 control signals, the memory protection signal and the realtime signals .

34 The *required signals* are as follows:

35 `Signal_Abort`

36 This is the abnormal process termination signal. It is a reserved signal. The
37 Ada language implementation may use this signal in the implementation
38 of the Ada **abort** operation.

39 `Signal_Alarm`

40 This is the timeout signal. It is a reserved signal. The Ada language imple-
41 mentation may use this signal in the implementation of language-defined
42 operations with time-outs.

43 `Signal_Floating_Point_Error`

44 This is the signal for detection of an erroneous arithmetic operation, such
45 as division by zero or an operation resulting in overflow. It is a reserved
46 signal. The Ada language implementation may translate this signal to `Con-
47 straint_Error`.

48 `Signal_Hangup`

49 This is the signal for hangup detected on the controlling terminal or the
50 termination of the controlling process. (See §7.1.1 and §3.1.3.)

1 **Signal_Illegal_Instruction**
2 This is the signal for detection of an invalid hardware instruction. It is a re-
3 served signal. The Ada language implementation may translate this signal
4 to an exception. If `Ada.Exceptions` is supported, the value of `Exception`
5 `Message` shall be `"Signal_Illegal_Instruction"`.
6 **Signal_Interrupt**
7 This is the interactive attention signal. (See §7.2.1.)
8 **Signal_Kill**
9 This is the process termination signal. It cannot be accepted, caught, or
10 ignored.
11 **Signal_Pipe_Write**
12 This is the signal for a write operation to a pipe that has no readers. (See
13 §6.1.4.)
14 **Signal_Quit**
15 This is the interactive termination signal. (See §7.2.1.)
16 **Signal_Segmentation_Violation**
17 This is the signal for detection of an invalid memory reference. It is a re-
18 served signal. Any occurrences that are not identifiable by the Ada language
19 implementation as corresponding to checks that require some other excep-
20 tion to be raised shall be mapped to the exception `Program_Error`, with
21 `Exception_Message` `"Signal_Segmentation_Violation"`.
22 **Signal_Terminate**
23 This is a process termination signal.
24 **Signal_User_1**
25 This is set aside for the use of applications; that is, the implementation shall
26 not generate this signal unless it is explicitly sent by an application.
27 **Signal_User_2**
28 This is set aside for the use of applications; that is, the implementation shall
29 not generate this signal unless it is explicitly sent by an application.
30 The *job control signals* are
31 **Signal_Child**
32 This is the signal indicating a child process has terminated or stopped.
33 **Signal_Continue**
34 This is the signal to continue if stopped.
35 **Signal_Stop**
36 This is the signal to stop. It cannot be accepted, caught, or ignored.
37 **Signal_Terminal_Stop**
38 This is the interactive stop signal. (See §7.1.1.9.)
39 **Signal_Terminal_Input**
40 This the signal for when a read from the controlling terminal is attempted
41 by a member of the background process group. (See §7.1.1.4.)

1 **Signal_Terminal_Output**

2 This is the signal for when a write to the controlling terminal is attempted
3 by a member of the background process group. (See §7.1.1.4.)

4 The *memory protection signal* is

5 **Signal_Bus_Error**

6 This is the signal indicating access to an undefined portion of a memory
7 object. It is a reserved signal, which the Ada language implementation shall
8 translate to the exception `Program_Error`. If Ada95 (ISO/IEC 8652: 1995)
9 is supported, the value of `Exception_Message` shall be "Signal_Bus_
10 Error".

11 The *realtime signals* are those in the range of the subtype `Realtime_Signal`.

12 The required support for the realtime signals is as follows:

- 13 — All the required signals shall be supported by every implementation.
- 14 — If the Job Control option is supported, job control signals shall be supported.
- 15 — If the Memory Protection option is supported, the memory protection signal shall
16 be supported.
- 17 — If the Realtime Signals option is supported, the realtime signals shall be sup-
18 ported.

19 If the Realtime Signals option is supported, the range `Realtime_Signal` shall include
20 at least `POSIX_Limits.Portable_Realtime_Signals_Maximum` values, and shall
21 not overlap with the named signals defined in this standard. It shall specify a range
22 of signal numbers that are reserved for application use, and for which the realtime
23 signal behavior specified in this standard shall be supported.

24 If the Realtime Signals option is not supported, the range of `Realtime_Signal` is
25 unspecified.

26 The implementation may support other signals, in addition to the signals defined by
27 this standard.

28 For signals that are supported, the implementation shall behave in accordance with
29 this standard. For other signals, the implementation shall not generate these sig-
30 nals, and attempts to send these signals or to examine or specify their signal actions
31 shall raise `POSIX_Error`.

32 It is implementation defined whether the realtime signal behavior specified in this
33 standard—specifically, the queueing of signals and the passing of application defined
34 data values—is supported for signals outside of the range `Realtime_Signal`.

35 An Ada application is not permitted to explicitly accept, block, catch, or ignore any
36 of the reserved signals (see §2.2.2.117). (A process executing an Ada program may
37 accept, block, catch, or ignore the reserved signals, but only indirectly, via actions
38 of the Ada runtime system.) The effect of any of these signals being generated for
39 the process of an Ada active partition, or for any task within it, is implementation
40 defined.

1 It is not permitted to accept, catch, or ignore `Signal_Kill` or `Signal_Stop`.
2 An implementation shall not impose restrictions on the ability of an application to
3 send, accept, block, or ignore the signals defined by this standard, except as specified
4 in this standard.

5 Some of the signals defined by this standard are named by the constants of type
6 `Signal` declared in this package. The values of these constants are implementation
7 defined. No additional names are permitted to be declared in the visible part of the
8 specification of any package defined by this standard. Two names are provided for
9 each signal. The short name is defined for historical reasons; use of the long name
10 in an Ada program is preferred. `Signal_Null` is a value of type `Signal` that cannot
11 be sent, accepted, or caught. It shall have the value zero.

12 **3.3.4 Default Signal Actions**

13 The following default signal actions shall be supported:

14 Terminate:

15 Terminate the process abnormally.

16 Ignore:

17 Ignore the signal.

18 Stop:

19 Stop the process.

20 Continue:

21 Continue the process if it is currently stopped; otherwise, ignore the signal.

22 If the default action is to stop the process, the execution of that process (including all
23 tasks within it) shall be temporarily suspended. When a process stops, a `Signal_`
24 `Child` signal shall be generated for its parent process, unless the parent process
25 has disabled this feature, by calling `Set_Stopped_Child_Signal` with parameter
26 `Enable` set to `False`. (See §3.3.10.)

27 While a process is stopped, any additional signals (except `Signal_Continue`) that
28 are sent to the process shall not be delivered until the process is continued, except
29 `Signal_Kill`, which shall always terminate the receiving process. A process that is
30 a member of an orphaned process group shall not be allowed to stop in response to
31 the `Signal_Terminal_Stop`, `Signal_Terminal_Input`, or `Signal_Terminal_`
32 `Output` signals. In cases where delivery of one of these signals would stop such a
33 process, the signal shall be discarded.

34 The default signal action for all the required signals defined by this standard that
35 are not reserved signals shall be to terminate the process abnormally. Terminating
36 the process associated with an Ada active partition shall cause the active partition
37 and all the tasks within it to cease execution permanently.

38 The default action for the reserved signals is implementation defined.

39 The default actions for the job-control signals shall be as shown in Table 3.1.

40 The default signal actions for the realtime signals shall be to terminate the process
41 abnormally.

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

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 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 Ada RM {1}.

Signals that are ignored shall not affect the behavior of any operation.

1 Signals that are blocked shall not affect the behavior of any operation until they are
2 delivered.

3 The interruptible operations defined in this standard are:

- 4 — POSIX_Asynchronous_IO.Await_IO_Or_Timeout
- 5 — POSIX_Asynchronous_IO.Await_IO
- 6 — POSIX_Asynchronous_IO.List_IO_Wait
- 7 — POSIX_File_Locking.Wait_to_Set_Lock
- 8 — POSIX_Generic_Shared_Memory.Open_And_Map_Shared_Memory
- 9 — POSIX_Generic_Shared_Memory.Open_Or_Create_And_Map_Shared_Memory
- 10 — POSIX_IO.Open_Or_Create
- 11 — POSIX_IO.Open
- 12 — POSIX_IO.Duplicate
- 13 — POSIX_IO.Close
- 14 — POSIX_IO.Read
- 15 — POSIX_IO.Write
- 16 — POSIX_Message_Queues.Generic_Message_Queues.Send
- 17 — POSIX_Message_Queues.Generic_Message_Queues.Receive
- 18 — POSIX_Message_Queues.Open
- 19 — POSIX_Message_Queues.Open_Or_Create
- 20 — POSIX_Message_Queues.Send
- 21 — POSIX_Message_Queues.Receive
- 22 — POSIX_Process_Primitives.Wait_For_Child_Process
- 23 — POSIX_Semaphores.Wait (optionally interruptible)
- 24 — POSIX_Shared_Memory_Objects.Open_Or_Create_Shared_Memory
- 25 — POSIX_Shared_Memory_Objects.Open_Shared_Memory
- 26 — POSIX_Signals.Await_Signal
- 27 — POSIX_Signals.Await_Signal_Or_Timeout
- 28 — POSIX_Terminal_Functions.Set_Terminal_Characteristics
- 29 — POSIX_Terminal_Functions.Drain

30 Return from a call to an interruptible POSIX operation, whether by normal return
31 or exception propagation, shall be an abort completion point. [Ada RM {1} 9.8 (15)]

32 While a task is blocked in an interruptible POSIX operation with the Signal_-
33 Masking value of No_Signals or RTS_Signals, abort of the task shall cause the
34 operation to be interrupted.

35 NOTE: A consequence of these rules is that the task must reach an abort completion
36 point, since interruption is required to raise POSIX_Error, and propagation of the ex-
37 ception is an abort completion point. Whether the task is aborted at that point depends
38 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: Since the application cannot explicitly block or ignore the reserved signals, if `Masked_Signals` is `No_Signals` the operation may be interrupted, and so the application should 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.

1 NOTE: `Add_All_Signals` adds all the implementation defined signals and `Delete_`
2 `All_Signals` deletes all implementation defined signals, as well as all the signals de-
3 fined in this standard.

4 `Is_Member` shall return the value `True` if and only if the set specified by the pa-
5 rameter `Set` includes the signal specified by the parameter `Sig` or the value of the
6 parameter `Sig` is `Signal_Null`.

7 **3.3.7.3 Error Handling**

8 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
9 with the corresponding error code:

10 `Invalid_Argument`

11 `Add_Signal`, `Delete_Signal`, or `Is_Member` was called with a value of
12 `Sig` that is not a valid signal in this implementation.

13 `Delete_All_Signals` and `Add_All_Signals` shall not raise any exception.

14 **3.3.8 Block and Unblock Signals**

15 **3.3.8.1 Synopsis**

```
16      procedure Set_Blocked_Signals
17          (New_Mask: in Signal_Set;
18          Old_Mask: out Signal_Set);
19      procedure Block_Signals
20          (Mask_to_Add: in Signal_Set;
21          Old_Mask: out Signal_Set);
22      procedure Unblock_Signals
23          (Mask_to_Subtract: in Signal_Set;
24          Old_Mask: out Signal_Set);
25      function Blocked_Signals return Signal_Set;
```

26 **3.3.8.2 Description**

27 Operations are provided to change and inquire about the signal mask of the calling
28 task.

29 NOTE: The main intended use of these operations is with the default signal action, in
30 which case the signal should be kept blocked in all but one task; blocking and unblocking
31 the signal in that one task will control when the default action may occur (for the whole
32 process). In contrast, it is recommended that applications keep all signals that are used
33 with signal-awaiting operations blocked in all tasks at all times.

34 The `Set_Blocked_Signals` operation shall replace the specified signal mask by
35 the parameter `New_Mask`, and return in the parameter `Old_Mask` the value that is
36 replaced. The operation shall be atomic, even in the presence of multiple Ada tasks.

37 The `Block_Signals` operation shall add the signals specified by the parameter
38 `Mask_to_Add` to the current signal mask, and return in the parameter `Old_Mask`
39 the current mask before this operation. The operation shall be atomic, even in the
40 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;
```

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 §2.2.2.117).

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.

3.3.9.3 Error Handling

If any of the following conditions occur, 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 other means 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 either for 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 and if queueing is enabled for that

1 signal, then the signal will be queued to the process and the Data attribute
2 of the event shall be the Data attribute of the generated signal occurrence. If
3 queueing is not enabled for that signal, it is unspecified whether the signal
4 is queued and what Data attribute, if any, is sent.

5 An implementation may add notification mechanisms to the `Notification` type, as
6 permitted in §1.3.1.1, point (2). Such extensions, which may change the behavior of
7 the application with respect to this standard when those attributes are uninitialized,
8 also require that the extension be enabled as required by §1.3.1.1.

9 The `Signal_Event` type is used to specify how notification of an event is to be deliv-
10 ered. A value of this type has (at least) the following attributes:

11 Notification

12 The notification mechanism to use. This is a value of type `Notification`.

13 Signal

14 The signal to be generated, if the notification mechanism specified makes
15 use of a signal.

16 Data

17 The information to be sent with the signal, if the signal is one for which
18 signal queueing is supported and enabled. The Data value is the application
19 defined data value to be passed as part of the `Signal_Info` return value of
20 one the `Await_Signal` functions (See §3.3.14.)

21 The type `Signal_Data` is used as a carrier for untyped information. The size of
22 this type shall be large enough to hold any value of type `Integer`, or type `System.-`
23 `Address`. Instantiation of `Unchecked_Conversion` shall be supported for conver-
24 sions between the type `Signal_Data` and the types `System.Address`, `Standard.-`
25 `Integer`, and `POSIX_Timers.Timer_ID`. Unchecked conversion of a value of one of
26 the latter types to `Signal_Data` and then back to the original type shall yield the
27 original value back again.

28 The function whose name is of the form `Get_X`, where X is the name of one of the at-
29 tributes of `Signal_Event`, shall return the value of the X attribute of the argument.

30 The procedure whose name is of the form `Set_X` shall set the X attribute of the object
31 specified by the first argument to the value specified by the second argument.

32 **3.3.12.3 Error Handling**

33 No error conditions are specified for these operations.

34 **3.3.13 Signal Information**

35 **3.3.13.1 Synopsis**

```
36    type Signal_Source is range implementation-defined;  
37    From_Send_Signal:   constant Signal_Source:= implementation-defined;  
38    From_Queue_Signal:  constant Signal_Source:= implementation-defined;  
39    From_Timer:         constant Signal_Source:= implementation-defined;  
40    From_Async_IO:      constant Signal_Source:= implementation-defined;  
41    From_Message_Queue: constant Signal_Source:= implementation-defined;
```

```

1  type Signal_Info is private;
2  function Get_Signal (Info: Signal_Info) return Signal;
3  procedure Set_Signal
4      (Info: in out Signal_Info;
5       Sig: in Signal);
6  function Get_Signal_Source (Info: Signal_Info) return Signal_Source;
7  procedure Set_Signal_Source
8      (Info: in out Signal_Info;
9       Source: in Signal_Source);
10 function Has_Data (Source: Signal_Source) return Boolean;
11 function Get_Data (Info: Signal_Info) return Signal_Data;
12 procedure Set_Data
13     (Info: in out Signal_Info;
14      Data: in Signal_Data);

```

15 3.3.13.2 Description

16 *The functionality described in this subclause is optional. If the Realtime Signals option*
 17 *is not supported, the implementation may cause all calls to the explicitly declared*
 18 *operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior*
 19 *shall be as specified below.*

20 Values of Signal_Source are used to identify the cause, or source of a signal occur-
 21 rence. The following constants of type Signal_Source are defined by this standard,
 22 with the indicated specifications:

23 From_Send_Signal
 24 the signal was generated by the Send_Signal procedure.

25 From_Queue_Signal
 26 the signal was generated by the Queue_Signal procedure.

27 From_Timer
 28 the signal was generated by the expiration of a timer (see package POSIX_-
 29 Timers).

30 From_Async_IO
 31 the signal was generated by the completion of an asynchronous I/O request
 32 (see package POSIX_Asynynchronous_IO).

33 From_Message_Queue
 34 the signal was generated by the arrival of a message on an empty message
 35 queue, for which notification via signal was requested (see package POSIX_-
 36 Message_Queue).

37 The Signal_Info type is used to specify information that can be associated with an
 38 occurrence of a signal. It has the following attributes:

39 Signal
 40 The signal of which this is an occurrence.

41 Source
 42 The cause, or source, that generated the signal. If the signal was generated
 43 by one of the operations or events listed above, the value of Source shall be
 44 as specified for that operation or event. If the signal was not generated by
 45 one of the operations or events listed above, the Source attribute of a signal
 46 occurrence shall be an implementation defined value that is not equal to any
 47 of the values defined above.

1 Data

2 Additional information that is provided for certain signal sources. If Source

3 is equal to one of From_Queue_Signal, From_Timer, From_Async_IO, or

4 From_Message_Queue, Data shall be the application specified signal data

5 value. Otherwise, the Data attributed is undefined.

6 An implementation may add other attributes to the Signal_Info type, as permitted

7 in §1.3.1.1, point (2). Such extensions, which may change the behavior of the appli-

8 cation with respect to this standard when those attributes are uninitialized, also

9 requires that the extension be enabled as required by §1.3.1.1.

10 The function whose name is of the form Get_X, where X is the name of one of the

11 attributes of Signal_Info, shall return the value of the X attribute of the argument.

12 The procedure whose name is of the form Set_X shall set the X attribute of the object

13 specified by the first argument to the value specified by the second argument.

14 Has_Data shall return True if and only if Source is a signal source for which the

15 Data attribute is defined.

16 **3.3.13.3 Error Handling**

17 No error conditions are specified for these operations.

18 **3.3.14 Control Signal Queueing**

19 **3.3.14.1 Synopsis**

20 **procedure** Enable_Queueing (Sig: in Signal);

21 **procedure** Disable_Queueing (Sig: in Signal);

22 **3.3.14.2 Description**

23 *The functionality described in this subclause is optional. If the Realtime Signals option*

24 *is not supported, the implementation may cause all calls to the explicitly declared*

25 *operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior*

26 *shall be as specified below.*

27 Enable_Queueing and Disable_Queueing are used to control whether data is or

28 is not queued for delivery with the designated signal, by those functions which are

29 able to do so. (See Queue_Signal §3.3.19, Request_Notify §15.1.8, Create_Timer

30 §14.1.5, and the asynchronous I/O operations §6.3.)

31 Enable_Queueing shall enable queueing for the specified signal.

32 Disable_Queueing shall disable queueing for the specified signal. It is unspecified

33 whether occurrences already queued for the signal are retained.

34 If queueing is enabled for the signal, a signal-awaiting operation for the signal shall

35 return a value of type Signal_Info, which is used to pass information about the

36 event associated with a signal occurrence.

37 The initial state of signal queueing for all signals in a new process is unspecified.

3.3.14.3 Error Handling

If any of the following conditions occur, 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 system's set of pending signals, 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 xreturns 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 which is caught by an asynchronous signal handler of the Ada language implementation, and this 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:

— When multiple signals in the range `Realtime_Signal` are pending, should any of them be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime 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 occur, 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:

1 **Invalid_Argument**
2 Set contains an invalid, or unsupported signal number.
3 The `Timeout` argument specified an invalid value (e.g., uninitialized data).
4 An implementation should only check for this error if no signal is pending
5 in `Set` and it is necessary to wait.

6 **3.3.16 Wait for Signal with Information**

7 **3.3.16.1 Synopsis**

```
8       function Await_Signal (Set: Signal_Set) return Signal_Info;  
9       function Await_Signal_Or_Timeout  
10          (Set:       Signal_Set;  
11          Timeout: POSIX.Timespec)  
12       return Signal_Info;
```

13 **3.3.16.2 Description**

14 *The functionality described in this subclause is optional. If the Realtime Signals option*
15 *is not supported, the implementation may cause all calls to the explicitly declared*
16 *operations defined in this subclause to raise `POSIX_Error`. Otherwise, the behavior*
17 *shall be as specified below.*

18 The effects of these signal-awaiting operations are like those of the signal-awaiting
19 operations defined in §3.3.15, except:

- 20 — The `Signal` attribute of the value returned by the signal-awaiting operation shall
21 be the selected signal.
- 22 — The `Source` attribute of the value returned by the signal-awaiting operation shall
23 be the cause of the signal.
- 24 — If any occurrence of the selected signal is queued, the first such occurrence shall
25 be dequeued and the `Data` attribute shall be returned as the `Data` attribute of
26 the value returned by the function. Otherwise, the `Data` attribute of the value
27 returned is undefined.

28 NOTE: If the signal was sent by one of the signal-queueing operations defined in this
29 standard and `Has_Data(Get_Source(Info))` is `True`, the `Data` attribute of `Info` will
30 generally be defined; otherwise, the recipient of the signal must rely on application-
31 specific conventions established between the sender and recipient of the signal.

32 **3.3.16.3 Error Handling**

33 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
34 with the corresponding error code:

35 **Invalid_Argument**
36 At least one of the signals in `Set` is attached to a task entry, or is a reserved
37 signal.

38 **Operation_Not_Implemented**
39 The functions `Await_Signal` and `Await_Signal_Or_Timeout` are not
40 supported by this implementation.

1 **Resource_Temporarily_Unavailable**
2 No signal specified by **Set** was delivered within the specified timeout period.

3 If any of the following conditions is detected, the exception **POSIX_Error** shall be
4 raised, with the corresponding error code:

5 **Invalid_Argument**
6 **Set** contains an invalid or unsupported signal number.
7 The **Timeout** argument specifies an invalid value (e.g., uninitialized data).
8 An implementation should only check for this error if no signal is pending
9 in **Set** and it is necessary to wait.

10 **3.3.17 Signal Entries**

11 **3.3.17.1 Synopsis**

```
12       -- obsolescent
13       Signal_Abort_Ref: constant System.Address:= implementation-defined;
14       -- Signal_Floating_Point_Error intentionally omitted.
15       Signal_Hangup_Ref: constant System.Address:= implementation-defined;
16       -- Signal_Illegal_Instruction intentionally omitted.
17       Signal_Interrupt_Ref: constant System.Address:= implementation-defined;
18       -- Signal_Kill intentionally omitted.
19       Signal_Pipe_Write_Ref: constant System.Address:= implementation-defined;
20       Signal_Quit_Ref: constant System.Address:= implementation-defined;
21       Signal_Terminate_Ref: constant System.Address:= implementation-defined;
22       Signal_User_1_Ref: constant System.Address:= implementation-defined;
23       Signal_User_2_Ref: constant System.Address:= implementation-defined;
24       Signal_Child_Ref: constant System.Address:= implementation-defined;
25       Signal_Continue_Ref: constant System.Address:= implementation-defined;
26       -- Signal_Stop intentionally omitted.
27       Signal_Terminal_Stop_Ref: constant System.Address:= implementation-defined;
28       Signal_Terminal_Input_Ref: constant System.Address:= implementation-defined;
29       Signal_Terminal_Output_Ref: constant System.Address:= implementation-defined;
30       function Signal_Reference (Sig: Signal) return System.Address;
```

31 **3.3.17.2 Description**

32 *The functionality described in this subclause is optional. If the Signal Entries option*
33 *is not supported, the implementation may cause all calls to the explicitly declared*
34 *operations defined in this subclause to raise POSIX_Error. Otherwise, the behavior*
35 *shall be as specified below.*

36 The semantic model for accepting a signal via a task entry is based on the concept of
37 interrupt entry, as defined in Annex I.7.1 of Ada RM {1} (where it is classified as an
38 obsolescent feature of the Ada language) and is described further below. The task
39 accepting the signal views it as an interrupt entry call.

40 The binding of a signal to an entry of a task shall be by means of an Ada address
41 clause. In order to permit this, there shall be a means for obtaining a value of type
42 **System.Address** corresponding to each value of type **Signal** that can be accepted
43 by an Ada application.

44 The constants defined by this standard with names of the form **Signal_XXX_Ref**
45 shall be usable in an address clause that binds the signal with name **Signal_XXX** to

1 an entry. In addition, for all signals, the function `Signal_Reference` shall return a
2 value that can be used in an address clause to bind a signal entry for the parameter
3 `Sig`. If `Signal_XXX` is the name of a signal declared in `POSIX_Signals`, then
4 `Signal_Reference(Signal_XXX) = Signal_XXX_Ref`.

5 For a signal that is bound to an entry of a task, the effect shall be as if there were
6 a virtual task that uses `Await_Signal` (see §3.3.15) to accept the signal, attempts
7 to rendezvous with the associated entry each time an occurrence of the signal is
8 accepted, and is ready to accept the signal again after the rendezvous. This virtual
9 task shall behave as if it has priority at least as high as that of the task to whose
10 entry the signal is bound.

11 There may be an implementation defined limit on the number of these virtual tasks
12 that can simultaneously exist, but there must be at least one for each signal that can
13 be accepted or caught. Even if a limit greater than one is provided, it is unspecified
14 whether a generated signal results in a new entry call and whether such delivery
15 results in another queued virtual task if there is at least one virtual task with an
16 entry call already queued for that signal.

17 NOTE: This means it is unspecified whether multiple generations of a signal can cause
18 the value of the 'Count' attribute of the associated entry to exceed one.

19 NOTE: A significant semantic change was made here between POSIX.5 {B3} and PO-
20 SIX.5b {3} in the effect of the signal mask on signal entries. The signal mask can no
21 longer prevent delivery of the signal to a task with an open **accept** statement for the
22 corresponding entry. This change was forced by POSIX.1c {4}.

23 Clearing pending occurrences of a given signal that is bound to a task entry may
24 abort any entry calls from virtual tasks that are due to previously accepted occur-
25 rences of that signal. This shall have no effect on any calls from normal Ada tasks
26 that may be queued for the entry.

27 Given the virtual task model described above, other details of the semantics of signal
28 accepting via task entries are determined by the Ada language standard. For exam-
29 ple, the times at which entries are bound to signals are defined by the semantics
30 of Ada task elaboration, and the conditions under which a signal entry call may be
31 accepted are defined by the semantics of rendezvous.

32 One restriction imposed by Ada RM {1} is that a task should not attempt to bind
33 an entry to a signal to which there is already another bound entry. If a program
34 attempts to do this, `Program_Error` shall be raised. However, on termination of a
35 task, the bindings of any of its entries to interrupts shall be broken so that these
36 interrupts shall become available for rebinding. When the binding is broken, the
37 signal action shall revert to the default for that signal.

38 For the `Ignore_Signal` operation if the signal is bound to a task entry, the effect
39 shall be to discard any pending or subsequent deliveries of that signal. The binding
40 to the entry may remain in force. Any queued signal entry calls to that entry may be
41 discarded. (The discarding of pending signal entry calls may depend on whether the
42 signal is blocked – see §3.3.3.)

43 NOTE: For the `Await_Signal` and `Await_Signal_Or_Timeout` operations, if the
44 parameter `Set` includes a signal to which an interrupt entry is attached, `POSIX_Error`
45 shall be raised, with error code `Invalid_Argument`. (See §3.3.15)

3.3.17.3 Error Handling

If any of the following conditions occur, 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, and 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 occur, 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 below.

`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, and 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 occur, 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 `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 Ada RM {1} C.7.1 (17) for some uses of invalid values of type `Task_ID`.

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Section 4: Process Environment

1 The environment for a POSIX process provides information about the system state
2 from which the process was started. This section specifies operations that access
3 this information, as defined in the packages `POSIX_Process_Identification`,
4 `POSIX_Process_Times`, `POSIX_Process_Environment`, `POSIX_Calendar`, and
5 `POSIX_Configurable_System_Limits`.

6 4.1 Package `POSIX_Process_Identification`

7 This package contains operations to

- 8 — Identify self and parent processes
- 9 — Set and get the process group ID
- 10 — Set the process group ID for job control and create a session
- 11 — Get the real and the effective identity of the user and the group of the user
- 12 — Get a list of the supplementary groups of the user
- 13 — Get the effective user login name

```
14 with POSIX;  
15 package POSIX_Process_Identification is  
16   -- 4.1.1 Process Identification Operations  
17   type Process_ID is private;  
18   Null_Process_ID: constant Process_ID;  
19   System_Process_ID: constant Process_ID;  
20   function Get_Process_ID return Process_ID;  
21   function Get_Parent_Process_ID return Process_ID;  
22   function Image (ID: Process_ID) return Standard.String;  
23   function Value (Str: Standard.String) return Process_ID;  
24   -- 4.1.2 Process Group Identification  
25   type Process_Group_ID is private;  
26   function Get_Process_Group_ID return Process_Group_ID;  
27   procedure Set_Process_Group_ID  
28     (Process: in Process_ID:= Get_Process_ID;  
29      Process_Group: in Process_Group_ID:= Get_Process_Group_ID);  
30   procedure Create_Process_Group  
31     (Process: in Process_ID;  
32      Process_Group: out Process_Group_ID);  
33   procedure Create_Session (Session_Leader: out Process_Group_ID);  
34   function Image (ID: Process_Group_ID) return Standard.String;  
35   function Value (Str: Standard.String) return Process_Group_ID;  
36   -- 4.1.3 User Identification  
37   type User_ID is private;  
38   function Get_Real_User_ID return User_ID;  
39   function Get_Effective_User_ID return User_ID;  
40   procedure Set_User_ID (ID: in User_ID);  
41   function Get_Login_Name return POSIX.POSIX_String;  
42   function Image (ID: User_ID) return Standard.String;  
43   function Value (Str: Standard.String) return User_ID;
```

```

1      -- 4.1.4 User and Group Identification
2      type Group_ID is private;
3      function Get_Real_Group_ID return Group_ID;
4      function Get_Effective_Group_ID return Group_ID;
5      procedure Set_Group_ID (ID: in Group_ID);
6      subtype Group_List_Index is positive range 1..POSIX.Groups_Maxima'Last;
7      type Group_List is array (Group_List_Index range <>) of Group_ID;
8      function Get_Groups return Group_List;
9      function Image (ID: Group_ID) return Standard.String;
10     function Value (Str: Standard.String) return Group_ID;
11 private
12     implementation-defined
13 end POSIX_Process_Identification;

```

14 4.1.1 Process Identification Operations

15 4.1.1.1 Synopsis

```

16     type Process_ID is private;
17     Null_Process_ID: constant Process_ID;
18     System_Process_ID: constant Process_ID;
19     function Get_Process_ID return Process_ID;
20     function Get_Parent_Process_ID return Process_ID;
21     function Image (ID: Process_ID) return Standard.String;
22     function Value (Str: Standard.String) return Process_ID;

```

23 4.1.1.2 Description

24 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.

29 `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.

31 `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.

35 `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`.

39 4.1.1.3 Error Handling

40 `Constraint_Error` shall be raised by `Value` if the parameter `Str` cannot be successfully translated to a `Process_ID`.

42 No exceptions are specified by this standard for `Get_Process_ID` and `Get_Parent_Process_ID`.

1 No exception shall be raised by Image.

2 **4.1.2 Process Group Identification**

3 **4.1.2.1 Synopsis**

```
4     type Process_Group_ID is private;  
5     function Get_Process_Group_ID return Process_Group_ID;  
6     procedure Set_Process_Group_ID  
7         (Process:      in Process_ID:= Get_Process_ID;  
8          Process_Group: in Process_Group_ID:= Get_Process_Group_ID);  
9     procedure Create_Process_Group  
10        (Process:      in Process_ID;  
11         Process_Group: out Process_Group_ID);  
12     procedure Create_Session (Session_Leader: out Process_Group_ID);  
13     function Image (ID: Process_Group_ID) return Standard.String;  
14     function Value (Str: Standard.String) return Process_Group_ID;
```

15 **4.1.2.2 Description**

16 The type Process_Group_ID shall define the values for *process group IDs*.

17 Get_Process_Group_ID shall return the process group ID of the calling process.

18 The effect of Set_Process_Group_ID depends on whether the Job Control option is
19 supported. *If the Job Control option is supported:*

20 Set_Process_Group_ID is used to join an existing process group. Upon success-
21 ful completion, the process group ID of the process whose process ID matches the
22 parameter Process shall be the value of the parameter Process_Group.

23 *Otherwise:*

24 Set_Process_Group_ID shall raise the exception POSIX_Error with error code
25 Operation_Not_Implemented.

26 The effect of Create_Process_Group depends on whether the Job Control option is
27 supported. *If the Job Control option is supported:*

28 Create_Process_Group shall make the process specified by Process a process
29 group leader. If the specified process was not previously a process group leader,
30 a new process group is created within the session of the calling process with the
31 specified process as the process group leader. The process group ID of a session
32 leader shall not change. If the specified process is already a process group leader,
33 the operation completes successfully without any change. Upon successful com-
34 pletion, the process group ID of the process specified by the parameter Process
35 shall be returned as the value of the parameter Process_Group.

36 *Otherwise:*

37 Create_Process_Group shall raise the exception POSIX.POSIX_Error with
38 error code Operation_Not_Implemented.

39 Create_Session shall create a new session if the calling process is not a process
40 group leader. If a new session is created, the calling process shall

41 — Be the session leader of the new session

- 1 — Be the process group leader of a new process group
 - 2 — Have no controlling terminal
 - 3 — Be the only process in the new process group
 - 4 — Be the only process in the new session
- 5 Upon successful completion, `Create_Session` shall return the new process group
6 ID of the calling process.

7 `Image` shall return a character string representing the parameter ID. The string
8 representation is implementation defined, but it shall be such that if two process
9 group IDs are different, then their images are different.

10 `Value` translates the parameter `Str` into a `Process_Group_ID`. `Value` shall trans-
11 late any string produced by `Image` into the `Process_Group_ID` value that was origi-
12 nally the input parameter to `Image`. It is implementation defined whether any other
13 string representation may be successfully translated into a value of type `Process_`-
14 `Group_ID`.

15 **4.1.2.3 Error Handling**

16 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
17 with the corresponding error code:

18 **Invalid_Argument**

19 The value for the parameter `Process` or `Process_Group` for `Set_Pro-`
20 `cess_Group_ID` or `Create_Process_Group` is a value not supported by
21 this implementation.

22 **Permission_Denied**

23 The parameter `Process` of the procedure `Set_Process_Group_ID` or `Cre-`
24 `ate_Process_Group` matches the process ID of a child process of the call-
25 ing process, and the child process has successfully executed a call to `Exec`,
26 `Exec_Search`, `Start_Process`, or `Start_Process_Search`.

27 **Operation_Not_Implemented**

28 `Set_Process_Group_ID` or `Create_Process_Group` was called, but it is
29 not supported by this implementation.

30 **No_Such_Process**

31 The parameter `Process` of the procedure `Set_Process_Group_ID` or `Cre-`
32 `ate_Process_Group` does not match the process ID of either the calling
33 process or of a child process of the calling process.

34 **Operation_Not_Permitted**

- 35 (1) The process indicated by the parameter `Process` of `Set_Process_`-
36 `Group_ID` is a session leader or is a child process of the calling process,
37 but that child process is not in the same session as the calling process.
- 38 (2) The process group indicated by the parameter `Process_Group` of the
39 procedure `Set_Process_Group_ID` or `Create_Process_Group` does
40 not match the process group ID of the indicated process, and there is no
41 process with that process group ID that is in the same session as the
42 calling process.

- 1 (3) The process calling `Create_Session` is already a process group leader.
- 2 `Constraint_Error` shall be raised if the parameter `Str` to `Value` cannot be suc-
- 3 cessfully translated to a `Process_Group_ID`.
- 4 No exceptions are specified by this standard for `Get_Process_Group_ID`.
- 5 No exception shall be raised by `Image`.

6 **4.1.3 User Identification**

7 **4.1.3.1 Synopsis**

```
8     type User_ID is private;  
9     function Get_Real_User_ID return User_ID;  
10    function Get_Effective_User_ID return User_ID;  
11    procedure Set_User_ID (ID: in User_ID);  
12    function Get_Login_Name return POSIX.POSIX_String;  
13    function Image (ID: User_ID) return Standard.String;  
14    function Value (Str: Standard.String) return User_ID;
```

15 **4.1.3.2 Description**

16 The type `User_ID` shall identify users of the system. The concepts of user ID, real
17 user ID, effective user ID, and saved set-user-ID are defined in §2.2.2.

18 The `Get_Real_User_ID` function shall return the real user ID of the calling process.

19 The `Get_Effective_User_ID` function shall return the effective user ID of the
20 calling process.

21 The effect of `Set_User_ID` depends on whether the Saved IDs option is supported. *If*
22 *the Saved IDs option is supported:*

23 If the calling process has appropriate privileges, `Set_User_ID` shall set the real
24 user ID, effective user ID, and saved set-user-ID to the value of the parameter
25 ID. If the calling process does not have appropriate privileges, but the parameter
26 ID is equal to the real user ID or the saved set-user-ID, `Set_User_ID` shall set
27 the effective user ID to the value of the parameter ID; the real user ID and the
28 saved set-user-ID shall remain unchanged.

29 *Otherwise:*

30 If the process has appropriate privileges, `Set_User_ID` shall set the real user ID
31 and the effective user ID to the value of the parameter ID. If the process does not
32 have appropriate privileges, but the parameter ID is equal to the real user ID,
33 `Set_User_ID` shall change the effective user ID to the value of the parameter
34 ID; the real user ID shall remain unchanged.

35 `Get_Login_Name` shall return a value of type `POSIX.POSIX_String` containing the
36 name associated by the login activity with the controlling terminal for the calling
37 process. If the login name of the user cannot be found, this function shall return a
38 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 occur, 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.

1 `Get_Effective_Group_ID` shall return the effective group ID of the calling process.

2 `Get_Groups` shall return a list containing the supplementary group IDs of the calling
3 process. This standard does not specify whether the effective group ID of the calling
4 process is included in or omitted from the returned list of supplementary group IDs.

5 `Image` shall return a character string representing the parameter ID. The string
6 representation is implementation defined, but shall be such that if two group IDs
7 are different, then their images are different.

8 `Value` translates the parameter `Str` into a `Group_ID`. `Value` shall translate any
9 string produced by `Image` into the `Group_ID` value that was originally the input
10 parameter to `Image`. It is implementation defined whether any other string repre-
11 sentation may be successfully translated into a `Group_ID`.

12 The effect of `Set_Group_ID` depends on whether the Saved IDs option is supported.
13 *If the Saved IDs option is supported:*

14 If the calling process has appropriate privileges, `Set_Group_ID` shall set the
15 real group ID, effective group ID, and saved set-group-ID to the value of the
16 parameter ID. If the calling process does not have appropriate privileges, but
17 the parameter ID is equal to the real group ID or the saved set-group-ID, `Set_`
18 `Group_ID` shall set the effective group ID to the parameter ID; the real group
19 ID and the saved set-group-ID shall remain unchanged.

20 *Otherwise:*

21 If the process has appropriate privileges, `Set_Group_ID` shall set the real group
22 ID and the effective group ID to the value of the parameter ID. If the process
23 does not have appropriate privileges, but the parameter ID is equal to the real
24 group ID, `Set_Group_ID` shall change the effective group ID to the value of the
25 parameter ID; the real group ID shall remain unchanged.

26 **4.1.4.3 Error Handling**

27 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
28 with the corresponding error code:

29 `Invalid_Argument`

30 The parameter ID to `Set_Group_ID` is not a valid group ID supported by
31 the implementation.

32 `Operation_Not_Permitted`

33 The process calling `Set_Group_ID` does not have appropriate privileges,
34 the parameter ID does not equal the real group ID, and either the Saved
35 IDs option is not supported or the parameter ID does not equal the saved
36 set-group-ID.

37 `Constraint_Error` shall be raised if the parameter `Str` to `Value` cannot be inter-
38 preted as a group ID.

39 No exceptions are specified by this standard for `Get_Real_Group_ID`, `Get_Effec-`
40 `tive_Group_ID`, and `Get_Groups`.

41 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.

1 **Elapsed_Real_Time** shall return the elapsed real time (“wall clock time”), in ticks,
2 between the start of the program and the time of its call.

3 NOTE: Though this is called “real time”, it is not sufficiently accurate or reliable for
4 use as a time base in realtime applications. Instead, see the realtime clock and timer
5 facilities which are defined in Section 14.

6 **Get_Process_Times** shall return a value of type **Process_Times** containing time
7 information about the current process at the time of the call. This value may then be
8 inspected with **Elapsed_Real_Time_Of**, **User_CPU_Time_Of**, **System_CPU_Time_**
9 **Of**, **Descendants_User_CPU_Time_Of**, and **Descendants_System_CPU_Time_Of**
10 to extract specific time values. All time values shall be of the type **Tick_Count**.

11 **Elapsed_Real_Time_Of** shall return the elapsed real time (“wall clock time”), in
12 ticks, between the start of the program and the time of its call as recorded in the
13 parameter **Times**.

14 **User_CPU_Time_Of** shall return the CPU time used by the process as recorded in
15 the parameter **Times**.

16 **System_CPU_Time_Of** shall return the CPU time used by the system on behalf of
17 the process as recorded in the parameter **Times**.

18 **Descendants_User_CPU_Time_Of** shall return the sum of the values that would be
19 returned by **User_CPU_Time_Of** (**Times**) and **Descendants_User_CPU_Time_Of**
20 (**Times**) for all terminated child processes for which **Wait_For_Child_Process** has
21 returned the child **Process_ID**. If a child process has not waited for its terminated
22 children, their times are not included in its **Descendants_User_CPU_Time_Of** and
23 thus are not included in this sum.

24 **Descendants_System_CPU_Time_Of** shall return the sum of the values that would
25 be returned by **System_CPU_Time_of** (**Times**) and **Descendants_System_CPU_**
26 **Time_Of** (**Times**) for all terminated child processes for which **Wait_For_Child_**
27 **Process** has returned the child **Process_ID**. If a child process has not waited for
28 its terminated children, their times are not included in its **Descendants_System_**
29 **CPU_Time_Of** and thus are not included in this sum.

30 **4.2.1.3 Error Handling**

31 No exceptions are specified by this standard for **Elapsed_Real_Time_Of**, **User_**
32 **CPU_Time_Of**, **System_CPU_Time_Of**, **Descendants_User_CPU_Time_Of**, or **De-**
33 **scendants_System_CPU_Time_Of**.

34 No exceptions are specified by this standard for **Elapsed_Real_Time** or **Get_Pro-**
35 **cess_Times**. Calling **Elapsed_Real_Time** or **Get_Process_Times** may raise **PO-**
36 **SIX_Error** with implementation defined error codes implementation defined condi-
37 tions.

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

```

15 with POSIX;
16 package POSIX_Process_Environment is
17   -- 4.3.1 Argument List
18   function Argument_List return POSIX.POSIX_String_List;
19   -- 4.3.2 Environment Variables
20   type Environment is limited private;
21   procedure Copy_From_Current_Environment
22     (Env: in out Environment);
23   procedure Copy_To_Current_Environment
24     (Env: in Environment);
25   procedure Copy_Environment
26     (Source: in Environment;
27      Target: in out Environment);
28   function Environment_Value_Of
29     (Name: POSIX.POSIX_String;
30      Env: Environment;
31      Undefined: POSIX.POSIX_String:= "" )
32     return POSIX.POSIX_String;
33   function Environment_Value_Of
34     (Name: POSIX.POSIX_String;
35      Undefined: POSIX.POSIX_String:= "" )
36     return POSIX.POSIX_String;
37   function Is_Environment_Variable
38     (Name: POSIX.POSIX_String;
39      Env: Environment)
40     return Boolean;
41   function Is_Environment_Variable
42     (Name: POSIX.POSIX_String)
43     return Boolean;
44   procedure Clear_Environment
45     (Env: in out Environment);
46   procedure Clear_Environment;
47   procedure Set_Environment_Variable
48     (Name: in POSIX.POSIX_String;
49      Value: in POSIX.POSIX_String;
50      Env: in out Environment);

```

```

1      procedure Set_Environment_Variable
2          (Name: in POSIX.POSIX_String;
3            Value: in POSIX.POSIX_String);
4      procedure Delete_Environment_Variable
5          (Name: in POSIX.POSIX_String;
6            Env: in out Environment);
7      procedure Delete_Environment_Variable
8          (Name: in POSIX.POSIX_String);
9      function Length (Env: Environment) return Natural;
10     function Length return Natural;
11     generic
12         with procedure Action
13             (Name: in POSIX.POSIX_String;
14              Value: in POSIX.POSIX_String;
15              Quit: in out Boolean);
16     procedure For_Every_Environment_Variable
17         (Env: in Environment);
18     generic
19         with procedure Action
20             (Name: in POSIX.POSIX_String;
21              Value: in POSIX.POSIX_String;
22              Quit: in out Boolean);
23     procedure For_Every_Current_Environment_Variable;
24     -- 4.3.3 Process Working Directory
25     procedure Change_Working_Directory
26         (Directory_Name: in POSIX.Pathname);
27     function Get_Working_Directory return POSIX.Pathname;
28 private
29     type Environment is implementation-defined;
30 end POSIX_Process_Environment;

```

31 4.3.1 Argument List

32 4.3.1.1 Synopsis

```
33     function Argument_List return POSIX.POSIX_String_List;
```

34 4.3.1.2 Description

35 Argument_List shall return the argument list of the current process as a value of
36 type POSIX.POSIX_String_List. If the current process was created by a Start_
37 Process, Start_Process_Search, Exec, or Exec_Search operation, this shall be
38 the value provided by the Arg_List parameter of that operation. If no arguments
39 were provided to the process at startup, the argument list shall contain at most
40 a single element. By convention, this element is the name of the program being
41 executed.

42 4.3.1.3 Error Handling

43 No exceptions shall be raised by Argument_List.

44 4.3.2 Environment Variables

45 4.3.2.1 Synopsis

```
46     type Environment is limited private;
47     procedure Copy_From_Current_Environment
48         (Env: in out Environment);
```



```

1  procedure Copy_To_Current_Environment
2  (Env: in Environment);
3  procedure Copy_Environment
4  (Source: in Environment;
5   Target: in out Environment);
6  function Environment_Value_Of
7  (Name:      POSIX.POSIX_String;
8   Env:       Environment;
9   Undefined: POSIX.POSIX_String:= "" )
10 return POSIX.POSIX_String;
11 function Environment_Value_Of
12 (Name:      POSIX.POSIX_String;
13  Undefined: POSIX.POSIX_String:= "" )
14 return POSIX.POSIX_String;
15 function Is_Environment_Variable
16 (Name: POSIX.POSIX_String;
17  Env:  Environment)
18 return Boolean;
19 function Is_Environment_Variable
20 (Name: POSIX.POSIX_String)
21 return Boolean;
22 procedure Clear_Environment
23 (Env: in out Environment);
24 procedure Clear_Environment;
25 procedure Set_Environment_Variable
26 (Name: in POSIX.POSIX_String;
27  Value: in POSIX.POSIX_String;
28  Env:   in out Environment);
29 procedure Set_Environment_Variable
30 (Name: in POSIX.POSIX_String;
31  Value: in POSIX.POSIX_String);
32 procedure Delete_Environment_Variable
33 (Name: in POSIX.POSIX_String;
34  Env:  in out Environment);
35 procedure Delete_Environment_Variable
36 (Name: in POSIX.POSIX_String);
37 function Length (Env: Environment) return Natural;
38 function Length return Natural;
39 generic
40   with procedure Action
41   (Name: in POSIX.POSIX_String;
42    Value: in POSIX.POSIX_String;
43    Quit: in out Boolean);
44 procedure For_Every_Environment_Variable
45 (Env: in Environment);
46 generic
47   with procedure Action
48   (Name: in POSIX.POSIX_String;
49    Value: in POSIX.POSIX_String;
50    Quit: in out Boolean);
51 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 are 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 whose value 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 generally 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.

- 1 In a multitasking program, the effect of one task calling an operation that modifies
2 an environment while another task is performing an operation on the same environ-
3 ment is undefined.
- 4 For iteration, distinct generic procedures are provided for the explicit and implicit
5 iterator forms, since Ada overloading rules require distinct names.
- 6 `Copy_From_Current_Environment` shall copy into `Env` a value of type `Environ-`
7 `ment` containing the same set of (name, value) pairs as are in the current environ-
8 ment of the calling process. The implementation of this procedure shall recover all
9 dynamically allocated storage that may have been allocated for the old value.
- 10 `Copy_To_Current_Environment` shall copy `Env` into the current environment of the
11 calling process, deleting all previous (name, value) pairs for the current environment.
12 The implementation of this procedure shall recover all dynamically allocated storage
13 that may have been allocated for the old value.
- 14 `Copy_Environment` shall copy the value of the parameter `Source` into the param-
15 eter `Target`. All previous values in `Target` are deleted. The implementation of
16 this procedure shall recover all dynamically allocated storage that may have been
17 allocated for the old value.
- 18 `Environment_Value_Of` shall search the specified environment for a variable whose
19 name matches the parameter `Name`. If such a variable is found, its value, possibly
20 null, shall be returned as a value of type `POSIX.POSIX_String`. If no matching
21 variable is found, the value provided by the parameter `Undefined` shall be returned.
- 22 `Is_Environment_Variable` shall return `True` if a variable with the name specified
23 by the parameter `Name` is in the specified environment. Otherwise, `False` shall be
24 returned.
- 25 `Clear_Environment` shall remove all variables from the specified environment. It
26 shall also recover all memory that may have been allocated dynamically by the PO-
27 SIX implementation for (name, value) pairs belonging to the specified environment.
- 28 `Set_Environment_Variable` shall add the (name, value) pair specified by `Name`
29 and `Value` to the specified environment. The value of the parameter `Value` may
30 be a null `POSIX.POSIX_String`. If the parameter `Name` is already the name of a
31 variable in the specified environment then the parameter `Value` shall become the
32 new value associated with the variable denoted by `Name`. The implementation of
33 this procedure shall recover all dynamically allocated storage that may have been
34 allocated for the old value.
- 35 `Delete_Environment_Variable` shall delete the variable named by the parameter
36 `Name` from the specified environment. If the name `Name` is not defined in the specified
37 environment, the procedure returns normally without any effect. The implementa-
38 tion of this procedure shall recover all dynamically allocated storage that may have
39 been allocated for the (name, value) pair.
- 40 `Length` shall return the number of (name, value) pairs defined in the specified envi-
41 ronment.

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 occur, 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.

1 `Get_Working_Directory` shall return the absolute pathname of the current work-
2 ing directory as a value of type `POSIX.Pathname`.

3 `Change_Working_Directory` shall change the process working directory to the
4 value of the parameter `Directory_Name`. If an exception is raised by `Change_`
5 `Working_Directory`, the current working directory shall be unchanged.

6 **4.3.3.3 Error Handling**

7 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
8 with the corresponding error code:

9 **Permission_Denied**

- 10 (1) The parameter `Directory_Name` to `Change_Working_Directory` con-
11 tains a component directory for which read or search permission is de-
12 nied.
- 13 (2) Read or search permission is denied for a component of the current pro-
14 cess working directory that would be returned by `Get_Working_Di-`
15 `rectory`.

16 **Filename_Too_Long**

17 The length of the parameter `Directory_Name` to `Change_Working_Di-`
18 `rectory` exceeds the system-imposed limit for that pathname. Information
19 on this limit is provided by `Pathname_Maximum` (see §5.4.1).

20 **Not_A_Directory**

21 The parameter `Directory_Name` to `Change_Working_Directory` contains
22 a component that is not a directory.

23 **No_Such_File_Or_Directory**

24 The parameter `Directory_Name` to `Change_Working_Directory` defines
25 a directory that does not exist or is null.

26 **4.4 Package POSIX_Calendar**

27 This package provides an analog to the facilities defined in the Ada standard package
28 `Calendar`. However, the time type defined in `POSIX_Calendar` includes sufficient
29 information to permit values of this type to be interpreted using the **TZ** environment
30 variable. (See §2.11.1.)

31 **with** `POSIX`,
32 `Calendar`;

33 **package** `POSIX_Calendar` **is**

34 *-- 4.4.1 Time Information*

35 **type** `POSIX_Time` **is private**;

36 **function** `Clock` **return** `POSIX_Time`;

37 **function** `To_Time` (`Date`: `POSIX_Time`) **return** `Calendar.Time`;

38 **function** `To_POSIX_Time` (`Date`: `Calendar.Time`) **return** `POSIX_Time`;

39 **function** `To_POSIX_Time` (`Date`: `POSIX.Timespec`) **return** `POSIX_Time`;

40 **function** `To_Timespec` (`Date`: `POSIX_Time`) **return** `POSIX.Timespec`;

41 **function** "+" (`L`: `POSIX_Time`; `R`: `Duration`) **return** `POSIX_Time`;

42 **function** "+" (`L`: `Duration`; `R`: `POSIX_Time`) **return** `POSIX_Time`;

43 **function** "-" (`L`: `POSIX_Time`; `R`: `Duration`) **return** `POSIX_Time`;

```

1  function "-" (L: POSIX_Time; R: POSIX_Time) return Duration;
2  function "<" (L, R: POSIX_Time) return Boolean;
3  function "<=" (L, R: POSIX_Time) return Boolean;
4  function ">" (L, R: POSIX_Time) return Boolean;
5  function ">=" (L, R: POSIX_Time) return Boolean;
6  -- 4.4.2 Operations on POSIX Times
7  subtype Year_Number is Calendar.Year_Number;
8  subtype Month_Number is Calendar.Month_Number;
9  subtype Day_Number is Calendar.Day_Number;
10 subtype Day_Duration is Calendar.Day_Duration;
11 function Year (Date: POSIX_Time) return Year_Number;
12 function Month (Date: POSIX_Time) return Month_Number;
13 function Day (Date: POSIX_Time) return Day_Number;
14 function Seconds (Date: POSIX_Time) return Day_Duration;
15 procedure Split
16   (Date: in POSIX_Time;
17    Year: out Year_Number;
18    Month: out Month_Number;
19    Day: out Day_Number;
20    Seconds: out Day_Duration);
21 function Time_Of
22   (Year: Year_Number;
23    Month: Month_Number;
24    Day: Day_Number;
25    Seconds: Day_Duration:= 0.0)
26   return POSIX_Time;
27 Time_Error: exception renames Calendar.Time_Error;
28 private
29   implementation-defined;
30 end POSIX_Calendar;

```

31 4.4.1 Time Information

32 4.4.1.1 Synopsis

```

33 type POSIX_Time is private;
34 function Clock return POSIX_Time;
35 function To_Time (Date: POSIX_Time) return Calendar.Time;
36 function To_POSIX_Time (Date: Calendar.Time) return POSIX_Time;
37 function To_POSIX_Time (Date: POSIX.Timespec) return POSIX_Time;
38 function To_Timespec (Date: POSIX_Time) return POSIX.Timespec;
39 function "+" (L: POSIX_Time; R: Duration) return POSIX_Time;
40 function "+" (L: Duration; R: POSIX_Time) return POSIX_Time;
41 function "-" (L: POSIX_Time; R: Duration) return POSIX_Time;
42 function "-" (L: POSIX_Time; R: POSIX_Time) return Duration;
43 function "<" (L, R: POSIX_Time) return Boolean;
44 function "<=" (L, R: POSIX_Time) return Boolean;
45 function ">" (L, R: POSIX_Time) return Boolean;
46 function ">=" (L, R: POSIX_Time) return Boolean;

```

47 4.4.1.2 Description

48 The type `POSIX_Time` is used to obtain times and dates from the system. Values of
49 this type shall be independent of the current time zone in effect for the system, so
50 that an arbitrary value of this type may be interpreted at a later date using informa-
51 tion in the **TZ** environment variable. (See §2.11.1.)

1 The function `Clock` shall return a value of type `POSIX_Time` reflecting the current
2 system time.

3 The functions named `To_POSIX_Time` shall convert a value of type `POSIX.Timespec`
4 or `Calendar.Time` to a value of type `POSIX_Time`.

5 The function `To_Timespec` shall convert a value of type `POSIX_Time` to a value of
6 type `POSIX.Timespec`.

7 The function `To_Time` shall convert a value of type `POSIX_Time` to a value of type
8 `Calendar.Time`.

9 When converting a value of the type `POSIX.Timespec`, the value shall be inter-
10 preted as an offset from the Epoch. (See §2.2.2.39.) If a value of `Calendar.Time` does
11 not contain sufficient information to interpret its value in a time-zone-independent
12 fashion, the value shall be interpreted using the current value of the `TZ` environment
13 variable.

14 The operators `"+"` and `"-"` provide for arithmetic on values of `POSIX_Time` and
15 the Ada standard type `Duration`. The operators `"<"`, `"<="`, `">"` and `">="` provide
16 relational operations on values of type `POSIX_Time`. The arithmetic and relational
17 operators have their conventional meanings.

18 **4.4.1.3 Error Handling**

19 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
20 with the corresponding error code:

21 **Invalid_Argument**

22 The value of the environment variable `TZ` is not recognized by this imple-
23 mentation as specifying valid time-zone information.

24 `Time_Error` may be raised by the operators `"+"` or `"-"` if the operations cannot
25 return a valid value of type `POSIX_Time` whose year number is in the range of the
26 subtype `Year_Number` or whose seconds value is not in the range of the type `Dura-`
27 `tion`.

28 `Constraint_Error` may be raised by `To_Time`, `To_POSIX_Time`, or `To_Timespec`
29 for implementation defined reasons, including the situation where a value of the
30 source type cannot be converted to a value of the target type.

31 **4.4.2 Operations on POSIX Times**

32 **4.4.2.1 Synopsis**

```
33     subtype Year_Number      is Calendar.Year_Number;
34     subtype Month_Number    is Calendar.Month_Number;
35     subtype Day_Number      is Calendar.Day_Number;
36     subtype Day_Duration    is Calendar.Day_Duration;
37     function Year (Date: POSIX_Time) return Year_Number;
38     function Month (Date: POSIX_Time) return Month_Number;
39     function Day (Date: POSIX_Time) return Day_Number;
40     function Seconds (Date: POSIX_Time) return Day_Duration;
```



```

1      procedure Split
2      (Date: in POSIX_Time;
3       Year: out Year_Number;
4       Month: out Month_Number;
5       Day: out Day_Number;
6       Seconds: out Day_Duration);
7      function Time_Of
8      (Year: Year_Number;
9       Month: Month_Number;
10      Day: Day_Number;
11      Seconds: Day_Duration:= 0.0)
12      return POSIX_Time;
13      Time_Error: exception renames Calendar.Time_Error;
```

14 4.4.2.2 Description

15 The procedure `Split` shall take a value of type `POSIX_Time` and shall interpret
16 it according to the current value of the **TZ** environment variable to provide time-
17 related information. (See §2.11.1.) The values returned by `Split` shall indicate
18 the Year, Month, Day, and Seconds represented by the `POSIX_Time` value. The
19 `Day_Duration` value in the parameter `Seconds` shall indicate the total number of
20 seconds in the indicated day. For instance, a value of `Seconds` of “3600.0” shall mean
21 1:00 AM on the given day.

22 The function `Year` shall return the year number corresponding to the date indicated
23 by the value of type `POSIX_Time`, interpreted using the current value of the **TZ**
24 environment variable. The function `Month` shall return the month number corre-
25 sponding to the date indicated by the value of type `POSIX_Time`, interpreted using
26 the current value of the **TZ** environment variable. The function `Day` shall return the
27 day number corresponding to the date indicated by the value of type `POSIX_Time`,
28 interpreted using the current value of the **TZ** environment variable. The function
29 `Seconds` shall return the number of seconds in the day corresponding to the date
30 indicated by the value of type `POSIX_Time`, interpreted using the current value of
31 the **TZ** environment variable.

32 The function `Time_Of` shall return a value of type `POSIX_Time`, interpreted in the
33 time zone indicated by the **TZ** environment variable.

34 If the value of **TZ** is null, then values of `POSIX_Time` shall be interpreted using
35 default system time-zone information.

36 4.4.2.3 Error Handling

37 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
38 with the corresponding error code:

39 Invalid_Argument

40 The value of the environment variable **TZ** is not recognized by this imple-
41 mentation as specifying valid time-zone information.

42 `Time_Error` shall be raised by `Time_Of` if the parameters cannot be interpreted to
43 form a valid date.

1 4.5 Package POSIX_Configurable_System_Limits

2 This package provides operations to retrieve the configurable system limits, options,
3 or versions at runtime.

```
4    with POSIX,  
5        POSIX_Limits,  
6        POSIX_Options;  
7    package POSIX_Configurable_System_Limits is  
8        -- 4.5.1 Get Configurable System Options  
9        function Asynchronous_IO_Is_Supported  
10        return POSIX_Options.Asynchronous_IO_Support;  
11        function File_Synchronization_Is_Supported  
12        return POSIX_Options.File_Synchronization_Support;  
13        function Job_Control_Supported                      -- obsolescent  
14        return POSIX.Job_Control_Support;                  -- obsolescent  
15        function Job_Control_Is_Supported  
16        return POSIX_Options.Job_Control_Support  
17        renames Job_Control_Supported;  
18        function Memory_Mapped_Files_Are_Supported  
19        return POSIX_Options.Memory_Mapped_Files_Support;  
20        function Memory_Locking_Is_Supported  
21        return POSIX_Options.Memory_Locking_Support;  
22        function Memory_Range_Locking_Is_Supported  
23        return POSIX_Options.Memory_Range_Locking_Support;  
24        function Memory_Protection_Is_Supported  
25        return POSIX_Options.Memory_Protection_Support;  
26        function Message_Queues_Are_Supported  
27        return POSIX_Options.Message_Queues_Support;  
28        function Mutex_Priority_Ceiling_Is_Supported  
29        return POSIX_Options.Mutex_Priority_Ceiling_Support;  
30        function Mutex_Priority_Inheritance_Is_Supported  
31        return POSIX_Options.Mutex_Priority_Inheritance_Support;  
32        function Mutexes_Are_Supported  
33        return POSIX_Options.Mutexes_Support;  
34        function Prioritized_IO_Is_Supported  
35        return POSIX_Options.Prioritized_IO_Support;  
36        function Process_Shared_Is_Supported  
37        return POSIX_Options.Process_Shared_Support;  
38        function Priority_Process_Scheduling_Is_Supported  
39        return POSIX_Options.Priority_Process_Scheduling_Support;  
40        function Priority_Task_Scheduling_Is_Supported  
41        return POSIX_Options.Priority_Task_Scheduling_Support;  
42        function Realtime_Signals_Are_Supported  
43        return POSIX_Options.Realtime_Signals_Support;  
44        function Saved_IDs_Supported                        -- obsolescent  
45        return POSIX.Saved_IDs_Support;                   -- obsolescent  
46        function Saved_IDs_Are_Supported  
47        return POSIX_Options.Saved_IDs_Support  
48        renames Saved_IDs_Supported;  
49        function Semaphores_Are_Supported  
50        return POSIX_Options.Semaphores_Support;  
51        function Shared_Memory_Objects_Are_Supported  
52        return POSIX_Options.Shared_Memory_Objects_Support;  
53        function Synchronized_IO_Is_Supported  
54        return POSIX_Options.Synchronized_IO_Support;  
55        function Timers_Are_Supported  
56        return POSIX_Options.Timers_Support;
```

```

1      -- 4.5.2 Get Configurable System Limits
2      type POSIX_Version is implementation-defined-integer;
3      function System_POSIX_Version return POSIX_Version;
4      function System_POSIX_Ada_Version return POSIX_Version;
5      function Argument_List_Maximum
6          return POSIX_Limits.Argument_List_Maxima;
7      function Asynchronous_IO_Maximum
8          return POSIX_Limits.Asynchronous_IO_Maxima;
9      function Asynchronous_IO_Priority_Delta_Maximum
10         return POSIX_Limits.Asynchronous_IO_Priority_Delta_Maxima;
11     function Child_Processes_Maximum
12         return POSIX_Limits.Child_Processes_Maxima;
13     function Groups_Maximum
14         return POSIX_Limits.Groups_Maxima;
15     function List_IO_Maximum
16         return POSIX_Limits.List_IO_Maxima;
17     function Open_Message_Queues_Maximum
18         return POSIX_Limits.Open_Message_Queues_Maxima;
19     function Message_Priority_Maximum
20         return POSIX_Limits.Message_Priority_Maxima;
21     function Open_Files_Maximum
22         return POSIX_Limits.Open_Files_Maxima;
23     function Page_Size
24         return POSIX_Limits.Page_Size_Range;
25     function Queued_Signals_Maximum
26         return POSIX_Limits.Queued_Signals_Maxima;
27     function Realtime_Signals_Maximum
28         return POSIX_Limits.Realtime_Signals_Maxima;
29     function Semaphores_Maximum
30         return POSIX_Limits.Semaphores_Maxima;
31     function Semaphores_Value_Maximum
32         return POSIX_Limits.Semaphores_Value_Maxima;
33     function Stream_Maximum -- obsolescent
34         return POSIX.Stream_Maxima; -- obsolescent
35     function Streams_Maximum
36         return POSIX_Limits.Streams_Maxima
37         renames Stream_Maximum;
38     function Timers_Maximum
39         return POSIX_Limits.Timers_Maxima;
40     function Timer_OVERRUNS_Maximum
41         return POSIX_Limits.Timer_OVERRUNS_Maxima;
42     function Time_Zone_String_Maximum
43         return POSIX_Limits.Time_Zone_String_Maxima;
44 end POSIX_Configurable_System_Limits;

```

45 4.5.1 Get Configurable System Options

46 4.5.1.1 Synopsis

```

47     function Asynchronous_IO_Is_Supported
48         return POSIX_Options.Asynchronous_IO_Support;
49     function File_Synchronization_Is_Supported
50         return POSIX_Options.File_Synchronization_Support;
51     function Job_Control_Supported -- obsolescent
52         return POSIX.Job_Control_Support; -- obsolescent

```

```

1  function Job_Control_Is_Supported
2      return POSIX_Options.Job_Control_Support
3      renames Job_Control_Supported;
4  function Memory_Mapped_Files_Are_Supported
5      return POSIX_Options.Memory_Mapped_Files_Support;
6  function Memory_Locking_Is_Supported
7      return POSIX_Options.Memory_Locking_Support;
8  function Memory_Range_Locking_Is_Supported
9      return POSIX_Options.Memory_Range_Locking_Support;
10 function Memory_Protection_Is_Supported
11     return POSIX_Options.Memory_Protection_Support;
12 function Message_Queues_Are_Supported
13     return POSIX_Options.Message_Queues_Support;
14 function Mutex_Priority_Ceiling_Is_Supported
15     return POSIX_Options.Mutex_Priority_Ceiling_Support;
16 function Mutex_Priority_Inheritance_Is_Supported
17     return POSIX_Options.Mutex_Priority_Inheritance_Support;
18 function Mutexes_Are_Supported
19     return POSIX_Options.Mutexes_Support;
20 function Prioritized_IO_Is_Supported
21     return POSIX_Options.Prioritized_IO_Support;
22 function Process_Shared_Is_Supported
23     return POSIX_Options.Process_Shared_Support;
24 function Priority_Process_Scheduling_Is_Supported
25     return POSIX_Options.Priority_Process_Scheduling_Support;
26 function Priority_Task_Scheduling_Is_Supported
27     return POSIX_Options.Priority_Task_Scheduling_Support;
28 function Realtime_Signals_Are_Supported
29     return POSIX_Options.Realtime_Signals_Support;
30 function Saved_IDS_Supported -- obsolescent
31     return POSIX.Saved_IDS_Support; -- obsolescent
32 function Saved_IDS_Are_Supported
33     return POSIX_Options.Saved_IDS_Support
34     renames Saved_IDS_Supported;
35 function Semaphores_Are_Supported
36     return POSIX_Options.Semaphores_Support;
37 function Shared_Memory_Objects_Are_Supported
38     return POSIX_Options.Shared_Memory_Objects_Support;
39 function Synchronized_IO_Is_Supported
40     return POSIX_Options.Synchronized_IO_Support;
41 function Timers_Are_Supported
42     return POSIX_Options.Timers_Support;

```

43 4.5.1.2 Description

44 These functions can be used to discover the support provided for certain options (see
45 §2.5) by a particular implementation. For system-wide Boolean valued options, the
46 function shall return a True if the corresponding option is supported, and False if
47 the option is not supported. For pathname-specific Boolean valued options, the func-
48 tion shall return True if the corresponding option is supported on all or some files,
49 and False if it is not supported on any files. For multi-valued options, the function
50 shall return the value that indicates the support for the option. The correspondence
51 of functions to options (see §2.5) shall be as shown in Table 4.1.

52 NOTE: For pathname-specific options, the application can only determine reliably that
53 an option is supported on a specific file by calling the corresponding pathname-specific
54 function, declared in the package `POSIX_Configurable_File_Limits` and described
55 in §5.4.

56 NOTE: Alternate names are provided for `Job_Control_Is_Supported` and `Saved_`
57 `IDS_Are_Supported`, for compatibility with the names of these operations in POSIX.5
58 {B3}.

Table 4.1 – Functions for System-Wide Options

Function	Option
Asynchronous_IO_Is_Supported	Asynchronous I/O
File_Synchronization_Is_Supported	File Synchronization
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
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
Semaphores_Are_Supported	Semaphores
Shared_Memory_Objects_Are_Supported	Shared Memory Objects
Synchronized_IO_Is_Supported	Synchronized I/O
Timers_Are_Supported	Timers

4.5.1.3 Error Handling

None of these functions shall raise any exception.

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;

```

```

1  function Open_Files_Maximum
2      return POSIX_Limits.Open_Files_Maxima;
3  function Page_Size
4      return POSIX_Limits.Page_Size_Range;
5  function Queued_Signals_Maximum
6      return POSIX_Limits.Queued_Signals_Maxima;
7  function Realtime_Signals_Maximum
8      return POSIX_Limits.Realtime_Signals_Maxima;
9  function Semaphores_Maximum
10     return POSIX_Limits.Semaphores_Maxima;
11 function Semaphores_Value_Maximum
12     return POSIX_Limits.Semaphores_Value_Maxima;
13 function Stream_Maximum          -- obsolescent
14     return POSIX.Stream_Maxima;          -- obsolescent
15 function Streams_Maximum
16     return POSIX_Limits.Streams_Maxima
17     renames Stream_Maximum;
18 function Timers_Maximum
19     return POSIX_Limits.Timers_Maxima;
20 function Timer_OVERRUNS_Maximum
21     return POSIX_Limits.Timer_OVERRUNS_Maxima;
22 function Time_Zone_String_Maximum
23     return POSIX_Limits.Time_Zone_String_Maxima;

```

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.

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
Streams_Maximum	Streams Maximum
Timers_Maximum	Timers Maximum
Timer_OVERRUNS_Maximum	Timer Overruns Maximum
Time_Zone_String_Maximum	Time Zone String Maximum

1 `System_POSIX_Version` shall return a value describing the implemented version
2 or revision of POSIX.1 {2}. The preferred interpretation of this integer is six decimal
3 digits representing when the version of the standard was approved by the IEEE
4 Standards Board; the year is the first four digits, and the month is the last two
5 digits. This interpretation is not required by this standard.

6 `System_POSIX_Ada_Version` shall return a value describing the implemented ver-
7 sion or revision of this standard. The preferred interpretation of this integer is six
8 decimal digits representing when the version of the standard was approved by the
9 IEEE Standards Board; the year is the first four digits, and the month is the last two
10 digits. This interpretation is not required by this standard.

11 **4.5.2.3 Error Handling**

12 None of these functions shall raise any exception.

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

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 input/output operations, which are defined in the packages `POSIX_IO` and `POSIX_File_Locking` in §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 if the desired access can be allowed.

```
12  package POSIX_Permissions is
13      -- 5.1.1 File Permissions
14      type Permission is
15          (Others_Execute, Others_Write, Others_Read,
16           Group_Execute, Group_Write, Group_Read,
17           Owner_Execute, Owner_Write, Owner_Read,
18           Set_Group_ID, Set_User_ID);
19      type Permission_Set is array (Permission) of Boolean;
20      Owner_Permission_Set: constant Permission_Set := Permission_Set'
21          (Owner_Read | Owner_Write | Owner_Execute => true,
22           others => false);
23      Group_Permission_Set: constant Permission_Set := Permission_Set'
24          (Group_Read | Group_Write | Group_Execute => true,
25           others => false);
26      Others_Permission_Set: constant Permission_Set := Permission_Set'
27          (Others_Read | Others_Write | Others_Execute => true,
28           Others => false);
29      Access_Permission_Set: constant Permission_Set := Permission_Set'
30          (Owner_Read | Owner_Write | Owner_Execute => true,
31           Group_Read | Group_Write | Group_Execute => true,
32           Others_Read | Others_Write | Others_Execute => true,
33           Others => false);
34      Set_Group_ID_Set: constant Permission_Set := Permission_Set'
35          (Set_Group_ID => true,
36           others => false);
37      Set_User_ID_Set: constant Permission_Set := Permission_Set'
38          (Set_User_ID => true,
39           others => false);
40      -- 5.1.2 Process Permission Set
41      function Get_Allowed_Process_Permissions return Permission_Set;
42      procedure Set_Allowed_Process_Permissions
43          (Permissions: in Permission_Set);
44      procedure Set_Allowed_Process_Permissions
45          (Permissions: in Permission_Set;
46           Old_Permis: out Permission_Set);
47  end POSIX_Permissions;
```

5.1.1 File Permissions

5.1.1.1 Synopsis

```

3  type Permission is
4      (Others_Execute, Others_Write, Others_Read,
5       Group_Execute, Group_Write, Group_Read,
6       Owner_Execute, Owner_Write, Owner_Read,
7       Set_Group_ID, Set_User_ID);
8  type Permission_Set is array (Permission) of Boolean;
9  Owner_Permission_Set: constant Permission_Set := Permission_Set'
10     (Owner_Read | Owner_Write | Owner_Execute => true,
11      others => false);
12  Group_Permission_Set: constant Permission_Set := Permission_Set'
13     (Group_Read | Group_Write | Group_Execute => true,
14      others => false);
15  Others_Permission_Set: constant Permission_Set := Permission_Set'
16     (Others_Read | Others_Write | Others_Execute => true,
17      Others => false);
18  Access_Permission_Set: constant Permission_Set := Permission_Set'
19     (Owner_Read | Owner_Write | Owner_Execute => true,
20      Group_Read | Group_Write | Group_Execute => true,
21      Others_Read | Others_Write | Others_Execute => true,
22      Others => false);
23  Set_Group_ID_Set: constant Permission_Set := Permission_Set'
24     (Set_Group_ID => true,
25      others => false);
26  Set_User_ID_Set: constant Permission_Set := Permission_Set'
27     (Set_User_ID => true,
28      others => false);

```

5.1.1.2 Description

The enumeration type `Permission` shall list the standard POSIX file modes. Sets of Permissions 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 Permissions 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.

1 Group_Read
2 Read permission for the file group class.
3 Group_Write
4 Write permission for the file group class.
5 Group_Execute
6 Search (if a directory) or execute (otherwise) permission for the file
7 group class.
8 Others_Permission_Set
9 Read, write, search (if a directory), or execute (otherwise) permission for the
10 file other class.
11 Others_Read
12 Read permission for the file other class.
13 Others_Write
14 Write permission for the file other class.
15 Others_Execute
16 Search (if a directory) or execute (otherwise) permission for the file
17 other class.
18 Set_Group_ID_Set
19 Contains only the single element Set_Group_ID. The effective group ID of
20 the process shall be set to that of the group of the file when the file is run as
21 a program. On a regular file, Set_Group_ID should be cleared on any write.
22 Set_User_ID_Set
23 Contains only the single element Set_User_ID. The effective user ID of the
24 process shall be set to that of the owner of the file when the file is run as a
25 program. On a regular file, Set_User_ID should be cleared on any write.

26 5.1.2 Process Permission Set

27 5.1.2.1 Synopsis

28 **function** Get_Allowed_Process_Permissions **return** Permission_Set;
29 **procedure** Set_Allowed_Process_Permissions
30 (Permissions: **in** Permission_Set);
31 **procedure** Set_Allowed_Process_Permissions
32 (Permissions: **in** Permission_Set;
33 Old_Perm: **out** Permission_Set);

34 5.1.2.2 Description

35 Each process shall have a set of allowed process permissions associated with it. The
36 process permission set shall contain only Permissions that represent file access
37 permissions, that is, those Permissions contained in the Access_Permission_
38 Set. It does not contain either Set_Group_ID or Set_User_ID.

39 The set is implicitly used whenever the process creates a new file (operations Cre-
40 ate_Directory, Create_FIFO, and POSIX_IO. Open_Or_Create; see §5.2.1 and
41 §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_Permis. 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;
with 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;
    -- 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;
```

```

1  generic
2      with procedure Action
3          (D_Entry: in Directory_Entry;
4           Quit:    in out Boolean);
5  procedure For_Every_Directory_Entry
6      (Pathname: in POSIX.Pathname);
7  -- 5.2.5 Update File Status Information
8  procedure Change_Owner_And_Group
9      (Pathname: in POSIX.Pathname;
10       Owner:   in POSIX_Process_Identification.User_ID;
11       Group:   in POSIX_Process_Identification.Group_ID);
12 procedure Change_Permissions
13     (Pathname: in POSIX.Pathname;
14      Permission: in POSIX_Permissions.Permission_Set);
15 procedure Set_File_Times
16     (Pathname:      in POSIX.Pathname;
17      Access_Time:   in POSIX_Calendar.POSIX_Time;
18      Modification_Time: in POSIX_Calendar.POSIX_Time);
19 procedure Set_File_Times (Pathname: in POSIX.Pathname);
20 -- 5.2.6 Check File Accessibility
21 type Access_Mode is (Read_Ok, Write_Ok, Execute_Ok);
22 type Access_Mode_Set is array (Access_Mode) of Boolean;
23 function Is_Accessible
24     (Pathname: POSIX.Pathname;
25      Access_Modes: Access_Mode_Set)
26     return Boolean;
27 function Accessibility
28     (Pathname: POSIX.Pathname;
29      Access_Modes: Access_Mode_Set)
30     return POSIX.Error_Code;
31 function Is_File_Present (Pathname: POSIX.Pathname) return Boolean;
32 function Existence (Pathname: POSIX.Pathname) return POSIX.Error_Code;
33 private
34     implementation-defined
35 end POSIX_Files;

```

36 5.2.1 Create and Remove Files

37 5.2.1.1 Synopsis

```

38 procedure Create_Directory
39     (Pathname: in POSIX.Pathname;
40      Permission: in POSIX_Permissions.Permission_Set);
41 procedure Create_FIFO
42     (Pathname: in POSIX.Pathname;
43      Permission: in POSIX_Permissions.Permission_Set);
44 procedure Unlink (Pathname: in POSIX.Pathname);
45 procedure Remove_Directory (Pathname: in POSIX.Pathname);

```

46 5.2.1.2 Description

47 A directory is a file that contains directory entries. Each directory entry shall have a
48 unique name within that directory and associate a filename with a file. A directory
49 entry is also referred to as a link, in that it “links” the filename to its file. A file may
50 be referenced by than one directory entry. Implementations may support linking of

1 files across file systems, and they may also support links to directories. A file exists
2 as long as there is at least one directory entry (link) that refers to it or there is at
3 least one process with the file open.

4 One of the attributes maintained for a file shall be its *link count*, the number of
5 directory entries (links) that refer to it.

6 `Create_Directory` shall create a directory. The name of the directory shall be
7 specified by `Pathname`. `Create_FIFO` shall create a FIFO. The name of the FIFO
8 shall be specified by `Pathname`. For each of the file creation operations, the file
9 permission given to the new file shall be obtained from `Permission` as modified by
10 the process permission set. (See §5.1.2.) The owner of the file shall be set to the
11 effective user ID of the creating process, and the group of the file shall be set to
12 the effective group ID of the process. Upon successful completion of a file creation
13 operation, the Last Access Time, Last Modification Time, and Last Status Change Time of
14 the new file shall be marked for update. (See §2.3.10.) Also, the Last Modification Time
15 and Last Status Change Time values for the parent directory of the new file shall be
16 marked for update.

17 File removal is accomplished through `Unlink`. `Unlink` shall remove the directory
18 entry (link) named by the `Pathname` and decrement the link count of the file ref-
19 erenced by the entry. If the decremented link count is zero and no process has the
20 file open, the space occupied by the file shall be freed, and the file shall no longer be
21 accessible. If one or more processes have the file open when the last link is removed,
22 removal of the file contents shall be postponed until all references to the file have
23 been closed.

24 The parameter `Pathname` shall not name a directory unless the process has appro-
25 priate privileges and the implementation supports using `Unlink` on directories. An
26 application should use `Remove_Directory` to remove a directory.

27 Upon successful completion, the Last Modification Time and Last Status Change Time of
28 the parent directory shall be marked for update. (See §2.3.10.) Additionally, if the
29 link count of the file is not zero, the Last Status Change Time of the file shall be marked
30 for update.

31 `Remove_Directory` shall remove a directory named by the `Pathname`. The directory
32 shall only be removed if it is an empty directory. (See §2.2.2.36.) If the directory is
33 the root directory or the current working directory of any process, the effect of this
34 procedure is implementation defined.

35 Removal of a directory shall cause the link named by the `Pathname` to be removed
36 and the link count of the directory to be decremented. When the link count becomes
37 zero and no process has the directory open, the space occupied by the directory shall
38 be freed, and the directory shall no longer be accessible. If one or more processes
39 have the directory open when the last link is removed, the dot and dot-dot entries,
40 if present, shall be removed, and no new entries may be created in the directory.
41 Removal of the directory is postponed until all references to the directory have been
42 closed. Upon successful completion of the `Remove_Directory` operation, the Last
43 Modification Time and Last Status Change Time of the parent directory shall be marked
44 for update. (See §2.3.10.)

5.2.1.3 Error Handling

If any of the following conditions occur, 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. 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` when using `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 when using `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 when using `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 when using `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 when using `Create_Directory`, `Create_FIFO`, or `Unlink`.
- (2) A component of the path named by `Pathname` is not a directory when using `Remove_Directory`.

`Resource_Busy`

The directory named by `Pathname` cannot be removed because it is being used by another process, and the implementation considers this to be an error when using `Unlink` or `Remove_Directory`.

`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 those 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;
```

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.

5.2.2.3 Error Handling

No exceptions shall be returned by `Is_File`, `Is_Directory`, `Is_FIFO`, `Is_Character_Special_File`, or `Is_Block_Special_File`.

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.

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.)

5.2.3.3 Error Handling

If any of the following conditions occur, 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` when using `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. 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 when using `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 when using `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 when using `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, when using `Link`.

`Too_Many_Links`

When using `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 when using `Link`.

1 **Operation_Not_Permitted**
2 The file referenced by `Old_Pathname` is a directory, and the process does
3 not have privilege to link directories or the implementation does not permit
4 linking directories when using `Link`.

5 **Improper_Link**
6 The file named by `Old_Pathname` and the file named by `New_Pathname`
7 reside on different file systems, and the implementation does not support
8 links across file systems.

9 **Resource_Busy**
10 The parent directory of either `Old_Pathname` or `New_Pathname` cannot be
11 modified because it is being used by another process, and the implementa-
12 tion considers this an error when using `Rename`.

13 **File_Exists**
14 or **Directory_Not_Empty**
15 The file named by `New_Pathname` is an existing directory that contains en-
16 tries other than dot and dot-dot when using `Rename`. (`Directory_Not_-`
17 **Empty** is the preferred error code.)

18 **Invalid_Argument**
19 `New_Pathname` contains a path prefix that names the same directory as a
20 component of `Old_Pathname`, when using `Rename`.

21 **Is_A_Directory**
22 `New_Pathname` names a directory and `Old_Pathname` does not name a di-
23 rectory when using `Rename`.

24 In those situations where multiple error codes may be returned, which possible error
25 code returned is unspecified.

26 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
27 with the corresponding error code:

28 **Permission_Denied**
29 A component of either path prefix denies search permission; or the requested
30 link requires writing in a directory with permissions that denies write per-
31 missions; or the calling process does not have permission to access the file
32 named by `Old_Pathname`, and this is required by the implementation of
33 **Link**.

34 **5.2.4 Directory Iteration**

35 **5.2.4.1 Synopsis**

```
36     type Directory_Entry is limited private;  
37     function Filename_Of (D_Entry: Directory_Entry) return POSIX.Filename;  
38     generic  
39         with procedure Action  
40             (D_Entry: in Directory_Entry;  
41             Quit: in out Boolean);  
42     procedure For_Every_Directory_Entry  
43         (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 `Action` 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 or not is unspecified.

5.2.4.3 Error Handling

`Filename_Of` shall not raise any exceptions.

If any of the following conditions occur, 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.

1 **Filename_Too_Long**
2 The length in POSIX characters of the specified pathname exceeds Pathname
3 Limit or the length in POSIX characters of a component of the specified path-
4 name is greater than Filename Maximum and the Filename Truncation option
5 is not supported for the pathname prefix of that component. To determine
6 these limits, the limit functions are applied to Pathname.

7 **5.2.5 Update File Status Information**

8 **5.2.5.1 Synopsis**

```
9     procedure Change_Owner_And_Group  
10         (Pathname: in POSIX.Pathname;  
11         Owner:     in POSIX_Process_Identification.User_ID;  
12         Group:     in POSIX_Process_Identification.Group_ID);  
13     procedure Change_Permissions  
14         (Pathname: in POSIX.Pathname;  
15         Permission: in POSIX_Permissions.Permission_Set);  
16     procedure Set_File_Times  
17         (Pathname:         in POSIX.Pathname;  
18         Access_Time:       in POSIX_Calendar.POSIX_Time;  
19         Modification_Time: in POSIX_Calendar.POSIX_Time);  
20     procedure Set_File_Times (Pathname: in POSIX.Pathname);
```

21 **5.2.5.2 Description**

22 Change_Owner_And_Group shall change the user ID and the group ID of the file
23 named by Pathname to the Owner_ID and Group_ID, respectively. Only processes
24 with an effective user ID equal to the user ID of the file or with appropriate privileges
25 may change the ownership of a file.

26 If the Change Owner Restriction option is imposed on Pathname,

- 27 — Changing the owner is restricted to processes with appropriate privileges.
- 28 — Changing the group is permitted by a process with an effective user ID equal to
29 the user ID of the file, but without appropriate privileges, if and only if Owner is
30 equal to the user ID of the file and Group is equal either to the effective group
31 ID of the calling process or to one of its supplementary group IDs.

32 If the Pathname refers to a regular file, Set_User_ID and Set_Group_ID of the
33 file mode shall be cleared upon successful return from Change_Owner_And_Group
34 unless the call is made from a process with appropriate privileges, in which case it is
35 implementation defined whether those modes are altered. The Last Status Change Time
36 of the file shall be marked for update upon successful completion of this operation.
37 (See §2.3.10.)

38 Change_Permissions shall change the permission set associated with the file nam-
39 ed by Pathname. If the effective user ID of the calling process matches the file owner
40 or the calling process has appropriate privileges, Change_Permissions shall change
41 the permissions of the file to those specified in Permission. The Last Status Change
42 Time of the file shall be marked for update upon completion of this operation. (See
43 §2.3.10.)

1 Additional implementation defined restrictions may cause `Set_Group_ID` and `Set_`
2 `User_ID` in `Permission` to be ignored.

3 If the calling process does not have appropriate privileges, and if the group ID of
4 the file does not match the effective group ID or one of the supplementary group
5 IDs, and one or more of `Owner_Execute`, `Group_Execute`, and `Others_Execute`
6 are specified in `Permission`, and the file is a regular file, `Set_Group_ID` in the
7 permission set of the file shall be cleared upon successful return from `Change_`
8 `Permissions`.

9 The effect on open file descriptions for files open at the time of the call to `Change_`
10 `Permissions` is implementation defined.

11 The Last Status Change Time of the file shall be marked for update upon completion of
12 this operation. (See §2.3.10.)

13 If the effective user ID of the process matches the owner of the file or if the process
14 has appropriate privileges, `Set_File_Times` shall change the Last Access Time and
15 Last Modification Time associated with the file named by `Pathname`. If both `Access_`
16 `Time` and `Modification_Time` are specified, then the access time of the file shall
17 be changed to `Access_Time`, and the Last Modification Time of the file shall be changed
18 to `Modification_Time`.

19 If neither `Access_Time` nor `Modification_Time` are specified, then the Last Access
20 Time and Last Modification Time of the file are set to the same value, the current value
21 of `POSIX_Calendar.Clock`.

22 The Last Status Change Time of the file shall be marked for update upon successful
23 completion of these operations. (See §2.3.10.)

24 **5.2.5.3 Error Handling**

25 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
26 with the corresponding error code:

27 **`Permission_Denied`**

28 Search permission is denied for a component of `Pathname`, or the process
29 does not have sufficient privilege to perform the operation.

30 **`Filename_Too_Long`**

31 The length in POSIX characters of the specified pathname exceeds `Pathname`
32 `Limit` or the length in POSIX characters of a component of the specified path-
33 name is greater than `Filename Maximum` and the `Filename Truncation` option
34 is not supported for the pathname prefix of that component. To determine
35 these limits, the limit functions are applied to `Pathname`.

36 **`No_Such_File_Or_Directory`**

37 The file named by `Pathname` does not exist.

38 **`Not_A_Directory`**

39 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

```
1  type Access_Mode is (Read_Ok, Write_Ok, Execute_Ok);
2  type Access_Mode_Set is array (Access_Mode) of Boolean;
3  function Is_Accessible
4      (Pathname: POSIX.Pathname;
5       Access_Modes: Access_Mode_Set)
6      return Boolean;
7  function Accessibility
8      (Pathname: POSIX.Pathname;
9       Access_Modes: Access_Mode_Set)
10     return POSIX.Error_Code;
11  function Is_File_Present (Pathname: POSIX.Pathname) return Boolean;
12  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 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. 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
    type Status is private;
    -- 5.3.1 Access File Status
    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;
```

```

1      function Last_Access_Time_Of
2          (File_Status: Status) return POSIX_Calendar.POSIX_Time;
3      function Last_Modification_Time_Of
4          (File_Status: Status) return POSIX_Calendar.POSIX_Time;
5      function Last_Status_Change_Time_Of
6          (File_Status: Status) return POSIX_Calendar.POSIX_Time;
7      function Is_Directory (File_Status: Status) return Boolean;
8      function Is_Character_Special_File
9          (File_Status: Status) return Boolean;
10     function Is_Block_Special_File
11         (File_Status: Status) return Boolean;
12     function Is_Regular_File (File_Status: Status) return Boolean;
13     function Is_FIFO (File_Status: Status) return Boolean;
14     function Is_Shared_Memory
15         (File_Status: in Status) return Boolean;
16     function Is_Message_Queue
17         (File_Status: in Status) return Boolean;
18     function Is_Semaphore
19         (File_Status: in Status) return Boolean;
20 private
21     implementation-defined
22 end POSIX_File_Status;

```

23 5.3.1 Access File Status

24 5.3.1.1 Synopsis

```

25     function Get_File_Status (Pathname: POSIX_Pathname) return Status;
26     function Get_File_Status
27         (File: POSIX_IO.File_Descriptor) return Status;

```

28 5.3.1.2 Description

29 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.

34 5.3.1.3 Error Handling

35 If any of the following conditions occur, the exception POSIX_Error shall be raised, with the corresponding error code:

37 Permission_Denied

38 Search permission is denied for a component of Pathname.

39 Filename_Too_Long

40 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. To determine these limits, the limit functions are applied to Pathname.

- 1 **No_Such_File_Or_Directory**
- 2 The file named by Pathname does not exist.
- 3 **Not_A_Directory**
- 4 A component of the path prefix of Pathname does not refer to a directory.
- 5 **Bad_File_Descriptor**
- 6 File is not a valid File_Descriptor.

7 **5.3.2 Access Status Information**

8 **5.3.2.1 Synopsis**

```

9     type File_ID is private;
10    type Device_ID is private;
11    subtype Links is natural range 0 .. POSIX.Link_Limit_Maxima'Last;
12    function Permission_Set_Of
13      (File_Status: Status) return POSIX.Permissions.Permission_Set;
14    function File_ID_Of (File_Status: Status) return File_ID;
15    function Device_ID_Of (File_Status: Status) return Device_ID;
16    function Link_Count_Of (File_Status: Status) return Links;
17    function Owner_Of
18      (File_Status: Status) return POSIX.Process_Identification.User_ID;
19    function Group_Of
20      (File_Status: Status) return POSIX.Process_Identification.Group_ID;
21    function Size_Of (File_Status: Status) return POSIX.IO_Count;
22    function Last_Access_Time_Of
23      (File_Status: Status) return POSIX_Calendar.POSIX_Time;
24    function Last_Modification_Time_Of
25      (File_Status: Status) return POSIX_Calendar.POSIX_Time;
26    function Last_Status_Change_Time_Of
27      (File_Status: Status) return POSIX_Calendar.POSIX_Time;
28    function Is_Directory (File_Status: Status) return Boolean;
29    function Is_Character_Special_File
30      (File_Status: Status) return Boolean;
31    function Is_Block_Special_File
32      (File_Status: Status) return Boolean;
33    function Is_Regular_File (File_Status: Status) return Boolean;
34    function Is_FIFO (File_Status: Status) return Boolean;
35    function Is_Shared_Memory
36      (File_Status: in Status) return Boolean;
37    function Is_Message_Queue
38      (File_Status: in Status) return Boolean;
39    function Is_Semaphore
40      (File_Status: in Status) return Boolean;

```

41 **5.3.2.2 Description**

42 Values of the type File_ID shall uniquely identify each file on a specific device.
 43 Values of the type Device_ID shall uniquely identify each device in the system.
 44 Together, they shall uniquely identify a file in the system.

45 Values of the subtype Links shall contain the number of links to a specific file (iden-
 46 tified by a File_ID and/or Device_ID).

47 Permission_Set_Of shall return the file permissions for the specified file.

- 1 `File_ID_Of` shall return the file serial number associated with the file.
- 2 `Device_ID_Of` shall return the device serial number for the device that contains the
3 file.
- 4 `Link_Count_Of` shall return the number of links to the specified file.
- 5 `Owner_Of` shall return the owner associated with the file.
- 6 `Group_Of` shall return the group associated with the file.
- 7 For regular files, `Size_Of` shall return the number of bytes stored in the file. For
8 other file types, the use of this function is unspecified.
- 9 `Last_Access_Time_Of` shall return the last time the file was accessed by the system.
- 10 `Last_Modification_Time_Of` shall return the last time the file was modified by
11 the system.
- 12 `Last_Status_Change_Time_Of` shall return the last time status information asso-
13 ciated with the file was changed.
- 14 Values of type `Status` can contain status information for one of several types of
15 objects. The functions described in this section are provided to determine the type of
16 object. Each of the functions takes a single input parameter of type `Status`.
- 17 `Is_Directory` shall return `True` if the file is a directory, and `False` otherwise.
- 18 `Is_Character_Special_File` shall return `True` if the file is a character special
19 file, and `False` otherwise.
- 20 `Is_Block_Special_File` shall return `True` if the file is a block special file, and
21 `False` otherwise.
- 22 `Is_Regular_File` shall return `True` if the file is a regular file, and `False` otherwise.
- 23 `Is_FIFO` shall return `True` if the file is a FIFO special file, and `False` otherwise.
- 24 `Is_Shared_Memory` shall return `True` if the file is that of a POSIX shared memory
25 object and it is implemented as a distinct file type; otherwise it shall return `False`.
- 26 `Is_Message_Queue` shall return `True` if the file is that of a POSIX message queue
27 object and it is implemented as a distinct file type; otherwise it shall return `False`.
- 28 `Is_Semaphore` shall return `True` if the file is that of a POSIX semaphore object and
29 it is implemented as a distinct file type; otherwise it shall return `False`.

30 NOTE: Message queues and semaphores may, but need not be, associated with file de-
31 scriptors. If they are associated with file descriptors, they may, but need not be, im-
32 plemented as distinct file types. Shared memory objects are always associated with
33 file descriptors, and they may, but need not be, implemented as a distinct file type.
34 Therefore, it is implementation defined whether `Is_Message_Queue`, `Is_Semaphore`,
35 or `Is_Shared_Memory` return `True`.

5.3.2.3 Error Handling

If any of the following conditions occur, the exception `POSIX_Error` shall be raised, with the corresponding error code:

`Invalid_Argument`

`File_Status` is invalid.

If any of the following conditions occur 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;
```



```

1      function Input_Line_Limit (Pathname: POSIX.Pathname)           -- obsolescent
2      return POSIX.Input_Line_Limit_Maxima;                         -- obsolescent
3      function Input_Line_Limit (File: POSIX_IO.File_Descriptor) -- obsolescent
4      return POSIX.Input_Line_Limit_Maxima;                         -- obsolescent
5      function Input_Line_Maximum (Pathname: POSIX.Pathname)
6      return POSIX_Limits.Input_Line_Maxima
7      renames Input_Line_Limit;
8      function Input_Line_Maximum (File: POSIX_IO.File_Descriptor)
9      return POSIX_Limits.Input_Line_Maxima
10     renames Input_Line_Limit;
11     -- Input queue limits
12     function Input_Queue_Is_Limited (Pathname: POSIX.Pathname)
13     return Boolean;
14     function Input_Queue_Is_Limited (File: POSIX_IO.File_Descriptor)
15     return Boolean;
16     function Input_Queue_Limit (Pathname: POSIX.Pathname)         -- obsolescent
17     return POSIX.Input_Queue_Limit_Maxima;                       -- obsolescent
18     function Input_Queue_Limit (File: POSIX_IO.File_Descriptor) -- obsolescent
19     return POSIX.Input_Queue_Limit_Maxima;                       -- obsolescent
20     function Input_Queue_Maximum (Pathname: POSIX.Pathname)
21     return POSIX_Limits.Input_Queue_Maxima
22     renames Input_Queue_Limit;
23     function Input_Queue_Maximum (File: POSIX_IO.File_Descriptor)
24     return POSIX_Limits.Input_Queue_Maxima
25     renames Input_Queue_Limit;
26     -- Filename and pathname limits
27     function Filename_Is_Limited (Pathname: POSIX.Pathname)
28     return Boolean;
29     function Filename_Is_Limited (File: POSIX_IO.File_Descriptor)
30     return Boolean;
31     function Filename_Limit (Pathname: POSIX.Pathname)           -- obsolescent
32     return POSIX.Filename_Limit_Maxima;                         -- obsolescent
33     function Filename_Limit (File: POSIX_IO.File_Descriptor)     -- obsolescent
34     return POSIX.Filename_Limit_Maxima;                         -- obsolescent
35     function Filename_Maximum (Pathname: POSIX.Pathname)
36     return POSIX_Limits.Filename_Maxima
37     renames Filename_Limit;
38     function Filename_Maximum (File: POSIX_IO.File_Descriptor)
39     return POSIX_Limits.Filename_Maxima
40     renames Filename_Limit;
41     function Pathname_Is_Limited (Pathname: POSIX.Pathname)
42     return Boolean;
43     function Pathname_Is_Limited (File: POSIX_IO.File_Descriptor)
44     return Boolean;
45     function Pathname_Limit (Pathname: POSIX.Pathname)           -- obsolescent
46     return POSIX.Pathname_Limit_Maxima;                         -- obsolescent
47     function Pathname_Limit (File: POSIX_IO.File_Descriptor)    -- obsolescent
48     return POSIX.Pathname_Limit_Maxima;                         -- obsolescent
49     function Pathname_Maximum (Pathname: POSIX.Pathname)
50     return POSIX_Limits.Pathname_Maxima
51     renames Pathname_Limit;
52     function Pathname_Maximum (File: POSIX_IO.File_Descriptor)
53     return POSIX_Limits.Pathname_Maxima
54     renames Pathname_Limit;
55     -- Pipe length limits
56     function Pipe_Length_Is_Limited (Pathname: POSIX.Pathname)
57     return Boolean;

```

```

1      function Pipe_Length_Is_Limited (File: POSIX_IO.File_Descriptor)
2      return Boolean;
3      function Pipe_Length_Limit (Pathname: POSIX.Pathname)           -- obsolescent
4      return POSIX.Pipe_Limit_Maxima;                                -- obsolescent
5      function Pipe_Length_Limit (File: POSIX_IO.File_Descriptor) -- obsolescent
6      return POSIX.Pipe_Limit_Maxima;                                -- obsolescent
7      function Pipe_Length_Maximum (Pathname: POSIX.Pathname)
8      return POSIX_Limits.Pipe_Length_Maxima
9      renames Pipe_Length_Limit;
10     function Pipe_Length_Maximum (File: POSIX_IO.File_Descriptor)
11     return POSIX_Limits.Pipe_Length_Maxima
12     renames Pipe_Length_Limit;
13     -- 5.4.2 File Restrictions
14     function Change_Owner_Is_Restricted (Pathname: POSIX.Pathname)
15     return POSIX_Options.Change_Owner_Restriction;
16     function Change_Owner_Is_Restricted (File: POSIX_IO.File_Descriptor)
17     return POSIX_Options.Change_Owner_Restriction;
18     function Filename_Is_Truncated (Pathname: POSIX.Pathname)
19     return POSIX_Options.Filename_Truncation;
20     function Filename_Is_Truncated (File: POSIX_IO.File_Descriptor)
21     return POSIX_Options.Filename_Truncation;
22     -- 5.4.3 Pathname-Specific Options
23     function Synchronized_IO_Is_Supported (Pathname: POSIX.Pathname)
24     return Boolean;
25     function Synchronized_IO_Is_Supported (File: POSIX_IO.File_Descriptor)
26     return Boolean;
27     function Asynchronous_IO_Is_Supported (Pathname: POSIX.Pathname)
28     return Boolean;
29     function Asynchronous_IO_Is_Supported (File: POSIX_IO.File_Descriptor)
30     return Boolean;
31     function Prioritized_IO_Is_Supported (Pathname: POSIX.Pathname)
32     return Boolean;
33     function Prioritized_IO_Is_Supported (File: POSIX_IO.File_Descriptor)
34     return Boolean;
35 end POSIX_Configurable_File_Limits;

```

36 5.4.1 File Limits

37 5.4.1.1 Synopsis

```

38     -- Link Limits
39     function Link_Is_Limited (Pathname: POSIX.Pathname)           -- obsolescent
40     return Boolean;                                                -- obsolescent
41     function Link_Is_Limited (File: POSIX_IO.File_Descriptor) -- obsolescent
42     return Boolean;                                                -- obsolescent
43     function Link_Limit (Pathname: POSIX.Pathname)               -- obsolescent
44     return POSIX.Link_Limit_Maxima;                                -- obsolescent
45     function Link_Limit (File: POSIX_IO.File_Descriptor)         -- obsolescent
46     return POSIX.Link_Limit_Maxima;                                -- obsolescent
47     function Links_Are_Limited (Pathname: POSIX.Pathname)
48     return Boolean
49     renames Link_Is_Limited;
50     function Links_Are_Limited (File: POSIX_IO.File_Descriptor)
51     return Boolean
52     renames Link_Is_Limited;
53     function Links_Maximum (Pathname: POSIX.Pathname)
54     return POSIX_Limits.Links_Maxima
55     renames Link_Limit;

```

```

1  function Links_Maximum (File: POSIX_IO.File_Descriptor)
2      return POSIX_Limits.Links_Maxima
3      renames Link_Limit;
4  -- Input line limits
5  function Input_Line_Is_Limited (Pathname: POSIX.Pathname)
6      return Boolean;
7  function Input_Line_Is_Limited (File: POSIX_IO.File_Descriptor)
8      return Boolean;
9  function Input_Line_Limit (Pathname: POSIX.Pathname)           -- obsolescent
10     return POSIX.Input_Line_Limit_Maxima;                     -- obsolescent
11  function Input_Line_Limit (File: POSIX_IO.File_Descriptor) -- obsolescent
12     return POSIX.Input_Line_Limit_Maxima;                     -- obsolescent
13  function Input_Line_Maximum (Pathname: POSIX.Pathname)
14     return POSIX_Limits.Input_Line_Maxima
15     renames Input_Line_Limit;
16  function Input_Line_Maximum (File: POSIX_IO.File_Descriptor)
17     return POSIX_Limits.Input_Line_Maxima
18     renames Input_Line_Limit;
19  -- Input queue limits
20  function Input_Queue_Is_Limited (Pathname: POSIX.Pathname)
21     return Boolean;
22  function Input_Queue_Is_Limited (File: POSIX_IO.File_Descriptor)
23     return Boolean;
24  function Input_Queue_Limit (Pathname: POSIX.Pathname)         -- obsolescent
25     return POSIX.Input_Queue_Limit_Maxima;                     -- obsolescent
26  function Input_Queue_Limit (File: POSIX_IO.File_Descriptor) -- obsolescent
27     return POSIX.Input_Queue_Limit_Maxima;                     -- obsolescent
28  function Input_Queue_Maximum (Pathname: POSIX.Pathname)
29     return POSIX_Limits.Input_Queue_Maxima
30     renames Input_Queue_Limit;
31  function Input_Queue_Maximum (File: POSIX_IO.File_Descriptor)
32     return POSIX_Limits.Input_Queue_Maxima
33     renames Input_Queue_Limit;
34  -- Filename and pathname limits
35  function Filename_Is_Limited (Pathname: POSIX.Pathname)
36     return Boolean;
37  function Filename_Is_Limited (File: POSIX_IO.File_Descriptor)
38     return Boolean;
39  function Filename_Limit (Pathname: POSIX.Pathname)           -- obsolescent
40     return POSIX.Filename_Limit_Maxima;                       -- obsolescent
41  function Filename_Limit (File: POSIX_IO.File_Descriptor)    -- obsolescent
42     return POSIX.Filename_Limit_Maxima;                       -- obsolescent
43  function Filename_Maximum (Pathname: POSIX.Pathname)
44     return POSIX_Limits.Filename_Maxima
45     renames Filename_Limit;
46  function Filename_Maximum (File: POSIX_IO.File_Descriptor)
47     return POSIX_Limits.Filename_Maxima
48     renames Filename_Limit;
49  function Pathname_Is_Limited (Pathname: POSIX.Pathname)
50     return Boolean;
51  function Pathname_Is_Limited (File: POSIX_IO.File_Descriptor)
52     return Boolean;
53  function Pathname_Limit (Pathname: POSIX.Pathname)           -- obsolescent
54     return POSIX.Pathname_Limit_Maxima;                       -- obsolescent
55  function Pathname_Limit (File: POSIX_IO.File_Descriptor)    -- obsolescent
56     return POSIX.Pathname_Limit_Maxima;                       -- obsolescent
57  function Pathname_Maximum (Pathname: POSIX.Pathname)
58     return POSIX_Limits.Pathname_Maxima
59     renames Pathname_Limit;

```

```

1      function Pathname_Maximum (File: POSIX_IO.File_Descriptor)
2      return POSIX_Limits.Pathname_Maxima
3      renames Pathname_Limit;
4      -- Pipe length limits
5      function Pipe_Length_Is_Limited (Pathname: POSIX.Pathname)
6      return Boolean;
7      function Pipe_Length_Is_Limited (File: POSIX_IO.File_Descriptor)
8      return Boolean;
9      function Pipe_Length_Limit (Pathname: POSIX.Pathname)           -- obsolescent
10     return POSIX.Pipe_Limit_Maxima;                                -- obsolescent
11     function Pipe_Length_Limit (File: POSIX_IO.File_Descriptor) -- obsolescent
12     return POSIX.Pipe_Limit_Maxima;                                -- obsolescent
13     function Pipe_Length_Maximum (Pathname: POSIX.Pathname)
14     return POSIX_Limits.Pipe_Length_Maxima
15     renames Pipe_Length_Limit;
16     function Pipe_Length_Maximum (File: POSIX_IO.File_Descriptor)
17     return POSIX_Limits.Pipe_Length_Maxima
18     renames Pipe_Length_Limit;

```

19 5.4.1.2 Description

20 There are two overloaded subprograms for each limit or option to be queried. One
21 accepts a pathname of a file or directory. The other accepts an open file descriptor.
22 (The parameters are named Pathname and File, respectively.)

23 The implementation shall support all of the limits and options identified by these
24 subprograms and may support others.

25 `Links_Are_Limited` shall return `True` if there is a limit to the number of links
26 permissible on the file, and `False` otherwise.

27 `Link_Is_Limited`, an alternate name for `Links_Are_Limited`, is obsolescent.

28 `Links_Maximum` shall provide the maximum value of the link count of a file. If
29 `Links_Are_Limited` for the given parameter returns `False`, then `Links_Maximum`
30 shall return `POSIX_Limits.Links_Maxima`'Last.

31 `Link_Limit`, an alternate name for `Links_Maximum`, is obsolescent.

32 `Input_Line_Is_Limited` shall return `True` if there is a maximum number of bytes
33 in a canonical input line as indicated by File or Pathname, and `False` otherwise.
34 The behavior of `Input_Line_Is_Limited` is unspecified if File or Pathname do
35 not refer to a terminal device.

36 `Input_Line_Maximum` shall provide the maximum number of bytes in a terminal
37 canonical input line identified by the File or Pathname. If `Input_Line_Is_Lim-`
38 `ited` for the given parameter returns `False`, the `Input_Line_Maximum` shall return
39 `POSIX_Limits.Input_Line_Maxima` 'Last. It is unspecified whether an imple-
40 mentation supports association of the `Input_Line_Maximum` if File or Pathname
41 does not refer to a terminal device.

42 `Input_Line_Limit`, an alternate name for `Input_Line_Maximum`, is obsolescent.

43 `Input_Queue_Is_Limited` shall return `True` if there is a maximum amount of
44 space available in the terminal input queue associated with File or Pathname, and

1 False otherwise. The behavior of `Input_Queue_Is_Limited` is unspecified if `File`
2 or `Pathname` does not refer to a terminal device.

3 `Input_Queue_Maximum` shall return the minimum number of bytes for which space
4 will be available in a terminal input queue associated with `File` or `Pathname`. This
5 is the maximum number of bytes a portable application may require to be typed
6 as input before reading those bytes. If `Input_Queue_Is_Limited` returns `False`
7 for the given parameter, then `Input_Queue_Maximum` shall return `POSIX_Limits`. -
8 `Input_Queue_Maxima` 'Last. It is unspecified whether an implementation supports
9 association of the `Input_Queue_Maximum` if `File` or `Pathname` does not refer to a
10 terminal device.

11 `Input_Queue_Limit`, an alternate name for `Input_Queue_Maximum`, is obsolescent.

12 `Filename_Is_Limited` shall return `True` if there is a limit to the length of a file-
13 name associated with `File` or `Pathname`, and `False` otherwise. If `File` or `Path-`
14 `name` refers to a directory, then `True` shall be returned to indicate that file names
15 within the indicated directory are limited. If `File` or `Pathname` do not refer to a
16 directory, then the behavior is unspecified.

17 `Filename_Maximum` shall return the maximum number of POSIX characters in
18 the file name associated with `File` or `Pathname`. If `File` or `Pathname` refer to a
19 directory, the value returned applies to the file names within the directory. If `File-`
20 `name_Is_Limited` returns `False` for the given parameter, then `Filename_Maximum`
21 shall return `POSIX_Limits.Filename_Maxima` 'Last. If `File` or `Pathname` do not
22 refer to a directory, then the behavior is unspecified.

23 `Filename_Limit`, an alternate name for `Filename_Maximum`, is obsolescent.

24 `Pathname_Is_Limited` shall return `True` if there is a limit to the length of any
25 pathname associated with `File` or `Pathname`, and `False` otherwise. If `File` or
26 `Pathname` refer to a directory, then `True` indicates that file names relative to the
27 named directory are limited. If `File` or `Pathname` do not refer to a directory, then
28 the behavior is unspecified.

29 `Pathname_Maximum` shall return the maximum number of POSIX characters in the
30 pathname associated with `File` or `Pathname`. If `File` or `Pathname` refer to a di-
31 rectory, the value returned applies to pathnames relative to the named directory. If
32 `Pathname_Is_Limited` returns `False` for the given parameter, then `Pathname_-`
33 `Maximum` shall return `POSIX_Limits.Pathname_Maxima` 'Last. If `File` or `Path-`
34 `name` do not refer to a directory, then the behavior is unspecified.

35 `Pathname_Limit`, an alternate name for `Pathname_Maximum`, is obsolescent.

36 `Pipe_Length_Is_Limited` shall return `True` if there is a limit to the number of
37 bytes that can be written at once to a FIFO special file named by `File` or `Path-`
38 `name`, and `False` otherwise. If `File` or `Pathname` refer to a directory, then this
39 characteristic applies to any FIFO special file created within the named directory. If
40 `File` or `Pathname` refer to any other type of file, the behavior is unspecified

41 `Pipe_Length_Maximum` shall return the maximum number of bytes that can be
42 written at once to a FIFO special file named by `File` or `Pathname`. If `Filename` or
43 `Pathname` refers to a directory, the value returned applies to any FIFO special files

that exists or can be created within the named directory. If `File` or `Pathname` refer 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.

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.

`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 or not the use of `Change_Owner_And_Group` (see §5.4.1) 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` refer 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` do 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` refer to a directory, the value returned shall apply to file names within the named directory. If `File` or `Pathname` do not refer to a directory, then the behavior is unspecified.

5.4.2.3 Error Handling

If any of the following conditions occur, 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.

`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

```
1  function Synchronized_IO_Is_Supported (Pathname: POSIX.Pathname)
2  return Boolean;
3  function Synchronized_IO_Is_Supported (File: POSIX_IO.File_Descriptor)
4  return Boolean;
5  function Asynchronous_IO_Is_Supported (Pathname: POSIX.Pathname)
6  return Boolean;
7  function Asynchronous_IO_Is_Supported (File: POSIX_IO.File_Descriptor)
8  return Boolean;
9  function Prioritized_IO_Is_Supported (Pathname: POSIX.Pathname)
10 return Boolean;
11 function Prioritized_IO_Is_Supported (File: POSIX_IO.File_Descriptor)
12 return Boolean;
13 function Prioritized_IO_Is_Supported (File: POSIX_IO.File_Descriptor)
14 return Boolean;
```

5.4.3.2 Description

16 Synchronized_IO_Is_Supported shall return True if the Synchronized I/O option
17 is supported for the file specified by File or Pathname.

18 Asynchronous_IO_Is_Supported shall return True if the Asynchronous I/O option
19 is supported for the file specified by File or Pathname.

20 Prioritized_IO_Is_Supported shall return True if the Prioritized I/O option is
21 supported for the file specified by File or Pathname.

22 If File or Pathname refers to a directory, it is unspecified whether the value of the
23 function is defined.

5.4.3.3 Error Handling

25 If any of the following conditions occur, the exception POSIX_Error shall be raised,
26 with the corresponding error code:

Filename_Too_Long

28 The length in POSIX characters of the specified pathname exceeds Pathname
29 Limit or the length in POSIX characters of a component of the specified path-
30 name is greater than Filename Maximum and the Filename Truncation option is
31 not supported for the pathname prefix of that component.

No_Such_File_Or_Directory

33 Pathname does not identify an existing file.

Not_A_Directory

35 A component of the path prefix of Pathname is not a directory.

Permission_Denied

37 Search permission is denied for a component of the path prefix of Pathname.

38 If any of the following conditions is detected, the exception POSIX_Error shall be
39 raised, with the corresponding error code:

1 **Bad_File_Descriptor**

2 File is not a valid file descriptor.

3 **Invalid_Argument**

4 The implementation does not define the value of the function for the speci-
5 fied file.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Section 6: Input and Output Primitives

1 This section contains the packages `POSIX_IO` and `POSIX_Asynchronous_IO`, in-
2 cluding all of the POSIX input/output operations, and the package `POSIX_File_-`
3 `Locking`, which provides support for file locking. The relationship between POSIX
4 I/O and Ada I/O is defined in §8.1.

5 6.1 Package `POSIX_IO`

```
6 with Ada_Streams,  
7     POSIX,  
8     POSIX_Permissions;  
9 package POSIX_IO is  
10    -- 6.1.1 Open or Create a File  
11    -- File Descriptors  
12    type File_Descriptor is range 0..POSIX.Open_Files_Maxima'Last - 1;  
13    Standard_Input: constant File_Descriptor := 0;  
14    Standard_Output: constant File_Descriptor := 1;  
15    Standard_Error: constant File_Descriptor := 2;  
16    type IO_Offset is implementation-defined-integer;  
17    -- File modes and options  
18    type File_Mode is (Read_Only, Write_Only, Read_Write);  
19    type Open_Option_Set is new POSIX.Option_Set;  
20    -- Empty_Set, "+", and "-" are derived operations  
21    Non_Blocking: constant Open_Option_Set := implementation-defined;  
22    Append: constant Open_Option_Set := implementation-defined;  
23    Truncate: constant Open_Option_Set := implementation-defined;  
24    Exclusive: constant Open_Option_Set := implementation-defined;  
25    Not_Controlling_Terminal: constant Open_Option_Set := implementation-defined;  
26    File_Synchronized: constant POSIX_IO.Open_Option_Set := implementation-defined;  
27    Data_Synchronized: constant POSIX_IO.Open_Option_Set := implementation-defined;  
28    Read_Synchronized: constant POSIX_IO.Open_Option_Set := implementation-defined;  
29    -- Operations to open or close file descriptors  
30    function Open  
31      (Name: POSIX.Pathname;  
32       Mode: File_Mode;  
33       Options: Open_Option_Set := Empty_Set;  
34       Masked_Signals: POSIX.Signal_Masking := POSIX.RTS_Signals)  
35      return File_Descriptor;  
36    function Open_Or_Create  
37      (Name: POSIX.Pathname;  
38       Mode: File_Mode;  
39       Permissions: POSIX_Permissions.Permission_Set;  
40       Options: Open_Option_Set := Empty_Set;  
41       Masked_Signals: POSIX.Signal_Masking := POSIX.RTS_Signals)  
42      return File_Descriptor;  
43    function Is_Open (File: File_Descriptor)  
44      return Boolean;  
45    procedure Close  
46      (File: in File_Descriptor;  
47       Masked_Signals: in POSIX.Signal_Masking := POSIX.RTS_Signals);  
48    function Duplicate  
49      (File: File_Descriptor;  
50       Target: File_Descriptor := 0)  
51      return File_Descriptor;
```

```

1  function Duplicate_And_Close
2    (File: File_Descriptor;
3     Target: File_Descriptor:= 0;
4     Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
5    return File_Descriptor;
6  procedure Create_Pipe
7    (Read_End: out File_Descriptor;
8     Write_End: out File_Descriptor);
9    -- 6.1.2 I/O Buffer Type
10 subtype IO_Buffer is POSIX.POSIX_String; -- obsolescent
11 -- 6.1.3 Read from a File
12 procedure Read -- obsolescent
13   (File: in File_Descriptor;
14    Buffer: out IO_Buffer;
15    Last: out POSIX.IO_Count;
16    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
17 procedure Read
18   (File: in File_Descriptor;
19    Buffer: out Ada_Streams.Stream_Element_Array;
20    Last: out Ada_Streams.Stream_Element_Offset;
21    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
22 generic
23   type T is private;
24 procedure Generic_Read
25   (File: in File_Descriptor;
26    Item: out T;
27    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
28 -- 6.1.4 Write to a File
29 procedure Write -- obsolescent
30   (File: in File_Descriptor;
31    Buffer: in IO_Buffer;
32    Last: out POSIX.IO_Count;
33    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
34 procedure Write
35   (File: in File_Descriptor;
36    Buffer: in Ada_Streams.Stream_Element_Array;
37    Last: out Ada_Streams.Stream_Element_Offset;
38    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
39 generic
40   type T is private;
41 procedure Generic_Write
42   (File: in File_Descriptor;
43    Item: in T;
44    Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
45 -- 6.1.5 File Position Operations
46 type Position is
47   (From_Beginning, From_Current_Position, From_End_Of_File);
48 procedure Seek
49   (File: in File_Descriptor;
50    Offset: in IO_Offset;
51    Result: out IO_Offset;
52    Starting_Point: in Position:= From_Beginning);
53 function File_Size (File: File_Descriptor)
54   return POSIX.IO_Count;
55 function File_Position (File: File_Descriptor)
56   return IO_Offset;

```

```

1      -- 6.1.6 Terminal Operations
2      function Is_A_Terminal (File: File_Descriptor)
3          return Boolean;
4      function Get_Terminal_Name (File: File_Descriptor)
5          return POSIX.Pathname;
6      -- 6.1.7 File Control
7      procedure Get_File_Control
8          (File: in File_Descriptor;
9           Mode: out File_Mode;
10          Options: out Open_Option_Set);
11      procedure Set_File_Control
12          (File: in File_Descriptor;
13          Options: in Open_Option_Set);
14      function Get_Close_On_Exec (File: File_Descriptor)
15          return Boolean;
16      procedure Set_Close_On_Exec
17          (File: in File_Descriptor;
18          To: in Boolean:= True);
19      -- 6.1.8 Update File Status Information
20      procedure Change_Permissions
21          (File: in POSIX_IO.File_Descriptor;
22          Permission: in POSIX_Permissions.Permission_Set);
23      -- 6.1.9 Truncate File to A Specified Length
24      procedure Truncate_File
25          (File: in POSIX_IO.File_Descriptor;
26          Length: in POSIX_IO.Count);
27      -- 6.1.10 Synchronize a File's State
28      procedure Synchronize_File (File: in POSIX_IO.File_Descriptor);
29      -- 6.1.11 Data Synchronization
30      procedure Synchronize_Data (File: in POSIX_IO.File_Descriptor);
31 end POSIX_IO;

```

32 6.1.1 Open or Create a File

33 6.1.1.1 Synopsis

```

34      -- File Descriptors
35      type File_Descriptor is range 0..POSIX.Open_Files_Maxima'Last - 1;
36      Standard_Input: constant File_Descriptor:= 0;
37      Standard_Output: constant File_Descriptor:= 1;
38      Standard_Error: constant File_Descriptor:= 2;
39      type IO_Offset is implementation-defined-integer;
40      -- File modes and options
41      type File_Mode is (Read_Only, Write_Only, Read_Write);
42      type Open_Option_Set is new POSIX.Option_Set;
43      -- Empty_Set, "+", and "-" are derived operations
44      Non_Blocking: constant Open_Option_Set:= implementation-defined;
45      Append: constant Open_Option_Set:= implementation-defined;
46      Truncate: constant Open_Option_Set:= implementation-defined;
47      Exclusive: constant Open_Option_Set:= implementation-defined;
48      Not_Controlling_Terminal: constant Open_Option_Set:= implementation-defined;
49      File_Synchronized: constant POSIX_IO.Open_Option_Set:= implementation-defined;
50      Data_Synchronized: constant POSIX_IO.Open_Option_Set:= implementation-defined;
51      Read_Synchronized: constant POSIX_IO.Open_Option_Set:= implementation-defined;

```

```

1      -- Operations to open or close file descriptors
2      function Open
3          (Name:          POSIX.Pathname;
4           Mode:          File_Mode;
5           Options:       Open_Option_Set:= Empty_Set;
6           Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
7      return File_Descriptor;
8      function Open_Or_Create
9          (Name:          POSIX.Pathname;
10         Mode:          File_Mode;
11         Permissions:   POSIX.Permissions.Permission_Set;
12         Options:       Open_Option_Set:= Empty_Set;
13         Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
14     return File_Descriptor;
15     function Is_Open (File: File_Descriptor)
16     return Boolean;
17     procedure Close
18         (File:          in File_Descriptor;
19         Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
20     function Duplicate
21         (File: File_Descriptor;
22         Target: File_Descriptor:= 0)
23     return File_Descriptor;
24     function Duplicate_And_Close
25         (File: File_Descriptor;
26         Target: File_Descriptor:= 0;
27         Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
28     return File_Descriptor;
29     procedure Create_Pipe
30         (Read_End: out File_Descriptor;
31         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_Maximal`. 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:

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 if 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

1 **Open_Option_Set** via the derived type semantics of Ada, from the operations avail-
2 able for **POSIX.Option_Set**. The user can create the option set by using the appro-
3 priate operations to create a set containing the options needed. In the following, the
4 term “open operation” means one of the operations **Open** or **Open_Or_Create**.

5 The meanings of the options are as follows:

6 **Non_Blocking**

7 This option shall denote the behavior of the open operations on FIFO files,
8 and on block special files or character special files that support nonblocking
9 opens.

10 (1) For a FIFO whose file mode is **Read_Only** or **Write_Only**:

11 (a) When set, an open operation shall return without delay. An **Open**
12 operation with file mode of **Write_Only** shall return an error if no
13 process currently has the file open for reading.

14 (b) When not set, an open operation with file mode of **Read_Only** shall
15 block until a process opens the file for writing. An open operation
16 with file mode of **Write_Only** shall block until a process opens the
17 file for reading.

18 (2) For a block special file or character special file that supports nonblock-
19 ing opens:

20 (a) When set, an open operation shall return without waiting for the de-
21 vice to be ready or available. Subsequent behavior of the device is
22 device-specific.

23 (b) When not set, an open operation shall wait until the device is ready
24 or available before returning.

25 (c) Otherwise, the behavior of an open operation with the **Non_Blocking**
26 option shall be undefined.

27 **Append**

28 This option shall denote where output shall be placed in the file. When
29 set, the file offset in the open file description shall be reset to the end of
30 the file prior to each write operation. When not set, the file offset shall be
31 updated as described by the **Write**, **Generic_Write** and **Seek** operations.
32 (See §6.1.4, §6.1.5.)

33 **Exclusive**

34 This option shall specify that **Open_Or_Create** shall fail if the file named
35 by the **Name** parameter exists. The check for existence and the creation of
36 the file if it does not exist shall be a single atomic action, with respect to
37 all other processes executing **Open_Or_Create** naming the same file within
38 the same directory that are also using this option. If this option is specified
39 for **Open**, the result is undefined.

40 **Truncate**

41 This option shall denote whether the file shall be truncated when opened.
42 When set, the file shall be truncated to zero length when opened. The mode
43 and owner are unchanged. If not set, truncation does not occur when the file
44 is opened. This option shall have no effect on FIFO special files or terminal
45 device files. Its effect on other file types is implementation defined. The
46 result of applying this option on a file opened with **Mode** specified as **Read_**
47 **Only** is undefined.

1 **Not_Controlling_Terminal**
2 This option shall specify whether the file shall be adopted as the terminal
3 controlling the process. (See §7.1.1.3.) If set and if the **Name** parameter
4 identifies a terminal device, it is unspecified whether that terminal device
5 becomes the controlling terminal for the process.

6 **Data_Synchronized**
7 Write operations on the file descriptor complete as defined by synchronized
8 I/O data integrity completion (see §2.2.2.137).

9 **File_Synchronized**
10 Write operations on the file descriptor complete as defined by synchronized
11 I/O file integrity completion (see §2.2.2.138).

12 **Read_Synchronized**
13 Read operations complete at the same level of integrity as specified by
14 **Data_Synchronized** and **File_Synchronized**. If both **Data_Synchro-**
15 **nized** and **Read_Synchronized** are specified, all I/O operations on the file
16 descriptor complete as defined by synchronized I/O data integrity comple-
17 tion. If both **File_Synchronized** and **Read_Synchronized** are specified,
18 all I/O operations on the file descriptor complete as defined by synchronized
19 I/O file integrity completion.

20 The **Open** operation shall open a file that already exists. The **Open_Or_Create**
21 operation shall either open an existing file or create a new file with the given name
22 if none exists. In both cases, the **Name** parameter specifies the name of the file to
23 be opened or created. These operations shall create a new open file description, and
24 shall return a value of type **File_Descriptor** referring to that open file description.
25 The **File_Descriptor** value shall be the lowest file descriptor not currently open
26 for that process. The file descriptor shall have the property that it is not closed by
27 the system when one of the **Exec** family of operations is called. (See §6.1.7.) The
28 open file description shall not be shared by any other process in the system. The file
29 offset shall be set to the beginning of the file. The options in the open file description
30 shall be the options provided by the **Options** parameter. The file access mode in the
31 open file description shall be the value provided by the **Mode** parameter. The effect
32 of calling **Open** or **Open_Or_Create** where **Name** specifies a FIFO special file and
33 **Mode** specifies **Read_Write** is undefined. The parameter **Masked_Signals** specifies
34 which signals shall be masked during the operation. (See §2.4.1.6.)

35 If **Open_Or_Create** creates a new file, the operation shall mark the Last Access Time,
36 the Last Modification Time and the Last Status Change Time for update. If an existing file
37 is opened, and the **Truncate** option is requested, the operation shall mark the last
38 modification time and the Last Status Change Time for update. (See §2.3.10.)

39 **Is_Open** shall indicate whether a file descriptor is open for I/O operations. This
40 function shall return a Boolean value but shall never raise an exception.

41 **Create_Pipe** shall create a pipe (a one-way communications channel). Upon return,
42 the parameter **Read_End** shall be the **File_Descriptor** value associated with the
43 pipe for Read operations. The parameter **Write_End** shall be the **File_Descrip-**
44 **tor** value associated with the pipe for Write operations. The file descriptor values
45 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 first-in-first-out 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 with the file descriptor **Write_End**. Upon successful completion, the operation shall mark for update the Last Access Time, file modification time and Last File Status Change Time for the pipe.

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. If the link count of the file is zero, when all file descriptors associated with the file have been closed, the space occupied by the file shall be freed and the file shall no longer be accessible. (See §5.2.3.)

If the Asynchronous I/O option is supported, when there is an outstanding cancelable asynchronous I/O 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, is 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 is the last close of a memory object and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object shall be removed.

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**.

1 — Otherwise, if `Target` is equal to `File`, `File` shall be returned, and the file shall
2 not be closed.

3 — Otherwise, if `Target` is open, it shall be closed. Then the value that is returned
4 shall equal `Target`. This file descriptor is opened and shall return to the same
5 open file description as the parameter `File`.

6 For example, if file descriptor 4 is open and `Duplicate_And_Close(4, 7)` is called:

7 — If file descriptor 7 is open, it shall be closed.

8 — Then the value returned shall be 7, and file descriptors 4 and 7 are now open to
9 the same open file description.

10 **6.1.1.3 Error Handling**

11 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
12 with the corresponding error code:

13 **`Permission_Denied`**

14 Upon `Open` or `Open_Or_Create`, search permission is denied on a compo-
15 nent of the path prefix, or the file exists and the permissions specified by
16 `Mode` are denied, or the file does not exist and write permission is denied for
17 the parent directory of the file to be created, or the `Truncate` option is set
18 and write permission is denied.

19 **`Interrupted_Operation`**

20 `Open`, `Open_Or_Create`, or `Close` was interrupted by a signal.

21 **`Filename_Too_Long`**

22 Upon `Open` or `Open_Or_Create`: The length in POSIX characters of the
23 specified pathname exceeds `Pathname Limit` or the length in POSIX characters
24 of a component of the specified pathname is greater than `Filename Maximum`
25 and the `Filename Truncation` option is not supported for the pathname prefix
26 of that component.

27 (See §5.4.2.)

28 **`No_Such_File_Or_Directory`**

29 Upon `Open`, the file does not exist.

30 **`Not_A_Directory`**

31 Upon `Open` or `Open_Or_Create`, a component of the parameter `Name` (except
32 the last component) is not a directory.

33 **`Is_A_Directory`**

34 Upon `Open` or `Open_Or_Create`, the parameter `Name` denotes a directory,
35 and the value of `Mode` is either `Read_Write` or `Write_Only`.

36 **`Too_Many_Open_Files`**

37 `Open`, `Open_Or_Create`, `Duplicate`, or `Create_Pipe` would cause the
38 program to have more than `Open Files Maximum` open file descriptors.

39 **`Too_Many_Open_Files_In_System`**

40 Upon `Open`, `Open_Or_Create`, or `Create_Pipe`, a systemwide implemen-
41 tation defined open file limit would be exceeded.

1 **No_Space_Left_On_Device**
2 Upon `Open_Or_Create`, the file is being created, and the device associated
3 with the file named by the parameter `Name` is full.

4 **No_Such_Device_Or_Address**
5 Upon `Open` or `Open_Or_Create`, the requested mode is `Write_Only` for
6 nonblocking I/O, and the file is a FIFO special file and no process has the
7 FIFO open for reading.

8 **Read_Only_File_System**
9 Upon `Open` or `Open_Or_Create`, the file system is read-only and the value
10 of parameter `Mode` is not `Read_Only`.

11 **File_Exists**
12 Upon `Open_Or_Create`, the value of the parameter `Options` has the op-
13 tion `Exclusive` set to `True` and the file named by parameter `Name` already
14 exists.

15 **Bad_File_Descriptor**
16 Upon `Close`, `Duplicate`, or `Duplicate_And_Close`, the file associated
17 with the parameter `File` is not open.

18 `Is_Open` shall not raise any exceptions.

19 **6.1.2 I/O Buffer Type**

20 **6.1.2.1 Synopsis**

21 **subtype** `IO_Buffer` **is** `POSIX.POSIX_String`; -- *obsolescent*

22 **6.1.2.2 Description**

23 The type `IO_Buffer` is used to represent a sequence of `POSIX_Character` values
24 for some I/O operations. An I/O operation transfers a number of `POSIX_Character`
25 values to or from an `IO_Buffer` object, returning the number of `POSIX_Character`
26 values transferred in an object of type `IO_Count`.

27 *The type `IO_Buffer` is obsolescent. It is preserved for compatibility with POSIX.5*
28 *{B3}, only. New applications should use `Ada_Streams.Stream_Element_Array`.*
29 *(See §2.4.1.4.)*

30 **6.1.3 Read from a File**

31 **6.1.3.1 Synopsis**

32 **procedure** `Read` -- *obsolescent*
33 (`File`: **in** `File_Descriptor`;
34 `Buffer`: **out** `IO_Buffer`;
35 `Last`: **out** `POSIX.IO_Count`;
36 `Masked_Signals`: **in** `POSIX.Signal_Masking`:= `POSIX.RTS_Signals`);
37 **procedure** `Read`
38 (`File`: **in** `File_Descriptor`;
39 `Buffer`: **out** `Ada_Streams.Stream_Element_Array`;
40 `Last`: **out** `Ada_Streams.Stream_Element_Offset`;
41 `Masked_Signals`: **in** `POSIX.Signal_Masking`:= `POSIX.RTS_Signals`);

```
1      generic
2      type T is private;
3      procedure Generic_Read
4      (File:          in File_Descriptor;
5       Item:          out T;
6       Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
```

7 6.1.3.2 Description

8 The Read operation shall transfer a number of values from the file associated with
9 the open file descriptor parameter File, into the parameter Buffer. Read shall
10 attempt to transfer enough data to completely fill the buffer, i.e., Read shall attempt
11 to read Buffer'Length values. On successful return, the parameter Last shall
12 contain the index of the last element of Buffer supplied by the Read operation. If
13 the parameter Buffer is a null array, then Last shall be Buffer'First - 1 and
14 Read shall have no other result. Two overloadings of Read are defined, operating on
15 arrays of POSIX.POSIX_Character and Ada.Streams.Stream_Element_Array.

16 *The version of Read with argument of type IO_Buffer is obsolescent.*

17 On a regular file or other file capable of seeking, Read shall start at the position
18 in the file given by the file offset in the open file description associated with the
19 file descriptor, File. Before a successful return from Read, the file offset shall be
20 incremented by the number of bytes occupied by the values actually read. On a file
21 not capable of seeking, Read shall start from the current position. The value of the
22 file offset associated with such a file is undefined.

23 The number of values read may be less than the number of values requested (i.e., the
24 length of the Buffer parameter). This can occur if the read request was interrupted
25 by a signal, or if the file descriptor refers to a pipe or FIFO or other special file that
26 has less than the requested amount of data immediately available for reading. For
27 example, a terminal device may return one typed line of data, where the number
28 of values in the line is less than the length of the Buffer parameter. If the Read
29 operation transfers all data, the value of Last shall be Buffer'Last.

30 If Read is interrupted by a signal before it reads any data, Read shall raise POSIX_
31 Error with error code Interrupted_Operation. If Read is interrupted by a signal
32 after it has successfully read some data, Read shall return the index of the last
33 value transferred to the Buffer parameter. Read shall never raise POSIX_Error
34 with error code Interrupted_Operation on a pipe or FIFO if the operation has
35 transferred any data.

36 No data transfer shall occur past the current end-of-file. If the starting position
37 is at or after the end-of-file, Read shall raise IO_Exceptions.End_Error. If the
38 parameter File refers to a device special file, the result of subsequent calls to Read
39 is implementation defined.

40 When an attempt is made to read from an empty pipe or FIFO special file:

- 41 (1) If no process has the pipe open for writing, Read shall raise IO_Exceptions. -
42 End_Error.

1 (2) If some process has the pipe open for writing, and the open file description asso-
2 ciated with the parameter `File` has the `Non_Blocking` option, `Read` shall raise
3 `POSIX_Error` with error code `Resource_Temporarily_Unavailable`.

4 (3) If some process has the pipe open for writing, and the open file description asso-
5 ciated with the parameter `File` does not have the `Non_Blocking` option, `Read`
6 shall block until some data is written or the pipe is closed by all processes that
7 had the pipe open for writing.

8 When an attempt is made to read a file (other than a pipe or FIFO) that supports
9 non-blocking reads and has no data currently available:

10 (1) If the open file description associated with the parameter `File` has the `Non_-`
11 `Blocking` option, `Read` shall raise `POSIX_Error` with error code `Resource_-`
12 `Temporarily_Unavailable`.

13 (2) If the open file description associated with the parameter `File` does not have
14 the `Non_Blocking` option, `Read` shall block until some data becomes available

15 The `Non_Blocking` option shall have no effect if there is some data available.

16 For any portion of a regular file, prior to end-of-file, that has not been written, `Read`
17 shall place the value `Stream_Element'(0)` or `POSIX_Character'val(0)` into the
18 appropriate position(s) in `Buffer`.

19 Upon successful completion, when `Read` successfully transfers any data, the `Last`
20 `Access Time` shall be marked for update.

21 The limit `Pipe Length Maximum` represents the number of bytes guaranteed to be trans-
22 ferred during a single `Read` operation where the parameter `File` represents a pipe
23 or a FIFO special file. An attempt to transfer quantities of values greater than this
24 limit may be interleaved with other I/O operations on the file or device.

25 The `Generic_Read` operation is instantiated with a user-supplied type. This opera-
26 tion shall execute one or more `Read` operations to obtain sufficient bytes to complete
27 the parameter `Item`. The number of values to be transferred shall be the size of the
28 parameter `Item` in bytes. `Generic_Read` shall either return, having transferred the
29 entire object into the parameter `Item` or shall raise an exception if unable to obtain
30 sufficient values to complete the object. If an exception is raised, the number of val-
31 ues actually transferred is unspecified and there is no way for the conforming POSIX
32 application to know this number. The implementation of `Generic_Read` shall retry if
33 the operation is blocked or is interrupted, until either the entire object is transferred
34 or an implementation defined limit on retries is reached. The effect of instantiating
35 `Generic_Read` is unspecified if the size of the parameter `Item` is greater than `Pipe`
36 `Length Maximum` and the external file is a pipe or FIFO special file.

37 For any given constrained type, instantiations of `Generic_Read` or `Generic_Write`
38 shall be invertible in the following sense: writing an object of the type to a file and
39 then reading the same file into a second object of the type shall result in both objects
40 having the same value. Instantiations for unconstrained types may be refused by the
41 implementation.

42 Upon successful completion, when `Generic_Read` successfully transfers any data,
43 the `Last Access Time` shall be marked for update.

The following additional specifications apply if the Synchronized I/O option is supported:

- 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 §2.2.2.137).
- 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 §2.2.2.138).
- 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 §2.2.2.138).

If `File` refers to a shared memory object or any object other than a file, the result of `Read` or `Generic_Read` is unspecified.

6.1.3.3 Error Handling

If any of the following conditions occur, 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 is 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);

```

```

1      procedure Write
2      (File:           in File_Descriptor;
3       Buffer:         in Ada_Streams.Stream_Element_Array;
4       Last:           out Ada_Streams.Stream_Element_Offset;
5       Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
6      generic
7      type T is private;
8      procedure Generic_Write
9      (File:           in File_Descriptor;
10     Item:            in T;
11     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 only room on the device for 20 values, then `Write` shall transfer 20 values, and return the value 20 for the parameter `Last`. The next write request (for a non-zero 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:

- (1) 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.
- (2) 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. `Writes` 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 has been read, it shall transfer at least `Pipe Length Maximum` bytes.

- 1 (ii) If no data can be written, the `Write` operation shall transfer no data and
2 shall raise `POSIX_Error` with error code `Resource_Temporarily_Un-`
3 `available`.

4 When an attempt is made to write to a file or device (other than a pipe or FIFO),
5 associated with the file descriptor `File`, that supports nonblocking writes and cannot
6 accept the data immediately:

- 7 (1) If the open file description associated with the file descriptor `File` does not have
8 the `Non_Blocking` option, the `Write` operation shall block until the data can be
9 accepted or written.
- 10 (2) If the open file description associated with the file descriptor `File` has the `Non_`
11 `Blocking` option, the `Write` operation shall not block. If some data can be
12 written without blocking, the `Write` operation shall transfer as much data as
13 it can without blocking, and the index of the last value from `Buffer` that was
14 transferred shall be returned in the parameter `Last`. Otherwise, the `Write`
15 operation shall not transfer any data, and shall raise `POSIX_Error`, with error
16 code `Resource_Temporarily_Unavailable`.

17 Upon successful completion, when `Write` successfully transfers any data, the `Last`
18 `Access Time` and `Last Modification Time` shall be marked for update.

19 The `Generic_Write` operation is instantiated with a user-supplied type. This op-
20 eration shall execute one or more `Write` operations to transfer sufficient bytes to
21 transfer the parameter `Item`. The number of values to be transferred shall be the
22 size of the parameter `Item` in bytes. `Generic_Write` shall either return, having
23 transferred the entire object from the parameter `Item` or shall raise an exception if
24 unable to transfer sufficient values to complete the object. If an exception is raised,
25 the number of values actually transferred is unspecified and there is no way for
26 the conforming POSIX application to know this number. The implementation of
27 `Generic_Write` shall retry if the operation is blocked or is interrupted, until ei-
28 ther the entire object is transferred or an implementation defined limit on retries is
29 reached. The effect of instantiating `Generic_Write` is unspecified if the size in bytes
30 of the parameter `Item` is greater than the limit returned by `Pipe Length Maximum` and
31 the external file is a pipe or FIFO special file.

32 For any given constrained type, instantiations of `Generic_Read` or `Generic_Write`
33 shall be invertible in the following sense: writing an object of the type to a file and
34 then reading the same file into a second object of the type shall result in both objects
35 having the same value. Instantiations for unconstrained types may be refused by the
36 implementation.

37 Upon successful completion, when `Generic_Write` successfully transfers any data,
38 the `Last Access Time` and `Last Modification Time` shall be marked for update.

39 *The following additional specifications apply if the `Synchronized I/O` option is sup-*
40 *ported:*

- 41 — If `Data_Synchronized` has been specified, write I/O operations on the file de-
42 scriptor complete as defined by synchronized I/O data integrity completion (see
43 §2.2.2.137).

- 1 — If `File_Synchronized` has been specified, write I/O operations on the file de-
2 scriptor complete as defined by synchronized I/O file integrity completion (see
3 §2.2.2.138).
- 4 — If `Data_Synchronized` and `File_Synchronized` have been specified, write
5 I/O operations on the file descriptor complete as defined by synchronized I/O file
6 integrity completion (see §2.2.2.138).
- 7 — If `Data_Synchronized` and `Read_Synchronized` have been specified, write
8 I/O operations on the file descriptor complete as defined by synchronized I/O
9 data integrity completion (see §2.2.2.137).
- 10 — If `File_Synchronized` and `Read_Synchronized` have been specified, write
11 I/O operations on the file descriptor complete as defined by synchronized I/O file
12 integrity completion (see §2.2.2.138).
- 13 — If `Data_Synchronized`, `File_Synchronized`, and `Read_Synchronized` have
14 been specified, write I/O operations on the file descriptor complete as defined by
15 synchronized I/O file integrity completion (see §2.2.2.138).

16 If `File` refers to a shared memory object or any object other than a file, the result of
17 `Write` or `Generic_Write` is unspecified.

18 **6.1.4.3 Error Handling**

19 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
20 with the corresponding error code:

21 **Bad_File_Descriptor**

22 The file descriptor `File` is not open.

23 The open file description associated with the file descriptor `File` is not open
24 for writing.

25 **Input_Output_Error**

26 The implementation supports job control, a process in a background process
27 group attempts to write to its controlling terminal, and either the process
28 group of the process is orphaned or the process ignores the signal `POSIX_-`
29 `Signals.Signal_Terminal_Output`.

30 A write was attempted to a terminal device and a modem disconnect was
31 detected by the terminal interface for that device. (See §7.1.1.10.)

32 **Interrupted_Operation**

33 The operation was interrupted by a signal.

34 **Resource_Temporarily_Unavailable**

35 If the `Non_Blocking` option is set in the open file description associated
36 with the file descriptor `File` and the task would be delayed in completing
37 the operation, the operation shall fail rather than blocking.

38 **Broken_Pipe**

39 The parameter `File` is associated with a pipe or FIFO special file whose
40 read end is not open.

41 **File_Too_Large**

42 The file size exceeds a system limit.

43 **No_Space_Left_On_Device**

44 No space remains on the device associated with the parameter `File`.

6.1.5 File Position Operations

6.1.5.1 Synopsis

```
1  type Position is
2      (From_Beginning, From_Current_Position, From_End_Of_File);
3  procedure Seek
4      (File:          in File_Descriptor;
5       Offset:        in IO_Offset;
6       Result:        out IO_Offset;
7       Starting_Point: in Position:= From_Beginning);
8  function File_Size (File: File_Descriptor)
9      return POSIX.IO_Count;
10 function File_Position (File: File_Descriptor)
11     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.

The type IO_Offset shall be used to represent a location within a file, expressed in terms of bytes. This type contains 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.

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

1 `Seek`, `File_Position`, or `File_Size` operations on such devices is implementation
2 defined.

3 The result of `Seek`, `File_Position`, or `File_Size` on shared memory objects and
4 any object other than a file is unspecified.

5 **6.1.5.3 Error Handling**

6 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
7 with the corresponding error code:

8 `Bad_File_Descriptor`

9 The parameter `File` is not open.

10 `Invalid_Seek`

11 The file associated with the parameter `File` is a pipe or a FIFO special file.

12 **6.1.6 Terminal Operations**

13 **6.1.6.1 Synopsis**

```
14 function Is_A_Terminal (File: File_Descriptor)
15     return Boolean;
16 function Get_Terminal_Name (File: File_Descriptor)
17     return POSIX.Pathname;
```

18 **6.1.6.2 Description**

19 `Is_A_Terminal` shall return `True` if the external file associated with the parameter
20 `File` is a terminal device, and `False` otherwise.

21 `Get_Terminal_Name` shall return a value of type `Pathname` that represents the
22 name of the external terminal device associated with the parameter `File`.

23 The null string shall be returned if the value of the parameter `File` is not a valid
24 file descriptor associated with a terminal or if the pathname cannot be determined.

25 **6.1.6.3 Error Handling**

26 This standard does not specify any error conditions that are required to be detected
27 for these operations. Errors may be reported under conditions that are unspecified
28 by this standard.

29 **6.1.7 File Control**

30 **6.1.7.1 Synopsis**

```
31 procedure Get_File_Control
32     (File: in File_Descriptor;
33      Mode: out File_Mode;
34      Options: out Open_Option_Set);
35 procedure Set_File_Control
36     (File: in File_Descriptor;
37      Options: in Open_Option_Set);
```



```
1      function Get_Close_On_Exec (File: File_Descriptor)
2          return Boolean;
3      procedure Set_Close_On_Exec
4          (File: in File_Descriptor;
5           To:   in Boolean:= True);
```

6.1.7.2 Description

The type `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 and `File` refers to a shared memory object or any object that other than a file, the effect of `Set_File_Control` is unspecified.

When one program executes one of the `exec` family of operations, files may be closed across the call or may remain open. (The `exec` family refers to the `Exec` and `Exec_Search` procedures in `POSIX_Unsafe_Process_Primitives` and the `Start_Process` and `Start_Process_Search` procedures in `POSIX_Process_Primitives`.) Each open file descriptor shall have associated with it information indicating whether the file descriptor should be closed by the `exec` 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`.

6.1.7.3 Error Handling

If any of the following conditions occur, 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 below.

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 and 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 and other permissions specified in the Permissions parameter may be ignored.

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 the calling process does not have appropriate privileges, and if the group ID of the file does not match the effective group ID or one of the supplementary group IDs, and one or more of Owner_Execute, Group_Execute, and 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 occur, the exception POSIX_Error shall be raised, with the corresponding error code:

Bad_File_Descriptor
The File parameter is invalid.

1 **Operation_Not_Implemented**

2 This implementation does not support the `Change_Permissions` proce-
3 dure.

4 **Operation_Not_Permitted**

5 The effective user ID does not match the owner of the file and the calling
6 process does not have the appropriate privileges.

7 **Read_Only_File_System**

8 The file resides on a read-only file system.

9 If any of the following conditions is detected, the exception `POSIX_Error` shall be
10 raised, with the corresponding error code:

11 **Invalid_Argument**

12 File refers to a pipe and the implementation does not support this opera-
13 tion for pipes.

14 **6.1.9 Truncate File to A Specified Length**

15 **6.1.9.1 Synopsis**

16 **procedure** `Truncate_File`
17 (`File`: **in** `POSIX_IO.File_Descriptor`;
18 `Length`: **in** `POSIX_IO.Count`);

19 **6.1.9.2 Description**

20 *The functionality described in this subclause is optional. If neither the Memory Mapped*
21 *Files option nor the Shared Memory Objects option is supported, the implementation may*
22 *cause all calls to the explicitly declared operations defined in this subclause to raise*
23 *POSIX_Error. Otherwise, the behavior shall be as specified below.*

24 If `File` refers to a regular file open for writing, `Truncate_File` shall cause the file
25 to be truncated to `Length` bytes. If the length of the file previously exceeded `Length`
26 bytes, the extra data shall be discarded. If the file previously was smaller than this
27 length, it is unspecified whether the file is changed or its length increased. If the file
28 is extended, the extended area shall appear as if it were zero-filled.

29 If `File` refers to a shared memory object, `Truncate_File` sets the length of the
30 shared memory object to `Length` bytes.

31 If the file does not refer to a regular file or a shared memory object, the result is
32 unspecified.

33 If the effect of `Truncate_File` is to decrease the length of a file or shared memory
34 object and whole pages beyond the new end were previously mapped, then the whole
35 pages beyond the new end shall be discarded and references to them shall result in
36 generation of the signal `Signal_Bus_Error`, which the Ada language implementa-
37 tion shall convert to the exception `Program_Error`.

38 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.

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.

6.1.9.3 Error Handling

If any of the following conditions occur, 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.1b {3}.

6.1.10 Synchronize a File's State

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 below.

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 is 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 §2.2.2.136). All I/O operations shall be completed as defined for synchronized I/O file integrity completion (see §2.2.2.138).

6.1.10.3 Error Handling

If any of the following conditions occur, 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.

In the event that any of the queued I/O operations fail, `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 below.

`Synchronize_Data` forces all currently queued I/O operations associated with the file indicated by `File` to the synchronized I/O completion state (see §2.2.2.136).

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 §2.2.2.137).

The hardware characteristics upon which the implementation relies to assure that data is successfully transferred are implementation defined.

6.1.11.3 Error Handling

If any of the following conditions occur, 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.

In the event that any of the queued I/O operations fail, `Synchronize_Data` shall return the error conditions defined for `POSIX_IO.Read` and `POSIX_IO.Write`.

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 if 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 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 `Result.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 was (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 occur, 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 below.

```
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);
```

```

1      -- 6.3.2 Attributes of AIO Control Blocks
2      type List_IO_Operations is
3          (No_Op,
4           Read,
5           Write);
6      type IO_Array_Pointer is access Ada_Streams.Stream_Element_Array;
7      function Get_File (AD: AIO_Descriptor)
8          return POSIX_IO.File_Descriptor;
9      procedure Set_File
10         (AD: in AIO_Descriptor;
11          File: in POSIX_IO.File_Descriptor);
12      function Get_Offset (AD: AIO_Descriptor) return POSIX_IO.IO_Offset;
13      procedure Set_Offset
14         (AD: in AIO_Descriptor;
15          Offset: in POSIX_IO.IO_Offset);
16      function Get_Buffer (AD: AIO_Descriptor) return IO_Array_Pointer;
17      procedure Set_Buffer
18         (AD: in AIO_Descriptor;
19          Buffer: in IO_Array_Pointer);
20      function Get_Length (AD: AIO_Descriptor)
21          return Ada_Streams.Stream_Element_Count;
22      procedure Set_Length
23         (AD: in AIO_Descriptor;
24          Length: in Ada_Streams.Stream_Element_Count);
25      function Get_Priority_Reduction (AD: AIO_Descriptor) return Natural;
26      procedure Set_Priority_Reduction
27         (AD: in AIO_Descriptor;
28          Priority_Reduction: in Natural);
29      function Get_Event (AD: AIO_Descriptor)
30          return POSIX_Signals.Signal_Event;
31      procedure Set_Event
32         (AD: in AIO_Descriptor;
33          Event: in POSIX_Signals.Signal_Event);
34      function Get_Operation (AD: AIO_Descriptor)
35          return List_IO_Operations;
36      procedure Set_Operation
37         (AD: in AIO_Descriptor;
38          Operation: in List_IO_Operations);
39      -- 6.3.3 Asynchronous Read
40      procedure Read (AD: in AIO_Descriptor);
41      -- 6.3.4 Asynchronous Write
42      procedure Write (AD: in AIO_Descriptor);
43      -- 6.3.5 List Directed I/O
44      type AIO_Descriptor_List is
45          array (Positive range <>) of AIO_Descriptor;
46      procedure List_IO_No_Wait
47         (List: in out AIO_Descriptor_List;
48          Event: in POSIX_Signals.Signal_Event);
49      procedure List_IO_Wait
50         (List: in out AIO_Descriptor_List;
51          Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
52      -- 6.3.6 Retrieve Status of Asynchronous I/O Request
53      type AIO_Status is
54          (In_Progress,
55           Completed_Successfully,
56           Canceled);

```

```

1      function Get_AIO_Status
2          (AD: AIO_Descriptor)
3      return AIO_Status;
4      function Get_AIO_Error_Code
5          (AD: AIO_Descriptor)
6      return POSIX.Error_Code;
7      -- 6.3.7 Retrieve Bytes Transferred by Asynchronous I/O Request
8      function Get_Bytes_Transferred
9          (AD: AIO_Descriptor) return Ada.Streams.Stream_Element_Count;
10     -- 6.3.8 Cancel Asynchronous I/O Request
11     type Cancellation_Status is
12         (Canceled,
13          Not_Canceled,
14          All_Done);
15     function Cancel
16         (AD: AIO_Descriptor)
17     return Cancellation_Status;
18     function Cancel
19         (File: POSIX_IO.File_Descriptor)
20     return Cancellation_Status;
21     -- 6.3.9 Wait for Asynchronous I/O Request to Complete
22     procedure Await_IO_Or_Timeout
23         (AD: in AIO_Descriptor;
24          Timeout: in POSIX.Timespec;
25          Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
26     procedure Await_IO
27         (AD: in AIO_Descriptor;
28          Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
29     procedure Await_IO_Or_Timeout
30         (List: in AIO_Descriptor_List;
31          Timeout: in POSIX.Timespec;
32          Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
33     procedure Await_IO
34         (List: in AIO_Descriptor_List;
35          Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
36     -- 6.3.10 Asynchronous File and Data Synchronization
37     procedure Synchronize_File (AD: in AIO_Descriptor);
38     procedure Synchronize_Data (AD: in AIO_Descriptor);
39     private
40         implementation-defined
41     end POSIX_Asynchronous_IO;

```

42 6.3.1 AIO Descriptor Type

43 6.3.1.1 Synopsis

```

44     type AIO_Descriptor is private;
45     function Create_AIO_Control_Block
46     return AIO_Descriptor;
47     procedure Destroy_AIO_Control_Block
48         (AD: in out AIO_Descriptor);

```

49 6.3.1.2 Description

50 Values of the type `AIO_Descriptor` are used as asynchronous I/O (AIO) descriptors.
51 A valid *AIO descriptor* is a reference to an asynchronous I/O control block, which

1 is not visible. A value of type `AIO_Descriptor` is a valid AIO descriptor if-and-
2 only if it was returned by a call to `Create_AIO_Control_Block`, and has not been
3 subsequently invalidated, by a call to `Destroy_AIO_Control_Block`.

4 The effect of assigning the value of one variable of type `AIO_Descriptor` to another
5 variable of type `AIO_Descriptor` is to make a copy of the reference.

6 The application is responsible for the creation of the reference and the allocation of
7 the control block, using `Create_AIO_Control_Block`. The application is respon-
8 sible for the destruction of the reference and the deallocation of the control block,
9 using `Destroy_AIO_Control_Block`.

10 `Create_AIO_Control_Block` shall allocate a control block, and return a value of
11 type `AIO_Descriptor` that refers to it. Since `Create_AIO_Control_Block` is a
12 function, the implementation shall be such that the value returned can be assigned
13 to a variable of type `AIO_Descriptor`.

14 A call to `Destroy_AIO_Control_Block` shall deallocate the control block to which
15 the AD parameter refers, and destroy the reference, unless the object corresponds to
16 an asynchronous I/O request that is still being processed.

17 While an asynchronous I/O control block is in use by the system for an asynchronous
18 I/O operation, the application is responsible for preserving the control block and the
19 asynchronous I/O buffer to which it refers, unchanged. In general, the effect of per-
20 forming an operation that modifies or deallocates an asynchronous I/O control block
21 or asynchronous I/O buffer while it is being used by the system, is undefined.

22 If `Destroy_AIO_Control_Block` is called for an asynchronous I/O control block
23 that corresponds to an asynchronous I/O request that is still being processed, and the
24 system detects this, the call shall raise `POSIX_Error` with error code `Operation_-`
25 `Not_Permitted`.

26 NOTE: An application can discover when the system is no longer using an asynchronous
27 I/O control block via the functions `Get_AIO_Status` and `Get_AIO_Error_Code`, which
28 return `In_Progress` or `Operation_In_Progress` so long as the control block is in use.

29 6.3.1.3 Error Handling

30 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
31 with the corresponding error code:

32 No_Space_Left_On_Device

33 Upon `Create_AIO_Control_Block`, a resource other than memory re-
34 quired to initialize the AIO control block has been exhausted.

35 Not_Enough_Space

36 Upon `Create_AIO_Control_Block`, insufficient memory is available.

37 Operation_Not_Implemented

38 `Create_AIO_Control_Block` and `Destroy_AIO_Control_Block` are not
39 supported by this implementation.

40 If any of the following conditions is detected, the exception `POSIX_Error` shall be
41 raised, with the corresponding error code:

- 1 **Invalid_Argument**
- 2 Upon `Destroy_AIO_Control_Block`, the value of `AD` is not a valid AIO
- 3 descriptor. For this error, the implementation under some circumstances
- 4 may instead raise `Constraint_Error`.
- 5 **Operation_Not_Permitted**
- 6 `Destroy_AIO_Control_Block` was invoked when the corresponding asyn-
- 7 chronous I/O operation was in progress.

8 **6.3.2 Attributes of AIO Control Blocks**

9 **6.3.2.1 Synopsis**

```

10     type List_IO_Operations is
11         (No_Op,
12          Read,
13          Write);
14     type IO_Array_Pointer is access Ada_Streams.Stream_Element_Array;
15     function Get_File (AD: AIO_Descriptor)
16         return POSIX_IO.File_Descriptor;
17     procedure Set_File
18         (AD:     in AIO_Descriptor;
19          File: in POSIX_IO.File_Descriptor);
20     function Get_Offset (AD: AIO_Descriptor) return POSIX_IO.IO_Offset;
21     procedure Set_Offset
22         (AD:     in AIO_Descriptor;
23          Offset: in POSIX_IO.IO_Offset);
24     function Get_Buffer (AD: AIO_Descriptor) return IO_Array_Pointer;
25     procedure Set_Buffer
26         (AD:     in AIO_Descriptor;
27          Buffer: in IO_Array_Pointer);
28     function Get_Length (AD: AIO_Descriptor)
29         return Ada_Streams.Stream_Element_Count;
30     procedure Set_Length
31         (AD:     in AIO_Descriptor;
32          Length: in Ada_Streams.Stream_Element_Count);
33     function Get_Priority_Reduction (AD: AIO_Descriptor) return Natural;
34     procedure Set_Priority_Reduction
35         (AD:     in AIO_Descriptor;
36          Priority_Reduction: in Natural);
37     function Get_Event (AD: AIO_Descriptor)
38         return POSIX_Signals.Signal_Event;
39     procedure Set_Event
40         (AD:     in AIO_Descriptor;
41          Event: in POSIX_Signals.Signal_Event);
42     function Get_Operation (AD: AIO_Descriptor)
43         return List_IO_Operations;
44     procedure Set_Operation
45         (AD:     in AIO_Descriptor;
46          Operation: in List_IO_Operations);

```

6.3.2.2 Description

An asynchronous I/O control block has (at least) the following attributes:

File

The file descriptor on which the asynchronous I/O 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 asynchronous I/O operation, the value of the file offset for the file is unspecified.

Buffer

A pointer to the buffer into which the data is to be read, or from which the data is 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 both the Prioritized I/O option and the Priority Process Scheduling option are supported, then asynchronous I/O is queued in priority order, with the priority of each asynchronous operation based on the current process priority of the calling process. The attribute Priority_Reduction can be used to lower (but not raise) the asynchronous I/O 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 *asynchronous I/O priority* of an asynchronous I/O request is computed as the process priority minus the Priority_Reduction attribute of the asynchronous I/O control block.

The asynchronous I/O priority assigned to each asynchronous I/O request is an indication of its desired order of execution relative to other asynchronous I/O 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;

1 the order of processing of requests of the same priority issued to files that
2 are not character special files is unspecified. Numerically higher priority
3 values indicate requests of higher priority. The value of the `Priority_Reduction`
4 attribute of the asynchronous I/O control block shall have no effect on
5 process or task scheduling priority. When prioritized asynchronous I/O re-
6 quests to the same file are blocked waiting for a resource required for the
7 I/O operations and the resource subsequently becomes available, the higher-
8 priority requests shall be granted the resource before lower-priority requests
9 are granted the resource. The relative priority of asynchronous I/O and syn-
10 chronous I/O is implementation defined.

11 If the `Prioritized I/O` option is supported, the implementation shall define for
12 which files I/O prioritization is supported.

13 Event

14 A specification of how the calling process is to be notified of completion of
15 the I/O operation. This shall be interpreted as specified in §3.3.12. If the
16 `Notification` attribute of `Event` equals `No_Notification`, then no signal shall
17 be generated on I/O completion, but the status of the request shall be set
18 appropriately.

19 Operation

20 The I/O operation to be performed. This is used only by `List_IO_No_`
21 `Wait` and `List_IO_Wait` (see §6.3.5), which allow multiple asynchronous
22 I/O operations to be submitted at a single time.

23 Implementations may add other attributes, as permitted in §1.3.1.1, point (2).

24 NOTE: Adding attributes which may change the behavior of applications with respect
25 to this standard when those attributes are uninitialized also requires that the extension
26 be activated as described in §1.3.1.1.

27 Values of the type `List_IO_Operations` are used to specify an I/O operation to be
28 performed, for use with the list directed I/O operations (see §6.3.5). The interpreta-
29 tions of values of this type, when they appear as the value of the `Operation` attribute
30 of the asynchronous I/O control block associated with an entry in an asynchronous
31 I/O descriptor list, are:

32 No_Op

33 The list entry shall be ignored.

34 Read

35 An I/O operation shall be submitted as if by a call to `Read` with `AD` equal to
36 the list entry.

37 Write

38 An I/O operation shall be submitted as if by a call to `Write` with `AD` equal
39 to the list entry.

40 The function whose name is of the form `Get_X`, where `X` is the name of one of the at-
41 tributes of an asynchronous I/O control block, shall return the value of the `X` attribute
42 of the argument.

43 The procedure whose name is of the form `Set_X` shall set the `X` attribute of the object
44 specified by the first argument to the value specified by the second argument.

1 Implementations may add other attributes, as permitted in §1.3.1.1, point (2).

2 NOTE: Adding attributes which may change the behavior of applications with respect
3 to this standard when those attributes are uninitialized also requires that the extension
4 be activated as described in §1.3.1.1.

5 **6.3.2.3 Error Handling**

6 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
7 with the corresponding error code:

8 `Operation_Not_Implemented`

9 The operation is not supported by this implementation.

10 If any of the following conditions is detected, the exception `POSIX_Error` shall be
11 raised, with the corresponding error code:

12 `Invalid_Argument`

13 The value of `AD` is not a valid AIO descriptor. For this error, the implemen-
14 tation under some circumstances may instead raise `Constraint_Error`.

15 **6.3.3 Asynchronous Read**

16 **6.3.3.1 Synopsis**

17 **procedure** `Read` (`AD`: **in** `AIO_Descriptor`);

18 **6.3.3.2 Description**

19 The `Read` operation allows the calling process to read the number of bytes specified
20 by the `Length` attribute of `AD` from the file specified by the `File` attribute, into the
21 buffer designated by the `Buffer` attribute. The procedure call returns when the read
22 operation has been initiated or queued to the file or device (even when the data
23 cannot be delivered immediately).

24 If the `Prioritized I/O` option is supported for this file and the `Priority Process Scheduling`
25 option is supported, then the request is submitted with asynchronous I/O priority as
26 specified in §6.3.2.2.

27 Provided that the `Read` request is successfully queued, `AD` can be used as an argu-
28 ment to `Get_AIO_Status`, `Get_AIO_Error_Code`, and `Get_Bytes_Transferred`
29 in order to retrieve the status of the asynchronous I/O request during its *lifetime* (see
30 §6.3.6).

31 The attributes of `AD` shall be interpreted as defined in §6.3.2. The `Operation` attribute
32 of the asynchronous I/O control block shall be ignored.

33 Concurrent asynchronous I/O requests using the same value of the `AD` parameter
34 produce undefined behavior.

35 If synchronized I/O is enabled for the `File` attribute of `AD`, the behavior of this pro-
36 cedure shall be according to the definitions of `File Synchronization` (see §6.1.10) and
37 `Data Synchronization` (see §6.1.11).

For any system action that changes the process memory space while an asynchronous I/O 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 occur, the exception `POSIX_Error` shall be raised, with the corresponding error code:

`Resource_Temporarily_Unavailable`

The requested asynchronous I/O operation was not queued due to system resource limitations.

`Operation_Not_Implemented`

The Read procedure 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 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.

In the case that 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 `POSIX_IO.Write`), 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 is supported for this file and the Priority Process Scheduling option is supported, then the request is submitted with asynchronous I/O priority as specified in §6.3.2.2.

1 Provided that the `Write` request is successfully queued, `AD` can be used as an argu-
2 ment to `Get_AIO_Status`, `Get_AIO_Error_Code`, and `Get_Bytes_Transferred`
3 in order to retrieve the status of the asynchronous I/O request during its *lifetime* (see
4 §6.3.6).

5 The attributes of `AD` shall be interpreted as defined in §6.3.2. The `Operation` attribute
6 of the asynchronous I/O control block shall be ignored.

7 Concurrent asynchronous requests using the same value of the `AD` parameter pro-
8 duce undefined behavior.

9 If the Synchronized I/O option is supported on the file specified by the `File` attribute
10 of `AD`, the behavior of this procedure shall be according to the definitions of `File`
11 Synchronization (see §6.1.10) and `Data Synchronization` (see §6.1.11).

12 For any system action that changes the process memory space while an asynchronous
13 I/O is outstanding to the address range being changed, the result of that action is
14 undefined.

15 **6.3.4.3 Error Handling**

16 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
17 with the corresponding error code:

18 **`Resource_Temporarily_Unavailable`**

19 The requested asynchronous I/O request was not queued due to system re-
20 source limitations.

21 **`Operation_Not_Implemented`**

22 The `Write` procedure is not supported by this implementation.

23 If any of the following conditions is detected, the exception `POSIX_Error` shall be
24 raised, with the corresponding error code:

25 **`Invalid_Argument`**

26 The value of `AD` is not a valid AIO descriptor. For this error, the implemen-
27 tation under some circumstances may instead raise `Constraint_Error`.

28 Certain conditions may be detected synchronously at the time of the call to `Write`,
29 or asynchronously. If any of the following conditions is detected synchronously, the
30 exception `POSIX_Error` shall be raised with the corresponding error code:

31 **`Bad_File_Descriptor`**

32 The `File` attribute of `AD` is not a valid file descriptor open for writing.

33 **`Invalid_Argument`**

34 The value of one or more of the `Offset`, `Priority_Reduction`, and `Length` attributes
35 of `AD` is invalid.

36 In the case that the operation successfully queues the I/O request, but it subse-
37 quently encounters an error, the error shall be reported via the `Status Code` attribute
38 of the request, as specified in §6.3.6.

6.3.5 List Directed I/O

6.3.5.1 Synopsis

```
1  type AIO_Descriptor_List is
2      array (Positive range <>) of AIO_Descriptor;
3  procedure List_IO_No_Wait
4      (List: in out AIO_Descriptor_List;
5       Event: in POSIX_Signals.Signal_Event);
6  procedure List_IO_Wait
7      (List: in out AIO_Descriptor_List;
8       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 asynchronous I/O 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 which 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 asynchronous I/O 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 occur, 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 any of the following conditions 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 asynchronous I/O 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 Asynchronous I/O 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 asynchronous I/O 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 asynchronous I/O request has the following attributes, which are collectively termed the *status* of the request.

Status Code

This is a value of type `POSIX.Error_Code` that 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 asynchronous I/O 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 asynchronous I/O descriptor is not a valid file descriptor open for writing.

Invalid_Argument

The file offset value implied by the `Offset` attribute of the specified asynchronous I/O 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.

1 **Completed_Successfully**
2 The asynchronous I/O request has completed successfully.
3 **Canceled**
4 The request was successfully queued, but was canceled before the
5 I/O completed, due to an explicit **Cancel** request.

6 **Bytes Transferred**
7 If the requested operation completed successfully or was canceled, it has an
8 associated count of the number of bytes actually transferred by the I/O oper-
9 ation. This value shall be the same as would be returned in the parameter
10 **Last** by the corresponding synchronous operation (the **POSIX_IO.Read** or
11 **POSIX_IO.Write**).

12 NOTE: The status attributes of an asynchronous I/O request defined here are not to be
13 confused with and are not part of the attributes of an asynchronous I/O control block
14 defined in §6.3.2.

15 **Get_AIO_Status** shall raise **POSIX_Error**, with the **Status Code** of the request as
16 error code, if the request has failed and **Get_Bytes_Transferred** has not yet been
17 called for AD. If the request has not failed and **Get_Bytes_Transferred** has not
18 yet been called for AD, the function shall return the value of type **AIO_Status** corre-
19 sponding to the **Status Code** according to Table 6.2:

20 **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

25 **Get_AIO_Error_Code** shall return the **Status Code** of the asynchronous I/O request
26 specified by AD, if **Get_Bytes_Transferred** has not yet been called for AD.

27 The AIO descriptor originally specified in the call that requests initiation of an asyn-
28 chronous I/O operation can be used, thereafter, as a handle for retrieving the status
29 of the I/O request. It shall remain valid for this purpose during its lifetime.

30 NOTE: An application may get status for an AIO descriptor repeatedly, but may not do
31 so after a call to **Get_Bytes_Transferred** for the AIO descriptor.

32 **6.3.6.3 Error Handling**

33 If any of the following conditions occur, the exception **POSIX_Error** shall be raised,
34 with the corresponding error code:

35 **Operation_Not_Implemented**
36 **Get_AIO_Status** and **Get_AIO_Error_Code** are not supported by this
37 implementation.

38 If any of the following conditions is detected, the exception **POSIX_Error** shall be
39 raised, with the corresponding error code:

Invalid_Argument

AD does not refer to an asynchronous I/O request whose Bytes Transferred attribute has not yet been retrieved. That is, either no asynchronous I/O has been requested via AD, or Get_Bytes_Transferred has already been called for AD since the last asynchronous I/O 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 asynchronous I/O 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, POSIX_Error shall be raised with the error code associated with the I/O request.

6.3.7 Retrieve Bytes Transferred by Asynchronous I/O 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 asynchronous I/O 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 asynchronous I/O 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 asynchronous I/O request later, then Get_Bytes_Transferred can again be called to retrieve the status of that request.

NOTE: It is not clear in POSIX.1b {3} whether Get_Bytes_Transferred is required to return the number of bytes transferred in the case 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 occur, the exception POSIX_Error shall be raised, with the corresponding error code:

Invalid_Argument

AD does not refer to an asynchronous I/O request whose Bytes Transferred attribute has not yet been retrieved. That is, either no asynchronous I/O 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 any of the following conditions 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 Asynchronous I/O Request

6.3.8.1 Synopsis

```
type Cancellation_Status is
    (Canceled,
     Not_Canceled,
     All_Done);
function Cancel
    (AD: AIO_Descriptor)
return Cancellation_Status;
function Cancel
    (File: POSIX_IO.File_Descriptor)
return Cancellation_Status;
```

6.3.8.2 Description

`Cancel` attempts to cancel one or more asynchronous I/O 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 asynchronous I/O requests against the file descriptor `File`.

For asynchronous I/O 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 asynchronous I/O request.

For asynchronous I/O 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 asynchronous I/O 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 occur, 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 any of the following conditions 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 Asynchronous I/O 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 asynchronous I/O request referenced by the `AD` argument has completed.
- (2) At least one of the asynchronous I/O 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 asynchronous I/O request via `Read`, `Write`, or `List_IO_No_Wait`. If it is associated with a

completed asynchronous I/O 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 asynchronous I/O 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 asynchronous I/O 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 asynchronous I/O, the effect is undefined.

NOTE: For operations with a `List` parameter, an application may determine which asynchronous 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`.

In the case of 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 occur, the exception `POSIX_Error` shall be raised, with the corresponding error code:

`Resource_Temporarily_Unavailable`

Upon `Await_IO_Or_Timeout`, no specified asynchronous I/O 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 asynchronous I/O 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 any of the following conditions 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 below.

`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).

In the case of 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.) In the case of 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 it is only the I/O that was queued at the time of the call to `Synchronize_File` or `Synchronize_Data` that is 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 asynchronous I/O request.

When the request is queued, the Status Code of the request shall be set to `In_Progress`. When all data has 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 §2.2.2.136).

All other attributes of the asynchronous I/O 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 is not guaranteed to have been successfully transferred.

6.3.10.3 Error Handling

If any of the following conditions occur, the exception `POSIX_Error` shall be raised, with the corresponding error code:

`Resource_Temporarily_Unavailable`

The asynchronous I/O 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.

In the event that any of the queued I/O requests fail, `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 any of the following conditions 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`.

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Section 7: Device- and Class-Specific Functions

This section contains a general terminal interface for communications ports on the system, and 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 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 apply, it is so noted.

7.1.1 Interface Characteristics

7.1.1.1 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.

7.1.1.2 Process Groups

A terminal may have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters, as discussed in §7.1.1.9. 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 §7.1.1.4.)

7.1.1.3 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 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 §7.1.1.10.)

7.1.1.4 Terminal Access Control

If a process is in the foreground process group of its controlling terminal, read operations shall be allowed as described in §7.1.1.5. For those implementations that support 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 special cases apply:

- 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 §7.1.1.8. Attempts by a process in a background process group to write to its controlling terminal shall cause the process group to be sent `POSIX_Signals.Signal_Terminal_Output` unless one of the following special cases apply:

- 1 – If `Send_Signal_For_BG_Output` is disabled, or if `Send_Signal_For_-`
2 `BG_Output` is enabled and the process is ignoring or blocking `Signal_-`
3 `Terminal_Output`, the process shall be allowed to write to the terminal
4 and no signal shall be sent.
- 5 – If `Send_Signal_For_BG_Output` is enabled and the process group of the
6 writing process is orphaned, and the writing process is not ignoring or block-
7 ing `Signal_Terminal_Output`, the write operation shall raise `POSIX_Er-`
8 ror with error code `Input_Output_Error`, and no signal shall be sent.

9 Certain calls that set terminal parameters shall be treated in the same fashion as
10 write operations, except that `Send_Signal_For_BG_Output` shall be ignored; that
11 is, the effect shall be identical to that for terminal write operations when `Send_-`
12 `Signal_For_BG_Output` is enabled. (See §7.2.)

13 7.1.1.5 Input Processing and Reading Data

14 A terminal device associated with a terminal device file may operate in full-duplex
15 mode, so that data may arrive while output is occurring. Each terminal device file
16 shall be associated with an input queue, into which incoming data shall be stored
17 by the system before being read by a process. If `POSIX_Configurable_File_-`
18 `Limits.Input_Queue_Is_Limited` returns `True`, the system shall impose a limit
19 on the number of bytes that may be stored in the input queue. This limit is `Input`
20 `Queue Maximum` (see §5.4). The behavior of the system when this limit is exceeded is
21 implementation defined.

22 Two general kinds of input processing shall be available, canonical and noncanonical.
23 These modes are described in §7.1.1.6 and §7.1.1.7. Additionally, input characters
24 shall be processed according to the `Input_Modes` (see §7.2.2) and `Local_Modes`
25 (see §7.2.5) characteristics. Such processing may include echoing, which in general
26 means transmitting input characters immediately back to the terminal when they
27 are received from the terminal. This is useful for terminals that can operate in full-
28 duplex mode.

29 The manner in which data is provided to a process reading from a terminal device file
30 shall be dependent on whether the terminal device file is in canonical or noncanonical
31 mode.

32 The `Non_Blocking` option for `POSIX_IO.Open` (see §6.1.1 and `POSIX_IO.Set_-`
33 `File_Control` (see §6.1.7) shall also affect the manner in which data is provided
34 to a process. If the option `Non_Blocking` was set to `False`, then any read request
35 on the specified file shall be blocked until data is available or a signal has been
36 received. If the option `Non_Blocking` was set to `True`, then the read request shall
37 be completed, without blocking, in one of three ways:

- 38 (1) If there is enough data available to satisfy the entire request, the read operation
39 shall complete successfully and return the index of the last byte read.
- 40 (2) If there is not enough data available to satisfy the entire request, the read op-
41 eration shall complete successfully, having read as much data as possible, and
42 return the index of the last byte it was able to read.
- 43 (3) If there is no data available, the read operation shall instead return immediately
44 with `Last` set to `Buffer'First - 1`.

When data is available depends on whether the input processing mode is canonical or noncanonical. Subclauses §7.1.1.6 and §7.1.1.7 describe each of these input processing modes.

7.1.1.6 Canonical Mode Input Processing

In canonical mode input processing, terminal input 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 §7.1.1.9 for more information on NL, EOF, and EOL.

This means that 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, the Erase and Kill characters (see §7.1.1.9), is received. This processing shall affect data in the input queue that has 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 there is no data in the current line. The Erase and Kill characters themselves shall not be placed in the input queue.

7.1.1.7 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_Maximum 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 below.

- 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 - 1` 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 values 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.

7.1.1.8 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`.

7.1.1.9 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 §7.1.1.6.) 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 §7.2.7), 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`

1 mode is enabled. The **Stop** character shall be used to temporarily sus-
2 pend output. It is useful with CRT terminals to prevent output from
3 disappearing before it can be read.

4 **Start:** The **Start** special character shall be recognized on both input and output
5 if the **Enable_Start_Stop_Input** or **Enable_Start_Stop_Output**
6 mode is enabled. The **Start** character shall be used to resume output
7 that has been suspended by a **Stop** character. The effect of a **Start** char-
8 acter in the absence of a previous **Stop** character is implementation de-
9 fined.

10 **CR:** The Carriage Return special character shall be recognized on input if
11 the **Canonical_Input** mode flag is enabled. When **Canonical_Input**
12 and **Map_CR_To_LF** are enabled and **Ignore_CR** is disabled, this char-
13 acter shall be translated into an NL and shall have the same effect as
14 an NL character.

15 If a special character (except NL, EOL, and CR) is recognized as described in the
16 preceding paragraphs, then the character shall be removed from the input stream
17 and discarded when it is processed. If the special character is not recognized, then it
18 shall be passed through the input buffer as a normal character.

19 The NL and CR characters cannot be changed. It is implementation defined whether
20 the **Start** and **Stop** characters can be changed. The values for **Interrupt**, **Quit**, **Erase**,
21 **Kill**, **EOF**, **EOL**, and **Suspend** (only if the Job Control option is supported) shall be
22 changeable to suit individual tastes using the procedure **Define_Special_Con-**
23 **trol_Character**. (See §7.2.7.)

24 If two or more special characters have the same value, the function performed when
25 that character is received is undefined.

26 A special character shall be recognized not only by its value, but also by its context;
27 for example, an implementation may define a sequence of multiple **POSIX_Charac-**
28 **ter** values that have a meaning different from the meaning of the **POSIX_Char-**
29 **acter** value when considered individually. Implementations may also define addi-
30 tional single **POSIX_Character** functions. These implementation defined sequences
31 of multiple **POSIX_Character** values or single **POSIX_Character** functions shall
32 be recognized only if the **Extended_Functions** mode is enabled; otherwise, data
33 shall be interpreted as normal characters or as the special characters defined in this
34 section.

35 **7.1.1.10 Modem Disconnect**

36 If a modem disconnect is detected by the terminal interface for a controlling termi-
37 nal, and if **Ignore_Modem_Status** mode is disabled for the terminal (see §7.2.4),
38 **POSIX_Signals.Signal_Hangup** shall be sent to the controlling process associated
39 with the terminal. Unless other arrangements have been made, this shall cause the
40 controlling process to terminate. Any subsequent read from the terminal device shall
41 return with an EOF indication until the device is closed. Thus, processes that read
42 a terminal file and test for EOF can terminate appropriately after a disconnect. Any
43 subsequent write to the terminal device shall raise **POSIX_Error** with error code
44 **Input_Output_Error**, until the device is closed.

7.1.1.11 Closing a Terminal Device File

The last process to close a terminal device file shall cause any output to be sent to the device and any input to be discarded. Then, if `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,

```

```

1      Extended_Functions, Enable_Signals, No_Flush,
2      Send_Signal_For_BG_Output);
3  -- 7.2.2 Input Modes
4  subtype Input_Modes is Terminal_Modes
5      range Interrupt_On_Break .. Mark_Parity_Errors;
6  -- 7.2.3 Output Modes
7  subtype Output_Modes is Terminal_Modes
8      range Perform_Output_Processing .. Perform_Output_Processing;
9  -- 7.2.4 Control Modes
10 subtype Control_Modes is Terminal_Modes
11     range Ignore_Modem_Status .. Odd_Parity;
12 -- 7.2.5 Local Modes
13 subtype Local_Modes is Terminal_Modes
14     range Echo .. Send_Signal_For_BG_Output;
15 -- 7.2.6 Retrieve and Define Terminal Modes and Bits per Character
16 type Terminal_Modes_Set is array (Terminal_Modes) of Boolean;
17 subtype Bits_Per_Character is Positive range implementation-defined;
18 function Terminal_Modes_Of
19     (Characteristics: Terminal_Characteristics)
20     return Terminal_Modes_Set;
21 procedure Define_Terminal_Modes
22     (Characteristics: in out Terminal_Characteristics;
23      Modes:          in      Terminal_Modes_Set);
24 function Bits_Per_Character_Of
25     (Characteristics: Terminal_Characteristics)
26     return Bits_Per_Character;
27 procedure Define_Bits_Per_Character
28     (Characteristics: in out Terminal_Characteristics;
29      Bits:           in      Bits_Per_Character);
30 -- 7.2.9 Baud Rate Subprograms
31 type Baud_Rate is
32     (B0, B50, B75, B110, B134, B150, B200, B300, B600,
33      B1200, B1800, B2400, B4800, B9600, B19200, B38400);
34 function Input_Baud_Rate_Of
35     (Characteristics: Terminal_Characteristics)
36     return Baud_Rate;
37 procedure Define_Input_Baud_Rate
38     (Characteristics: in out Terminal_Characteristics;
39      Input_Baud_Rate: in      Baud_Rate);
40 function Output_Baud_Rate_Of
41     (Characteristics: Terminal_Characteristics)
42     return Baud_Rate;
43 procedure Define_Output_Baud_Rate
44     (Characteristics: in out Terminal_Characteristics;
45      Output_Baud_Rate: in      Baud_Rate);
46 -- 7.2.7 Special Control Characters
47 type Control_Character_Selector is
48     (EOF_Char, EOL_Char, Erase_Char, Interrupt_Char,
49      Kill_Char, Quit_Char, Suspend_Char, Start_Char, Stop_Char);
50 function Special_Control_Character_Of
51     (Characteristics: Terminal_Characteristics;
52      Selector: Control_Character_Selector)
53     return POSIX.POSIX_Character;
54 procedure Define_Special_Control_Character
55     (Characteristics: in out Terminal_Characteristics;
56      Selector:       in      Control_Character_Selector;
57      Char:           in      POSIX.POSIX_Character);

```

```

1  procedure Disable_Control_Character
2      (Characteristics: in out Terminal_Characteristics;
3       Selector:      in      Control_Character_Selector);
4      -- 7.2.8 Noncanonical Controls
5  function Input_Time_Of
6      (Characteristics: Terminal_Characteristics)
7      return Duration;
8  procedure Define_Input_Time
9      (Characteristics: in out Terminal_Characteristics;
10     Input_Time:      in      Duration);
11 function Minimum_Input_Count_Of
12     (Characteristics: Terminal_Characteristics)
13     return Natural;
14 procedure Define_Minimum_Input_Count
15     (Characteristics:      in out Terminal_Characteristics;
16     Minimum_Input_Count: in      Natural);
17 -- 7.2.10 Line Control Operations
18 procedure Send_Break
19     (File:      in POSIX_IO.File_Descriptor;
20     The_Duration: in Duration := 0.0);
21 procedure Drain
22     (File:      in POSIX_IO.File_Descriptor;
23     Masked_Signals: in POSIX.Signal_Masking := POSIX.RTS_Signals);
24 type Queue_Selector is
25     (Received_But_Not_Read, Written_But_Not_Transmitted, Both);
26 procedure Discard_Data
27     (File:      in POSIX_IO.File_Descriptor;
28     Selector: in Queue_Selector);
29 type Flow_Action is
30     (Suspend_Output, Restart_Output, Transmit_Stop, Transmit_Start);
31 procedure Flow
32     (File:      in POSIX_IO.File_Descriptor;
33     Action: in Flow_Action);
34 -- 7.2.11 Foreground Process Group ID
35 function Get_Process_Group_ID
36     (File: POSIX_IO.File_Descriptor)
37     return POSIX_Process_Identification.Process_Group_ID;
38 procedure Set_Process_Group_ID
39     (File:      in POSIX_IO.File_Descriptor;
40     Group_ID: in POSIX_Process_Identification.Process_Group_ID);
41 -- 7.2.12 Generate Terminal Pathname
42 function Get_Controlling_Terminal_Name return POSIX.Pathname;
43 private
44     implementation-defined
45 end POSIX_Terminal_Functions;

```

46 7.2.1 Terminal Characteristics

47 7.2.1.1 Synopsis

```

48 type Terminal_Characteristics is private;
49 Invalid_Terminal_Characteristics: constant Terminal_Characteristics;
50 function Get_Terminal_Characteristics
51     (File : POSIX_IO.File_Descriptor)
52     return Terminal_Characteristics;

```

```

1  type Terminal_Action_Times is
2      (Immediately, After_Output, After_Output_And_Input);
3  procedure Set_Terminal_Characteristics
4      (File:          in POSIX_IO.File_Descriptor;
5       Characteristics: in Terminal_Characteristics;
6       Apply:         in Terminal_Action_Times := Immediately;
7       Masked_Signals: in POSIX.Signal_Masking := POSIX.RTS_Signals);
8  type Terminal_Modes is
9      -- Subtype Input Modes:
10     (Interrupt_On_Break, Map_CR_To_LF, Ignore_Break,
11      Ignore_CR, Ignore_Parity_Errors, Map_LF_To_CR,
12      Enable_Parity_Check, Strip_Character, Enable_Start_Stop_Input,
13      Enable_Start_Stop_Output, Mark_Parity_Errors,
14      -- Subtype Output Modes:
15      Perform_Output_Processing,
16      -- Subtype Control Modes:
17      Ignore_Modem_Status, Enable_Receiver, Send_Two_Stop_Bits,
18      Hang_Up_On_Last_Close, Parity_Enable, Odd_Parity,
19      -- Subtype Local Modes:
20      Echo, Echo_Erase, Echo_Kill, Echo_LF, Canonical_Input,
21      Extended_Functions, Enable_Signals, No_Flush,
22      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 has 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
<code>Terminal_Modes_Set</code>	Subdivided into Input modes, Output modes, Control modes, and Local modes. The modes control the functions of the terminal.
<code>Input_Baud_Rate</code>	Terminal input data rate.
<code>Output_Baud_Rate</code>	Terminal output data rate.
<code>Control_Characters</code>	Array of terminal interface Control Characters.
<code>Input_Time</code>	Time value for timer used in noncanonical input.
<code>Minimum_Input_Count</code>	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.

- 1 — **Immediately** shall instruct the system to change the values immediately.
- 2 — **After_Output** shall instruct the system to wait until all output written to the
- 3 parameter **File** has been transmitted. This mode should be used when the
- 4 change affects output.
- 5 — **After_Output_And_Input** shall instruct the system to wait until all output
- 6 has been transmitted. All input received and not read shall be discarded before
- 7 the change is made.

8 The current state of the terminal characteristics shall be obtained via the function
9 **Get_Terminal_Characteristics**. Components of the terminal characteristics
10 shall be modified by subprograms defined in the following subclauses. Modifica-
11 tions to the terminal characteristics then shall be applied to the terminal via the
12 **Set_Terminal_Characteristics** procedure. The parameter **Masked_Signals** in
13 **Set_Terminal_Characteristics** specifies which signals shall be masked during
14 the operation. (See §2.4.1.6.)

15 The zero baud rate **B0** shall be used to terminate the connection. If **B0** is specified as
16 the output baud rate in the parameter **Characteristics** when **Set_Terminal_-**
17 **Characteristics** is called, the modem control lines no longer shall be asserted.
18 Normally, this will disconnect the line.

19 If the input baud rate is equal to **B0** in the parameter **Characteristics** when
20 **Set_Terminal_Characteristics** is called, the input baud rate will be changed by
21 **Set_Terminal_Characteristics** to the same value as that specified by the value
22 of the output baud rate, exactly as if the input rate had been set to the output rate
23 by a call to **Define_Input_Baud_Rate**. This usage of **B0** is obsolescent.

24 **Set_Terminal_Characteristics** shall return successfully if it was able to per-
25 form any of the requested actions, even if some of the requested actions could not be
26 performed. It shall set all the attributes that the implementation does support as
27 requested and leave all the attributes not supported by the hardware unchanged. If
28 no part of the request can be honored, it shall raise **POSIX_Error** with error code
29 **Invalid_Argument**. If the input and output baud rates differ and are a combina-
30 tion that is not supported, neither baud rate shall be changed. A subsequent call
31 to **Get_Terminal_Characteristics** shall return the actual state of the terminal
32 device (reflecting both the changes made and not made in the previous call to **Set_-**
33 **Terminal_Characteristics**).

34 No actions defined by this standard, other than a call to **Set_Terminal_Charac-**
35 **teristics** or a call to **POSIX_IO.Close** on the last file descriptor in the system
36 associated with this terminal device, shall cause any of the terminal attributes de-
37 fined by this standard to change.

38 The type **Terminal_Modes** is divided into subranges **Input_Modes**, **Output_Modes**,
39 **Control_Modes**, and **Local_Modes**.

40 **7.2.1.3 Error Handling**

41 If any of the following conditions occur, the exception **POSIX_Error** shall be raised,
42 with the corresponding error code:

- 1 **Bad_File_Descriptor**
2 The parameter `File` is not a valid `POSIX_IO.File_Descriptor`.
3 **Inappropriate_IO_Control_Operation**
4 The file associated with the parameter `File` is not a terminal.
5 **Interrupted_Operation**
6 A signal interrupted the `Set_Terminal_Characteristics` procedure.
7 **Invalid_Argument**
8 Either the value of `Characteristics` was not initialized via a call to `Get_`
9 `Terminal_Characteristics`, or none of the terminal characteristics re-
10 quested by `Characteristics` is supported for the terminal device associ-
11 ated with the parameter `File`.

12 7.2.2 Input Modes

13 7.2.2.1 Synopsis

14 **subtype** `Input_Modes` **is** `Terminal_Modes`
15 **range** `Interrupt_On_Break` .. `Mark_Parity_Errors`;

16 7.2.2.2 Description

17 Values of the subtype `Input_Modes`, shown in Table 7.2, shall describe the basic
18 terminal input control.

19 **Table 7.2 – Terminal_Modes Values for Input Control**

Value	Meaning
<code>Enable_Parity_Check</code>	Enable input parity check
<code>Enable_Start_Stop_Input</code>	Enable start/stop input control
<code>Enable_Start_Stop_Output</code>	Enable start/stop output control
<code>Ignore_Break</code>	Ignore break condition
<code>Ignore_CR</code>	Ignore CR
<code>Ignore_Parity_Errors</code>	Ignore characters with parity errors
<code>Interrupt_On_Break</code>	Signal interrupt on break
<code>Map_CR_To_LF</code>	Map carriage return (CR) to NL on input
<code>Map_LF_To_CR</code>	Map NL to CR on input
<code>Mark_Parity_Errors</code>	Mark parity errors
<code>Strip_Character</code>	Strip character

32 In the context of asynchronous serial data transmission, a break condition is defined
33 as a sequence of zero-valued bits that continues for more than the time to send one
34 byte. The entire sequence of zero-valued bits shall be interpreted as a single break
35 condition, even if it continues for a time equivalent to more than one byte. In contexts
36 other than asynchronous serial data transmission, the definition of a break condition
37 is implementation defined.

38 If `Ignore_Break` mode is enabled, a break condition detected on input shall be ig-
39 nored, that is, not put on the input queue and therefore not read by any process. If
40 `Ignore_Break` mode is disabled and `Interrupt_On_Break` is enabled, the break

condition shall flush the input and output queues, and if the terminal is the controlling terminal of a foreground process group, the break condition shall generate a single `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 or not.

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
<code>Perform_Output_Processing</code>	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.

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.

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

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.

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_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 this was the case or do nothing.

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. This enables the erase and kill edit functions and the assembly of input characters into lines delimited by NL, EOF, and EOL, as described in §7.1.1.6. If 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_Characters have been received or the timeout value Input_Time has expired between POSIX_Characters. (See §7.1.1.7.)

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 `Send_Signal_For_BG_Output` mode is enabled and the implementation supports the Job Control option, `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 any of the following conditions occur, 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 Special Control Characters

7.2.7.1 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);
```

7.2.7.2 Description

The special control character name and description in both canonical and noncanonical modes are shown in Table 7.6.

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.

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

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; that is, no input data shall be recognized as the disabled special character.

The initial values of all control characters are implementation defined.

7.2.7.3 Error Handling

If any of the following conditions occur, 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.

7.2.8 Noncanonical Controls

7.2.8.1 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);
```

7.2.8.2 Description

Define_Input_Time shall set the input time value used in noncanonical mode. (See §7.1.1.7.) 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 §7.1.1.7.) Minimum_Input_Count_Of shall retrieve the input count value from the control structure.

7.2.8.3 Error Handling

If any of the following conditions occur, 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.

7.2.9 Baud Rate Subprograms

7.2.9.1 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);
```


7.2.9.2 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.

7.2.9.3 Error Handling

If any of the following conditions occur, 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.

7.2.10 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.

1 The `Drain` procedure shall wait until all output written to the object referred to by
2 the parameter `File` has been transmitted.

3 The `Discard_Data` procedure shall discard data written to the object referred to by
4 the parameter `File`, based on the value of the parameter `Selector`:

5 — If `Written_But_Not_Transmitted`, data written to the object designated by
6 the parameter `File` but not transmitted shall be discarded.

7 — If `Received_But_Not_Read`, data received from the object designated by the
8 parameter `File` but not read shall be discarded.

9 — If `Both`, data both transmitted to and received from the object designated by the
10 parameter `File` shall be discarded.

11 The `Flow` procedure shall suspend transmission or reception of data on the object
12 referred to by the parameter `File`, depending on the value of parameter `Action`:

13 — If `Suspend_Output`, output shall be suspended. That is, output characters will
14 not be sent to the output object when requested. They shall be held in an internal
15 buffer and/or the write will block.

16 — If `Restart_Output`, suspended output will be restarted.

17 — If `Transmit_Stop`, the system shall transmit a `Stop` character, which is in-
18 tended to cause the terminal device to stop transmitting data to the system.

19 — If `Transmit_Start`, the system shall transmit a `Start` character, which is in-
20 tended to cause the terminal device to start transmitting data to the system.

21 The default upon open of a terminal file is that neither its input nor its output is
22 suspended.

23 The parameter `Masked_Signals` in the operation `Drain` specifies which signals
24 shall be masked during the operation. (See §2.4.1.6.)

25 **7.2.10.3 Error Handling**

26 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
27 with the corresponding error code:

28 **Bad_File_Descriptor**

29 Upon `Send_Break`, `Drain`, `Discard_Data`, or `Flow`, the parameter `File` is
30 not a valid `POSIX_IO.File_Descriptor`.

31 **Inappropriate_IO_Control_Operation**

32 Upon `Send_Break`, `Drain`, `Discard_Data`, or `Flow`, the object designated
33 by the parameter `File` is not a terminal.

34 **Interrupted_Operation**

35 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 for this implementation, `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 raise `POSIX_Error` with error code `Operation_Not_Implemented`.

If the Job Control option is supported and 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 occur, 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 `Group_ID`.

1 **7.2.12 Generate Terminal Pathname**

2 **7.2.12.1 Synopsis**

3 **function** Get_Controlling_Terminal_Name **return** POSIX.Pathname;

4 **7.2.12.2 Description**

5 The function Get_Controlling_Terminal_Name shall return a value of type PO-
6 SIX.Pathname that shall refer to the current controlling terminal of the current
7 process. Get_Controlling_Terminal_Name shall return an empty Pathname if
8 the pathname that would refer to the controlling terminal cannot be determined or
9 if the function is unsuccessful.

10 Access to the file is not guaranteed, even if Get_Controlling_Terminal_Name
11 returns a valid pathname.

12 **7.2.12.3 Error Handling**

13 No exceptions are specified by this standard for Get_Controlling_Terminal_Name.

14 Get_Controlling_Terminal_Name may raise POSIX_Error for implementation
15 defined conditions.

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

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-conformant Ada compiler/library/runtime 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 those 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 §7.1.1.7.)

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 parameter 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 paragraphs 7 and 8 of Annex A.10 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 input/output 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 Ada RM {1} to be implementation dependent.

Ada RM {1} A.7, paragraph 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 charactermapping discussed in §2.4.2. The Ada Form parameter shall be interpreted as described elsewhere in this section.

Ada RM {1} A.7, paragraph 2:

The term “external file” shall be interpreted as referring to a POSIX file (defined in §2.2.2.43).

Ada RM {1} A.7, paragraph 6:

Upon the termination (as defined by the Ada RM {1}) 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.

Ada RM {1} A.8.2, paragraph 22:

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.

Ada RM {1} A.10, paragraph 2:

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.

Ada RM {1} A.10, paragraph 6:

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.

1 Ada RM {1} A.13, paragraph 13:

2 The phrases “input character sequence” and “value input” shall be interpreted as referring to the characters and values resulting from applying the
3 character mapping described in §2.4.2 to the data appearing in the external
4 file.
5

6 **8.1.1.5 Error Handling**

7 The implementation shall raise all exceptions required by Annex A of the Ada RM-
8 {1}. In addition, the implementation shall raise the exceptions listed in the following
9 paragraphs.

10 IO_Exceptions.Use_Error shall be raised under the following circumstances:

- 11 — Text_IO.Open or Text_IO.Create This operation is called when the Form
12 parameter contains a field name or field value that is not defined by the
13 implementation or by the standard.
- 14 — Text_IO.Open or Text_IO.Create This operation is called when a partic-
15 ular field name appears more than once in the Form parameter.
- 16 — Text_IO.Open or Text_IO.Create This operation is called when an un-
17 supported value for Blocking appears in the Form parameter.
- 18 — Text_IO.Open This operation is called when the Form parameter contains
19 an Owner field, a Group field, an Other field, or a FIFO field.
- 20 — Text_IO.Open This operation is called when the value given for the File_-
21 Descriptor field does not correspond to an open file descriptor or if the
22 modes are incompatible.
- 23 — Text_IO.Create This operation is called when the Form parameter con-
24 tains an Append field.
- 25 — Text_IO.Create This operation is called when the Form parameter con-
26 tains a File_Descriptor field.
- 27 — Text_IO.Name This operation is attempted upon an Ada file object that
28 was opened with the File_Descriptor field specified within the Form pa-
29 rameter, or the operation is attempted upon a file that is redirected to a
30 pipe or other facility so that a pathname or terminal device name cannot be
31 determined.
- 32 — Text_IO.New_Page This operation is performed on an output file opened
33 with “Page_Terminators => False.”
- 34 — Text_IO.Set_Line This operation is performed on an output file opened
35 with “Page_Terminators => False,” and the value specified by the To
36 parameter is less than the current line number.

37 **8.1.2 Additional Interpretation of the POSIX.1 Standard**

38 Execution of the Text_IO.Open and Text_IO.Create operations shall never result
39 in the Ada file object becoming associated with the controlling terminal of the POSIX
40 process associated with the active partition. The application may associate a file
41 descriptor for a controlling terminal with an object of type Text_IO.File_Type
42 by using the File_Descriptor field of the Text_IO.Open Form parameter. The
43 functionality (see §7.1.1.3) 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

```

3  type File_Structure_Values is (Regular, FIFO);
4  type Terminal_Input_Values is (Lines, Characters);
5  type Possible_File_Descriptor (Valid: Boolean := False) is
6      record
7          case Valid is
8              when True =>
9                  Descriptor: POSIX_IO.File_Descriptor;
10             when False => null;
11          end case;
12      end record;
13  type Form_Values_for_Open is
14      record
15          Append: Boolean := False;
16          Blocking: POSIX.Text_IO_Blocking_Behavior
17              := POSIX.IO_Blocking_Behavior;
18          Terminal_Input: Terminal_Input_Values := Lines;
19          Page_Terminators: Boolean := True;
20          File_Descriptor: Possible_File_Descriptor;
21      end record;
22  type Form_Values_for_Create is
23      record
24          Permission_Mask: POSIX_Permissions.Permission_Set
25              := POSIX_Permissions.Access_Permission_Set;
26          Blocking: POSIX.Text_IO_Blocking_Behavior
27              := POSIX.IO_Blocking_Behavior;
28          Terminal_Input: Terminal_Input_Values := Lines;
29          File_Structure: File_Structure_Values := Regular;
30          Page_Terminators: Boolean := True;
31      end record;
32  function Form_String (Val: Form_Values_for_Open) return String;
33  function Form_Value (Str: String) return Form_Values_for_Open;
34  function Form_String (Val: Form_Values_for_Create) return String;
35  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 {2} 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;
```



```

1      function User_Name_Of (DB_Item: User_Database_Item)
2      return POSIX.POSIX_String;
3      function User_ID_Of (DB_Item: User_Database_Item)
4      return POSIX_Process_Identification.User_ID;
5      function Group_ID_Of (DB_Item: User_Database_Item)
6      return POSIX_Process_Identification.Group_ID;
7      function Initial_Directory_Of (DB_Item: User_Database_Item)
8      return POSIX.POSIX_String;
9      function Initial_Program_Of (DB_Item: User_Database_Item)
10     return POSIX.POSIX_String;

```

11 9.1.1.2 Description

12 The User Database contains the name of the user, the user ID, the group ID, and
 13 pathnames for initial default directory and default program.

14 User_Name_Of shall return the textual name of the user associated with the param-
 15 eter DB_Item.

16 User_ID_Of shall return the user ID associated with the value of the parameter
 17 DB_Item.

18 Group_ID_Of shall return the primary group membership of the user that is associ-
 19 ated with the value of the parameter DB_Item.

20 Initial_Directory_Of shall return the pathname to the home directory of the
 21 user that is associated with the value the parameter DB_Item. If this operation
 22 returns a null POSIX.POSIX_String, the meaning is implementation defined.

23 Initial_Program_Of shall return the pathname of the initial program of the user
 24 that is associated with the value of the parameter DB_Item. If the operation returns
 25 a null POSIX.POSIX_String, this indicates that the system-default user program
 26 shall be used.

27 9.1.1.3 Error Handling

28 If any of the following conditions occur, the exception POSIX_Error shall be raised,
 29 with the corresponding error code:

30 Invalid Argument

31 The DB_Item parameter is not valid. The caller has to obtain a User Data-
 32 base item from one of the access operations described in §9.1.2 before calling
 33 any of these operations.

34 9.1.2 Access User Database Items

35 9.1.2.1 Synopsis

```

36     function Get_User_Database_Item
37     (UID: POSIX_Process_Identification.User_ID)
38     return User_Database_Item;
39     function Get_User_Database_Item
40     (Name: POSIX.POSIX_String)
41     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 any of the following conditions occur, 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;
```

```

1  private
2      implementation-defined
3  end POSIX_Group_Database;
```

4 9.2.1 Access Contents of a Group Database Item

5 9.2.1.1 Synopsis

```

6      type Group_Database_Item is private;
7      type Group_ID_List is private;
8      function Group_Name_Of (DB_Item: Group_Database_Item)
9          return POSIX.POSIX_String;
10     function Group_ID_Of (DB_Item: Group_Database_Item)
11         return POSIX_Process_Identification.Group_ID;
12     function Group_ID_List_Of (DB_Item: Group_Database_Item)
13         return Group_ID_List;
```

14 9.2.1.2 Description

15 The Group Database contains, for each group, the group name, the group ID, and a
16 list of group members. The type `Group_ID_List` is a private type with an operation
17 to iterate over the members of the given group, as described in §9.2.3.

18 `Group_Name_Of` shall return the group name associated with the value of parameter
19 `DB_Item`

20 `Group_ID_Of` shall return the group ID associated with the value of parameter `DB_`
21 `Item`.

22 `Group_ID_List_Of` shall return the list of group members associated with the value
23 of parameter `DB_Item`.

24 9.2.1.3 Error Handling

25 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
26 with the corresponding error code:

27 `Invalid_Argument`

28 The `DB_Item` parameter is not valid. The caller has to obtain a Group Data-
29 base item from one of the access operations described in §9.2.3 before calling
30 any of these operations.

31 9.2.2 Access Elements of the Group Item List of Members

32 9.2.2.1 Synopsis

```

33     generic
34         with procedure Action
35             (ID: in POSIX.POSIX_String;
36              Quit: in out Boolean);
37     procedure For_Every_Member (List: in Group_ID_List);
38     function Length (Member_List: Group_ID_List)
39         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 0 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 0), then the actual procedure associated with the formal parameter `Action` shall not be called.

9.2.2.3 Error Handling

If any of the following conditions occur, 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 any of the following conditions occur, 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.

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Section 10: Data Interchange Format

- 1 *This section position is reserved for a future Ada binding to the section on Data Inter-*
- 2 *change Format of POSIX.1 {2}.*

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Section 11: Synchronization

1 This section describes the services that this standard provides for process and task
2 synchronization via counting semaphores, mutexes, and condition variables.

3 11.1 Package POSIX_Semaphores

4 This package provides access to services related to counting semaphores. Counting
5 semaphores are a type of synchronization object that can be used for interprocess
6 and intertask synchronization.

7 NOTE: Semaphores are provided in this standard primarily for interprocess communi-
8 cation. The preferred mechanisms for intertask synchronization are those provided by
9 the Ada language, namely, protected objects and task entries.

10 *The functionality described in this clause is optional. If the Semaphores option is not*
11 *supported, the implementation may cause all calls to the explicitly declared operations*
12 *defined in this clause to raise POSIX_Error. Otherwise, the behavior shall be as*
13 *specified below.*

```
14 with POSIX,  
15     POSIX_IO,  
16     POSIX_Permissions;  
17 package POSIX_Semaphores is  
18     -- 11.1.1 Semaphore and Semaphore Descriptor Types  
19     type Semaphore is limited private;  
20     type Semaphore_Descriptor is private;  
21     -- 11.1.2 Initialize an Unnamed Semaphore  
22     procedure Initialize  
23         (Sem:          in out Semaphore;  
24          Value:        in Natural;  
25          Is_Shared:    in Boolean:= False);  
26     function Descriptor_of  
27         (Sem:          Semaphore)  
28         return Semaphore_Descriptor;  
29     -- 11.1.3 Finalize an Unnamed Semaphore  
30     procedure Finalize (Sem: in out Semaphore);  
31     -- 11.1.4 Create/Open a Named Semaphore  
32     function Open  
33         (Name:          POSIX.POSIX_String;  
34          Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)  
35         return Semaphore_Descriptor;  
36     function Open_or_Create  
37         (Name:          POSIX.POSIX_String;  
38          Permissions:   POSIX.Permissions.Permission_Set;  
39          Value:          Natural;  
40          Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;  
41          Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)  
42         return Semaphore_Descriptor;  
43     -- 11.1.5 Close a Named Semaphore  
44     procedure Close (Sem: in out Semaphore_Descriptor);  
45     -- 11.1.6 Remove a Named Semaphore  
46     procedure Unlink_Semaphore (Name: in POSIX.POSIX_String);
```

```

1      -- 11.1.7 Decrement a Semaphore
2      procedure Wait
3          (Sem:          in Semaphore_Descriptor;
4           Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
5      function Try_Wait
6          (Sem: Semaphore_Descriptor)
7          return Boolean;
8      -- 11.1.8 Increment a Semaphore
9      procedure Post (Sem: in Semaphore_Descriptor);
10     -- 11.1.9 Get the Value of a Semaphore
11     function Get_Value (Sem: Semaphore_Descriptor)
12         return Integer;
13 private
14     implementation-defined
15 end POSIX_Semaphores;

```

16 11.1.1 Semaphore and Semaphore Descriptor Types

17 11.1.1.1 Synopsis

```

18     type Semaphore is limited private;
19     type Semaphore_Descriptor is private;

```

20 11.1.1.2 Description

21 The type `Semaphore` is used to represent semaphore objects. Semaphores may, but
 22 need not, be implemented within the file system, and open semaphores may use file
 23 descriptors. Thus, the use of semaphores may reduce the number of file descriptors
 24 available for other uses (see Open Files Maximum).

25 Operating on copies of `Semaphore` objects (which may be created by returning func-
 26 tion results of this type or by using the type as an **in out** parameter) results in
 27 undefined behavior.

28 This standard defines two kinds of semaphores: unnamed semaphores (see §11.1.2)
 29 and named semaphores (see §11.1.4).

30 Objects of type `Semaphore_Descriptor` are used as semaphore descriptors, that
 31 is, handles for semaphore objects. A semaphore descriptor is *valid* if it is associated
 32 with an open named semaphore or an initialized unnamed semaphore.

33 Copies of `Semaphore_Descriptor` objects shall always specify the same semaphore
 34 object. Operating on an object of type `Semaphore_Descriptor`, after the corre-
 35 sponding semaphore object no longer exists, results in undefined behavior which
 36 might affect other processes sharing the same semaphore.

37 This standard specifies two attributes of semaphore objects:

38 Value

39 The Value attribute of a semaphore is an integer representing the state of the
 40 semaphore. This attribute is initialized when the semaphore is initialized
 41 (unnamed semaphore) or created (named semaphore). Three operations:
 42 `Wait`, `Try_Wait`, and `Post` may be applied to semaphores and operate on
 43 the Value attribute. (See §11.1.7 and §11.1.8.)

1 Process Shared

2 The Process Shared attribute of a semaphore is a Boolean that indicates
3 whether the semaphore is sharable between tasks in different processes.
4 A task in any process that can access a sharable semaphore can safely oper-
5 ate upon it using the operations defined in this section. For unnamed sem-
6 aphores the Process Shared attribute is initialized by the parameter `IS_`-
7 Shared (see §11.1.2). For named semaphores the Process Shared attribute
8 is always set to `True` when the semaphore is created (see §11.1.4), that is,
9 named semaphores are always sharable.

10 11.1.2 Initialize an Unnamed Semaphore

11 11.1.2.1 Synopsis

```
12 procedure Initialize
13   (Sem:      in out Semaphore;
14    Value:    in Natural;
15    Is_Shared: in Boolean:= False);
16 function Descriptor_Of
17   (Sem:      Semaphore)
18   return Semaphore_Descriptor;
```

19 11.1.2.2 Description

20 Initialize shall initialize the unnamed semaphore `Sem`, setting the `Value` attribute
21 to `Value` and the Process Shared attribute to `IS_Shared`. Following a normal return
22 from Initialize the semaphore is *initialized*, meaning it can be used in subsequent
23 calls to the operations defined in this section. This semaphore remains initialized un-
24 til it is finalized. If Initialize is called on a semaphore that is already initialized
25 (without an intervening `Finalize`), the effect is undefined.

26 NOTE: In order to provide the correct semantics, implementations must ensure that the
27 parameter `Sem` of Initialize is passed by reference.

28 If `Is_Shared` is `False`, then `Sem` is shared only among tasks of the same process;
29 any task in this process can use `Sem` for performing the operations defined in this
30 section. The use of `Sem` by tasks other than those created in the same process is
31 undefined.

32 NOTE: Several processes may access a sharable unnamed semaphore, if it is allocated
33 in shared memory.

34 `Descriptor_Of` shall return a descriptor for the semaphore object `Sem` that can later
35 be used in semaphore operations. The semaphore corresponding to `Sem` is required to
36 already be initialized, otherwise the effect is undefined. If `Descriptor_Of` is called
37 again, by the same task or by another task within the same process, and `Sem` has not
38 since been finalized, the same descriptor value shall be returned.

39 11.1.2.3 Error Handling

40 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
41 with the corresponding error code:

1 **Invalid_Argument**
2 The value parameter exceeds Semaphores Value Maximum.

3 **No_Space_Left_On_Device**
4 A resource required to initialize the semaphore has been exhausted.
5 The limit on semaphores (Semaphores Maximum) has been reached.

6 **Operation_Not_Implemented**
7 Initialize is not supported by this implementation.

8 **Operation_Not_Permitted**
9 The process lacks the appropriate privilege to initialize the semaphore.

10 **11.1.3 Finalize an Unnamed Semaphore**

11 **11.1.3.1 Synopsis**

12 **procedure** Finalize (Sem: **in out** Semaphore);

13 **11.1.3.2 Description**

14 Finalize is used to finalize the unnamed semaphore Sem. The effect of subsequent
15 use of Sem is undefined until the semaphore is re-initialized by another call to Ini-
16 tialize.

17 NOTE: In order to provide the correct semantics, implementations must ensure that the
18 parameter Sem of Finalize is passed by reference.

19 The effect of finalizing a semaphore on which other tasks are currently blocked is
20 undefined.

21 **11.1.3.3 Error Handling**

22 If any of the following conditions occur, the exception POSIX_Error shall be raised,
23 with the corresponding error code:

24 **Invalid_Argument**
25 Sem is not a properly initialized semaphore object.

26 **Operation_Not_Implemented**
27 Finalize is not supported by this implementation.

28 If any of the following conditions is detected, the exception POSIX_Error shall be
29 raised, with the corresponding error code:

30 **Resource_Busy**
31 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 filesystem 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 one of the following values from `POSIX_IO.Open_Option_Set`:

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: Semaphore creation is not required to be atomic with respect to tasks. This is derived from POSIX.1b {3} and POSIX.1c {4}.

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 semaphore's user ID shall be set to the effective user ID of the process; the semaphore's group ID 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 occur, 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.

- 1 **Interrupted_Operation**
2 Open or Open_Or_Create were interrupted by a signal.
- 3 **Invalid_Argument**
4 Open and Open_Or_Create are not supported for Name. The implementa-
5 tion shall document under what circumstances this error may be returned.
6 Value is greater than Semaphores Value Maximum.
- 7 **Too_Many_Open_Files**
8 Too many semaphore descriptors or file descriptors are currently in use by
9 this process.
- 10 **Too_Many_Open_Files_In_System**
11 Too many semaphores are currently open in the system.
- 12 **Filename_Too_Long**
13 The length in POSIX characters of the specified pathname exceeds Pathname
14 Limit or the length in POSIX characters of a component of the specified path-
15 name is greater than Filename Maximum and the Filename Truncation option is
16 not supported for the pathname prefix of that component.
- 17 **No_Such_File_Or_Directory**
18 Upon a call to Open, the named semaphore does not exist.
- 19 **No_Space_Left_On_Device**
20 There is insufficient space for the creation of the new named semaphore.
- 21 **Operation_Not_Implemented**
22 Open and Open_Or_Create are not supported by this implementation.

23 **11.1.5 Close a Named Semaphore**

24 **11.1.5.1 Synopsis**

25 **procedure** Close (Sem: **in out** Semaphore_Descriptor);

26 **11.1.5.2 Description**

27 The procedure Close shall only be called for named semaphores and is used to indi-
28 cate that all tasks within the calling process are finished using the named semaphore
29 specified by Sem.

30 Close shall deallocate (that is, make available for reuse by a subsequent Open or
31 Open_Or_Create by this process) any system resources allocated by the system
32 for use by this process for this semaphore. The effect of subsequent use of Sem is
33 undefined. If the semaphore has not been removed by a call to Unlink_Semaphore,
34 then Close shall have no effect on the state of the semaphore.

1 **11.1.5.3 Error Handling**

2 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
3 with the corresponding error code:

4 `Invalid_Argument`

5 `Sem` is not a valid semaphore descriptor.

6 `Operation_Not_Implemented`

7 `Close` is not supported by this implementation.

8 **11.1.6 Remove a Named Semaphore**

9 **11.1.6.1 Synopsis**

10 **procedure** `Unlink_Semaphore` (`Name`: **in** `POSIX.POSIX_String`);

11 **11.1.6.2 Description**

12 `Unlink_Semaphore` shall remove (i.e., make inaccessible) the semaphore named
13 by the string `Name`. If the semaphore specified by `Name` is currently referenced by
14 other processes, then `Unlink_Semaphore` shall have no effect on the state of the
15 semaphore. If one or more processes have the semaphore open when `Unlink_`
16 `Semaphore` is called, removal of the semaphore shall be postponed until all refer-
17 ences to the semaphore have been severed by calls to `Close`, `POSIX_Unsafe_Proc-`
18 `ess_Primitives.Exec`, `POSIX_Unsafe_Process_Primitives.Exec_Search`,
19 or `POSIX_Process_Primitives.Exit_Process`. The call to `Unlink_Semaphore`
20 shall not block the caller; it shall return immediately.

21 After `Unlink_Semaphore` is called, subsequent calls to `Open` or `Open_Or_Create`
22 to re-create or re-connect to the semaphore shall refer to a new semaphore object.

23 **11.1.6.3 Error Handling**

24 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
25 with the corresponding error code:

26 `Permission_Denied`

27 `Permission` is denied to unlink the named semaphore.

28 `Filename_Too_Long`

29 The length in POSIX characters of the specified pathname exceeds `Pathname`
30 `Limit` or the length in POSIX characters of a component of the specified path-
31 name is greater than `Filename_Maximum` and the `Filename Truncation` option is
32 not supported for the pathname prefix of that component.

33 `No_Such_File_Or_Directory`

34 The named semaphore does not exist.

35 `Operation_Not_Implemented`

36 `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 §2.2.2.124) 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 it either 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 occur, 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.

1 **11.1.8 Increment a Semaphore**

2 **11.1.8.1 Synopsis**

3 **procedure** Post (Sem: **in** Semaphore_Descriptor);

4 **11.1.8.2 Description**

5 Post performs the semaphore increment operation (see §2.2.2.125) on the semaphore
6 specified by Sem. If the semaphore Value resulting from this operation is positive,
7 then no tasks were blocked waiting to decrement the semaphore; the semaphore
8 Value is simply incremented.

9 If the Value of the semaphore resulting from this operation is zero, then one of the
10 tasks blocked waiting to decrement the semaphore shall be unblocked and allowed
11 to return from its call to Wait.

12 If the Priority Process Scheduling option is supported, and if more than one task is
13 blocked waiting to decrement the semaphore, the task to be unblocked shall be se-
14 lected in a manner appropriate to the scheduling policies and parameters in effect
15 for the blocked tasks and their processes.

16 If the Priority Process Scheduling option is not supported, the selection of which task to
17 unblock is unspecified.

18 **11.1.8.3 Error Handling**

19 If any of the following conditions occur, the exception POSIX_Error shall be raised,
20 with the corresponding error code:

21 Invalid_Argument

22 Sem is not a valid semaphore descriptor.

23 Operation_Not_Implemented

24 Post is not supported by this implementation.

25 **11.1.9 Get the Value of a Semaphore**

26 **11.1.9.1 Synopsis**

27 **function** Get_Value (Sem: Semaphore_Descriptor)
28 **return** Integer;

29 **11.1.9.2 Description**

30 Get_Value returns the Value attribute of the semaphore specified by Sem with-
31 out affecting the state of the semaphore. The value returned represents an actual
32 semaphore Value that occurred at some unspecified time during the call. This value
33 may be different from the actual Value of the semaphore when Get_Value returns.

34 If the Value of the semaphore specified by Sem is zero, then Get_Value shall return
35 either zero or a negative number whose absolute value represents the number of
36 tasks waiting for the semaphore at some unspecified time during the call.

11.1.9.3 Error Handling

If any of the following conditions occur, 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 below.

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`);

```

1      function Get_Locking_Policy
2          (Attr: Attributes)
3          return Locking_Policy;
4      procedure Set_Ceiling_Priority
5          (Attr: in out Attributes;
6            New_Ceiling: in Ceiling_Priority);
7      function Get_Ceiling_Priority (Attr: Attributes)
8          return Ceiling_Priority;
9      -- 11.2.6 Initialize and Finalize a Mutex
10     procedure Initialize
11         (M: in out Mutex;
12          Attr: in Attributes);
13     procedure Initialize (M: in out Mutex);
14     function Descriptor_Of (M: Mutex) return Mutex_Descriptor;
15     procedure Finalize (M: in out Mutex);
16     -- 11.2.7 Change the Ceiling Priority of a Mutex
17     procedure Set_Ceiling_Priority
18         (M: in Mutex_Descriptor;
19          New_Ceiling: in Ceiling_Priority;
20          Old_Ceiling: out Ceiling_Priority);
21     function Get_Ceiling_Priority (M: Mutex_Descriptor)
22         return Ceiling_Priority;
23     -- 11.2.8 Lock and Unlock a Mutex
24     procedure Lock (M: in Mutex_Descriptor);
25     function Try_Lock (M: Mutex_Descriptor) return Boolean;
26     procedure Unlock (M: in Mutex_Descriptor);
27 private
28     implementation-defined
29 end POSIX_Mutexes;

```

30 11.2.1 Mutex Ownership

31 A task becomes the *owner* of a mutex specified by the mutex descriptor M (see
32 §2.2.2.74) when either:

- 33 — a call to `Lock` with parameter M returns normally, or
- 34 — a call to `Try_Lock` with parameter M returns the value `True`, or
- 35 — a call to `POSIX_Condition_Variables.Wait` with parameter M returns nor-
36 mally or with an exception (except as explicitly indicated otherwise for certain
37 errors), or
- 38 — a call to `POSIX_Condition_Variables.Timed_Wait` with parameter M returns
39 normally or with an exception (except as explicitly indicated otherwise for cer-
40 tain errors).

41 The task remains the owner of the mutex specified by the mutex descriptor M until it
42 does one of the following:

- 43 — executes `Unlock` with parameter M, or
- 44 — is blocked in a call to `POSIX_Condition_Variables.Wait` with parameter M,
45 or
- 46 — is blocked in a call to `POSIX_Condition_Variables.Timed_Wait` with param-
47 eter 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 which has locked a mutex is implementation defined, and shall be documented for each implementation.

NOTE: It is the user's responsibility to ensure 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:

1 Process Shared
2 defined in §11.2.4.
3 Locking Policy
4 defined in §11.2.5.
5 Ceiling Priority
6 defined in §11.2.5.
7 Additional attributes, their default values, and the names of the associated opera-
8 tions to get and set those attribute values are implementation defined.
9 Initialize shall initialize the mutex attributes object Attr with the default value
10 for all of the attributes defined by the implementation.
11 The effect of initializing an already initialized mutex attributes object is undefined.
12 After a mutex attributes object has been used to initialize one or more mutexes, any
13 operation affecting the attributes object (including Finalize) does not affect any
14 previously initialized mutexes.
15 Finalize shall finalize the mutex attributes object Attr; the object becomes, in ef-
16 fect, uninitialized. An implementation may cause Finalize to set Attr to some
17 invalid value. A finalized mutex attributes object can be reinitialized using Ini-
18 tialize; the results of otherwise referencing the attributes object after it has been
19 finalized are undefined.

20 11.2.3.3 Error Handling

21 If any of the following conditions is detected, the exception POSIX_Error shall be
22 raised, with the corresponding error code:

23 Not_Enough_Space
24 Upon Initialize, insufficient memory exists to initialize the mutex at-
25 tributes object.
26 Invalid_Argument
27 Upon Finalize, Attr does not specify a properly initialized mutex at-
28 tributes object.

29 11.2.4 Mutex Process Shared Attribute

30 11.2.4.1 Synopsis

31 **function** Get_Process_Shared (Attr: Attributes)
32 **return** Boolean;
33 **procedure** Set_Process_Shared
34 (Attr: **in out** Attributes;
35 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 below.

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 only be operated upon 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 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.

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 below.

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 [Ada RM {1} D.2.1] is supported:

- 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 or not.
- 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:

- 1 — `Set_Ceiling_Priority` shall set the Ceiling Priority attribute of the mutex at-
- 2 tributes object specified by `Attr` to the value of `New_Ceiling`.
- 3 — `Get_Ceiling_Priority` shall return the Ceiling Priority attribute value of `Attr`.
- 4 — If `Set_Ceiling_Priority` and `Get_Ceiling_Priority` are called for a mutex
- 5 attributes object whose Locking Policy attribute is not equal to `Highest_Ceil-`
- 6 `ing_Priority`, `POSIX_Error` shall be raised.

7 NOTE: In order to avoid priority inversion, the priority ceiling of a mutex should be

8 set to a priority equal to or greater than the anticipated maximum active priority of

9 all tasks that may lock the mutex.

10 NOTE: All the rules above governing the locking and unlocking of mutexes also ap-

11 ply to the implicit locking and unlocking of mutexes through the `Wait` and `Timed_`

12 `Wait` operations on condition variables (see §3.3).

13 11.2.5.3 Error Handling

14 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,

15 with the corresponding error code:

16 Operation_Not_Implemented

17 Neither the Mutex Priority Inheritance option nor the Mutex Priority Ceiling option

18 are supported, and the implementation does not support the `Get_Lock-`

19 `ing_Policy` and `Set_Locking_Policy` operations.

20 There is no support for the Mutex Priority Ceiling option, and the implemen-

21 tation does not support the `Set_Ceiling_Priority` and `Get_Ceiling_`

22 `Priority` operations.

23 Operation_Not_Supported

24 The value specified to `Set_Locking_Policy` by `Locking` is an unsupported

25 value.

26 If any of the following conditions is detected, the exception `POSIX_Error` shall be

27 raised, with the corresponding error code:

28 Invalid_Argument

29 `Attr` does not specify a properly initialized mutex attributes object.

30 Either `Set_Ceiling_Priority` or `Get_Ceiling_Priority` were called

31 on a mutex attributes object whose Locking Policy attribute is not equal to

32 `Highest_Ceiling_Priority` locking policy.

33 Operation_Not_Permitted

34 The caller does not have the appropriate privilege to perform the operation.

35 11.2.6 Initialize and Finalize a Mutex

36 11.2.6.1 Synopsis

```

37 procedure Initialize
38   (M: in out Mutex;
39    Attr: in Attributes);
40 procedure Initialize (M: in out Mutex);
41 function Descriptor_Of (M: Mutex) return Mutex_Descriptor;
42 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 some 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 occur, 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 below.

`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 mutex's Ceiling Priority attribute 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 mutex's Ceiling Priority attribute shall not be changed.

11.2.7.3 Error Handling

If any of the following conditions occur, 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 there are tasks 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, and 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 there are tasks executing with the policies `Sched_FIFO` or `Sched_RR` 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 any of the following conditions occur, 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 mutex's ceiling. 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 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 below.

```
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);
```



```

1      -- 11.3.4 Initialize and Finalize a Condition
2      procedure Initialize
3          (Cond: in out Condition;
4           Attr: in Attributes);
5      procedure Initialize (Cond: in out Condition);
6      function Descriptor_Of (Cond: Condition)
7          return Condition_Descriptor;
8      procedure Finalize (Cond: in out Condition);
9      -- 11.3.5 Broadcast and Signal a Condition
10     procedure Signal (Cond: in Condition_Descriptor);
11     procedure Broadcast (Cond: in Condition_Descriptor);
12     -- 11.3.6 Wait on a Condition
13     procedure Wait
14         (Cond: in Condition_Descriptor;
15          M: in POSIX_Mutexes.Mutex_Descriptor);
16     procedure Timed_Wait
17         (Cond: Condition_Descriptor;
18          M: POSIX_Mutexes.Mutex_Descriptor;
19          Timeout: POSIX.Timespec);
20 private
21     implementation-defined
22 end POSIX_Condition_Variables;

```

23 11.3.1 Condition and Condition Descriptor Types

24 11.3.1.1 Synopsis

```

25     type Condition is limited private;
26     type Condition_Descriptor is private;

```

27 11.3.1.2 Description

28 The type Condition is used to represent condition variable objects.

29 Operating on copies of Condition objects (which may be created by returning func-
30 tion results of this type or by using the type as an **in out** parameter) results in
31 undefined behavior.

32 Objects of type Condition_Descriptor are used as condition variable descriptors,
33 that is, handles for condition variable objects. A condition variable descriptor is *valid*
34 if it is associated with a properly initialized condition variable, that is, the condition
35 variable descriptor is returned by a call to Descriptor_Of with a parameter that
36 has been used in a prior call to Initialize and not yet used in a call to Finalize.

37 Copies of Condition_Descriptor objects shall always specify the same condition
38 variable. Operating on an object of type Condition_Descriptor after the corre-
39 sponding condition variable object is finalized results in undefined behavior, which
40 might affect other processes sharing the same condition variable.

41 11.3.2 Condition Variable Attributes Type

42 11.3.2.1 Synopsis

```

43     type Attributes is private;
44     procedure Initialize (Attr: in out Attributes);
45     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 some 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 any of the following conditions occur, 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 any of the following conditions 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 below.

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 only be operated upon 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 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 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 some 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 occur, 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

These two operations 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, and 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 there are no tasks currently blocked on the condition variable specified by `Cond`.

11.3.5.3 Error Handling

If any of the following conditions is detected, the exception `POSIX_Error` shall be raised, with the corresponding error code:

`Invalid_Argument`

`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`; that is, 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, there must be an independent way to check whether the logical condition or event for which a task is waiting has actually occurred. This can be 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 a synchronization point with respect to the Ada **abort** statement (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 re-lock the mutex specified by `M`.

The operation `Wait` shall not be interruptible by a signal.

1 **11.3.6.3 Error Handling**

2 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
3 with the corresponding error code:

4 **Timed_Out**

5 The time specified by `Timeout` has passed.

6 If any of the following conditions is detected, the exception `POSIX_Error` shall be
7 raised, with the corresponding error code:

8 **Invalid_Argument**

9 `Cond` is not a valid condition variable descriptor.

10 `M` is not a valid mutex descriptor.

11 Different mutexes were specified for concurrent `Wait` or `Timed_Wait` oper-
12 ations on the same condition variable.

13 The mutex was not owned by the current task at the time of the call.

14 The `Timeout` value cannot be interpreted as a valid `Timespec` value. The
15 implementation may instead raise `Constraint_Error`.

16 Except for `Timed_Out`, all these error checks shall act as if they were performed
17 immediately at the beginning of the operation, and shall raise an exception prior to
18 modifying the state of the mutex specified by `M` or the condition variable specified by
19 `Cond`.

Section 12: Memory Management

This section specifies interfaces for memory management, including locking portions of a process's address space 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 run-time 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`, and 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 process's address space. Once a file is "mapped" into a process address space the data can be manipulated as memory. If more than one process maps 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 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`.

1 — Write attempts to memory that was mapped without write access, or any access
2 to memory mapped with `Protection_Options` equal to `POSIX.Empty_Set`,
3 shall result in `Signal_Segmentation_Violation`.

4 — References to unmapped addresses shall result in `Signal_Segmentation_-`
5 `Violation`.

6 NOTE: As specified in §3.3.3, the Ada language implementation is required to translate
7 occurrences of `Signal_Bus_Error` and `Signal_Segmentation_Violation` to `Pro-`
8 `gram_Error`, unless they are identifiable as corresponding to a check that requires some
9 other exception to be raised (such as `Storage_Error` or `Constraint_Error`).

10 If the Memory Protection option is not supported, the effect of references to unmapped
11 addresses is undefined.

12 In any case, the size of a memory object is unaffected by access beyond the end of the
13 object.

14 12.1 Package `POSIX_Memory_Locking`

15 This package provides access to services that a process can use to control whether all
16 of its address space is continually resident in memory.

17 *The functionality described in this clause is optional. If the Memory Locking option*
18 *is not supported, the implementation may cause all calls to the explicitly declared*
19 *operations defined in this clause to raise `POSIX_Error`. Otherwise, the behavior shall*
20 *be as specified below.*

```
21 with POSIX;
22 package POSIX_Memory_Locking is
23   -- 12.1.1 Lock/Unlock a Process's Address Space
24   type Memory_Locking_Options is new POSIX.Option_Set;
25   Current_Pages: constant Memory_Locking_Options:= implementation-defined;
26   Future_Pages:  constant Memory_Locking_Options:= implementation-defined;
27   procedure Lock_All (Options: in Memory_Locking_Options);
28   procedure Unlock_All;
29 end POSIX_Memory_Locking;
```

30 12.1.1 Lock/Unlock a Process's Address Space

31 12.1.1.1 Synopsis

```
32 type Memory_Locking_Options is new POSIX.Option_Set;
33 Current_Pages: constant Memory_Locking_Options:= implementation-defined;
34 Future_Pages:  constant Memory_Locking_Options:= implementation-defined;
35 procedure Lock_All (Options: in Memory_Locking_Options);
36 procedure Unlock_All;
```

37 12.1.1.2 Description

38 `Lock_All` shall cause all of the pages mapped by the process's address space to
39 be memory resident until unlocked, or until the process exits or issues `POSIX_-`
40 `Unsafe_Process_Primitives.Exec` or `POSIX_Unsafe_Process_Primitives.-`
41 `Exec_Search` for another process image. `Options` determines whether the pages to

be locked are those currently mapped by the process's address space, 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 process's address space.

Future_Pages

Lock all of the pages that become mapped into the process's address space 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 process's address space. After a call to `Unlock_All` any pages that become mapped into the process's address space 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 process's address space are also mapped into other processes' address spaces and are locked by those processes, the locks established by the other processes are unaffected by this process's call to `Unlock_All`.

Upon successful return from the `Lock_All` operation that specifies `Current_Pages`, all currently mapped pages of the process's address space shall be memory resident and locked. Upon return from the `Unlock_All` operation, all currently mapped pages of the process's address space shall be unlocked with respect to the process's address space. 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 occur, 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 process's address space would exceed an implementation defined limit on the amount of memory that the process can lock.

1 **Operation_Not_Permitted**
2 Upon **Lock_All**, the calling process does not have the appropriate privilege
3 to perform the requested operation.

4 **12.2 Package POSIX_Memory_Range_Locking**

5 This package provides access to services that a process can use to control whether
6 specific ranges of its address space are continually resident in memory.

7 *The functionality described in this clause is optional. If the Memory Range Locking op-*
8 *tion is not supported, the implementation may cause all calls to the explicitly declared*
9 *operations defined in this clause to raise POSIX_Error. Otherwise, the behavior shall*
10 *be as specified below.*

```
11  with System_Storage_Elements,
12       System;
13  package POSIX_Memory_Range_Locking is
14    -- 12.2.1 Lock/Unlock a Range of Process Address Space
15    procedure Lock_Range
16      (First:   in System.Address;
17       Length: in System_Storage_Elements.Storage_Offset);
18    procedure Unlock_Range
19      (First:   in System.Address;
20       Length: in System_Storage_Elements.Storage_Offset);
21  end POSIX_Memory_Range_Locking;
```

22 **12.2.1 Lock/Unlock a Range of Process Address Space**

23 **12.2.1.1 Synopsis**

```
24    procedure Lock_Range
25      (First:   in System.Address;
26       Length: in System_Storage_Elements.Storage_Offset);
27    procedure Unlock_Range
28      (First:   in System.Address;
29       Length: in System_Storage_Elements.Storage_Offset);
```

30 **12.2.1.2 Description**

31 **Lock_Range** shall cause those whole pages containing any part of the process's ad-
32 dress space starting at address **First** and continuing for **Length** storage units to be
33 memory resident until unlocked or until the process exits or issues **POSIX_Unsafe_-**
34 **Process_Primitives.Exec** or **POSIX_Unsafe_Process_Primitives.Exec_-**
35 **Search** for another process image. The implementation may require that **First** be
36 a multiple of **Page Size**.

37 NOTE: It is expected that a later amendment of this standard will disallow the imple-
38 mentation from imposing the restriction on the alignment of **First**.

39 **Unlock_Range** shall unlock those whole pages containing any part of the process's
40 address space starting at address **First** and continuing for **Length** storage units,
41 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 any 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 this process's call to `Unlock_Range`. If any of the pages in the range specified by a call to `Unlock_Range` are also mapped into other portions of the calling process's address space 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 process's address space. 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 occur, 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 process's address space.

`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 this violates an implementation-imposed alignment restriction.

`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 process's address space.

```

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

```

3   type Protection_Options is new POSIX.Option_Set;
4   Allow_Read:    constant Protection_Options:= implementation-defined;
5   Allow_Write:   constant Protection_Options:= implementation-defined;
6   Allow_Execute: constant Protection_Options:= implementation-defined;
7   type Mapping_Options is range implementation-defined;
8   Map_Shared:    constant Mapping_Options:= implementation-defined;
9   Map_Private:   constant Mapping_Options:= implementation-defined;
10  type Location_Options is range implementation-defined;
11  Exact_Address:  constant Location_Options:= implementation-defined;
12  Nearby_Address: constant Location_Options:= implementation-defined;
13  function Map_Memory
14    (First:      System.Address;
15     Length:     System.Storage_Elements.Storage_Offset;
16     Protection: Protection_Options;
17     Mapping:    Mapping_Options;
18     Location:   Location_Options;
19     File:       POSIX_IO.File_Descriptor;
20     Offset:     POSIX_IO.IO_Offset)
21    return System.Address;
22  function Map_Memory
23    (Length:     System.Storage_Elements.Storage_Offset;
24     Protection: Protection_Options;
25     Mapping:    Mapping_Options;
26     File:       POSIX_IO.File_Descriptor;
27     Offset:     POSIX_IO.IO_Offset)
28    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 below.

There are two overloaded subprograms 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 process's address space, 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 those whole pages containing any part of the specified region of the process's address space.

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 non-zero, 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 which 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 which 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 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 noted below.

1 If the Memory Protection option is supported, the implementation shall not permit a
2 write to succeed where `Allow_Write` has not been specified, or permit any access
3 where `Empty_Set` has been specified.

4 If the Memory Protection option is supported, the implementation shall support at least
5 the following values of `Protection`: `Empty_Set`, `Allow_Read`, `Allow_Write` and
6 the union ("+") of `Allow_Read` and `Allow_Write`. Otherwise, if the Memory Protection
7 option is not supported, the result of any access which conflicts with the specified
8 protection is undefined.

9 The file descriptor `File` is required to have been opened previously by the application
10 with read permission, regardless of the protection options specified. If `Allow_Write`
11 is specified, the application is required to have opened the file descriptor `File` with
12 write permission unless `Map_Private` is specified in `Mapping` as described below.

13 Mapping values of type `Mapping_Options` describe the disposition of write refer-
14 ences to the memory object.

15 If `Map_Shared` is specified, write references change the underlying object. If the im-
16 plementation supports `Map_Private` and it is specified, modifications to the mapped
17 data by the calling process shall be visible only to the calling process and shall not
18 change the underlying object. It is unspecified whether modifications to the under-
19 lying object done after the `Map_Private` mapping is established are visible through
20 the `Map_Private` mapping. `Mapping` is required to specify either `Map_Shared` or
21 `Map_Private`. The type of memory mapping in effect (private or shared) for each
22 region of the address space is retained across `POSIX_Unsafe_Process_Primi-`
23 `tives.Fork`.

24 Implementations in which the Memory Mapped Files option is not supported are not
25 required to support `Map_Private`.

26 The system shall always zero-fill any partial page at the end of an object. Further,
27 the system shall never write out any modified portions of the last page of an object
28 which are beyond the end of the object. If the Memory Protection option is supported,
29 references to addresses that are both within the mapped region and within whole
30 pages that are past the end of the mapped memory object shall result in raising
31 `Program_Error`. Otherwise, if the Memory Protection option is not supported, the
32 result of such references is undefined.

33 An implementation may raise the exception `Program_Error` when a reference would
34 cause an error in the mapped object, such as out of space condition.

35 **12.3.1.3 Error Handling**

36 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
37 with the corresponding error code:

38 **`Permission_Denied`**

39 The file descriptor `File` is not open for read, regardless of the protection
40 specified.

41 The file descriptor `File` is not open for write and `Allow_Write` was speci-
42 fied for a `Map_Shared` mapping.

1 **Resource_Temporarily_Unavailable**
2 The mapping could not be locked in memory, if required by `Lock_All` due
3 to a lack of resources.

4 **Bad_File_Descriptor**
5 The `File` argument is not a valid open file descriptor.

6 **Invalid_Argument**
7 The value in `Mapping` is invalid: not equal to either `Map_Private` or `Map_`
8 `Shared`.
9 The value in `Location` is invalid: not equal to either `Exact_Address` or
10 `Nearby_Address`.

11 **No_Such_Operation_On_Device**
12 The `File` argument refers to an object for which `Map_Memory` is meaning-
13 less, such as a terminal.

14 **Not_Enough_Space**
15 `Exact_Address` was specified, and the address range starting at `First`
16 and continuing for `Length` storage units exceeds that allowed for the ad-
17 dress space of a process; or `Exact_Address` was not specified and there is
18 insufficient room in the address space to effect the mapping.
19 The mapping could not be locked in memory, if required by `Lock_All` be-
20 cause it would require more space than the system is able to supply.

21 **Operation_Not_Implemented**
22 The operation `Map_Memory` is not supported by this implementation.

23 **Operation_Not_Supported**
24 `Exact_Address` was specified in the `Location` parameter and the imple-
25 mentation does not support this functionality.
26 `Map_Private` was specified in the `Mapping` parameter and the implemen-
27 tation does not support this functionality.
28 The implementation does not support the combination of accesses requested
29 in the `Protection` parameter.

30 **No_Such_Device_Or_Address**
31 The addresses in the range starting at `Offset` storage units and continuing
32 for `Length` storage units are invalid for the object specified by `File`.
33 `Exact_Address` was specified in `Location` and the combination of `First`,
34 `Length`, and `Offset` are invalid for the object specified by `File`.

35 If any of the following conditions is detected, the exception `POSIX_Error` shall be
36 raised, with the corresponding error code:

37 **Invalid_Argument**
38 The arguments `First` (if `Exact_Address` was specified) or `Offset` are not
39 multiples of `Page Size`, and this violates an implementation-imposed align-
40 ment restriction.

41 If `Map_Memory` fails for reasons other than `Bad_File_Descriptor`, `Invalid_`
42 `Argument`, or `Operation_Not_Supported`, some of the mappings in the address
43 range starting at `First` and continuing for `Length` storage units may have become
44 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 process's address space 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 implementation shall support at a minimum the recommended levels of representation support specified by Ada RM {1} C.2 Required Representation Support, and
- The implementation shall support the `Atomic`, `Atomic_Components`, `Volatile`, and `Volatile_Components` pragmas specified by Ada RM {1} C.6 Shared Variable Control.

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 below.

`Unmap_Memory` removes any mappings for those entire pages containing any part of the process's address space starting at `First` and continuing for `Length` storage units. Further references to these pages shall result in the exception `Program_Error` being raised. If there are no mappings 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 occur, 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 this violates an implementation-imposed alignment restriction.

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 below.

`Change_Protection` changes the access protections to be that specified by `Protection` for those whole pages containing any part of the process's address space starting at address `First` and continuing for `Length` storage units. The parameter `Protection` determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The `Protection` argument should 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 of 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 occur, 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 which 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`

The procedure `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 this violates an implementation-imposed alignment restriction.

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 below.

The Synchronize_Memory operation writes all modified data to permanent storage locations, if any, in those whole pages containing any part of the process's address space 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 has 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 such 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 occur, 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 which 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 this violates an implementation-imposed alignment restriction.

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 below.

```

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;

```

```
1      -- 12.4.2 Remove a Shared Memory Object
2      procedure Unlink_Shared_Memory
3          (Name: in POSIX.POSIX_String);
4      end POSIX_Shared_Memory_Objects;
```

5 12.4.1 Open a Shared Memory Object

6 12.4.1.1 Synopsis

```
7      function Open_Shared_Memory
8          (Name:          POSIX.POSIX_String;
9           Mode:          POSIX_IO.File_Mode;
10          Options:       POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
11          Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
12      return POSIX_IO.File_Descriptor;
13      function Open_Or_Create_Shared_Memory
14          (Name:          POSIX.POSIX_String;
15           Mode:          POSIX_IO.File_Mode;
16           Permissions:   POSIX.Permissions.Permission_Set;
17           Options:       POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
18           Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
19      return POSIX_IO.File_Descriptor;
```

20 12.4.1.2 Description

21 The `Open_Shared_Memory` and `Open_Or_Create_Shared_Memory` operations es-
22 tablish a connection between a shared memory object and a file descriptor. `Open_`-
23 `Shared_Memory` shall open a shared memory object that already exists. `Open_`-
24 `Or_Create_Shared_Memory` shall either open an existing shared memory object or
25 create a new shared memory object if none exists. Both functions create an open file
26 description that refers to the shared memory object and a file descriptor that refers
27 to that open file description. The file descriptor is used by other functions to refer to
28 that shared memory object.

29 The `Name` argument specifies a string naming a shared memory object. It is unspeci-
30 fied whether the name appears in the file system and is visible to other functions that
31 take pathnames as arguments. The `Name` argument shall conform to the construc-
32 tion rules for a pathname. If `Name` begins with the slash character, then processes
33 calling `Open_Shared_Memory` and `Open_Or_Create_Shared_Memory` with the same
34 value of `Name` shall refer to the same shared memory object, as long as that name has
35 not been removed. If `Name` does not begin with the slash character, the effect is im-
36 plementation defined. The interpretation of slash characters other than the leading
37 slash character in `Name` is implementation defined.

38 If successful, `Open_Shared_Memory` and `Open_Or_Create_Shared_Memory` return
39 a file descriptor for the shared memory object that is the lowest numbered file de-
40 scriptor not currently open for that process. The open file description is new, and
41 therefore the file descriptor does not share it with any other processes. It is unspec-
42 ified whether the file offset is set. The new file descriptor shall be marked such that
43 `POSIX_IO.Get_Close_On_Exec` would return `True`.

44 `Mode` specifies the file access modes of the open file description, using some of the
45 values of type `POSIX_IO.File_Mode`. Applications shall specify exactly one of the
46 following two values of type `POSIX_IO.File_Mode`:

1 **Read_Only**
2 Open for read access only.

3 **Read_Write**
4 Open for read or write access.

5 **Open_Shared_Memory** uses only one value from `POSIX_IO.Open_Option_Set` in
6 **Options**; the effect of setting other values from `POSIX_IO.Open_Option_Set` in
7 **Options** is undefined:

8 **Truncate**
9 If the shared memory object is successfully opened **Read_Write**, the ob-
10 ject shall be truncated to zero length and the mode and owner shall be un-
11 changed by this function call. The result of using **Truncate** with **Read_**
12 **Only** is undefined.

13 **Open_Or_Create_Shared_Memory** either opens an existing shared memory object,
14 except as noted under **Exclusive** below, or creates the shared memory object. If the
15 shared memory object is created, the shared memory object's user ID shall be set to
16 the effective user ID of the process; the shared memory object's group ID shall be set
17 to a system default group ID or to the effective group ID of the process. The shared
18 memory object's permissions shall be set to the value of the **Permissions** argument
19 except those set in the file mode creation mask of the process. The **Permissions**
20 argument does not affect whether the shared memory object is opened for reading, for
21 writing, or for both. The shared memory object shall have a size of zero. **Open_Or_**
22 **Create_Shared_Memory** uses either or both of the following values from `POSIX_`
23 **IO.Open_Option_Set** in **Options**; the effect of setting other values from `POSIX_`
24 **IO.Open_Option_Set** in **Options** is undefined:

25 **Exclusive**
26 If **Exclusive** is set, **Open_Or_Create_Shared_Memory** shall fail if the
27 shared memory object exists. The check for the existence of the shared mem-
28 ory object and the creation of the object if it does not exist shall be atomic
29 with respect to other processes executing **Open_Or_Create_Shared_Mem-**
30 **ory** naming the same shared memory object with **Exclusive** set.

31 **Truncate**
32 If the shared memory object exists, and it is successfully opened **Read_**
33 **Write**, the object shall be truncated to zero length and the mode and owner
34 shall be unchanged by this function call. The result of using **Truncate** with
35 **Read_Only** is undefined.

36 **Open_Shared_Memory** and **Open_Or_Create_Shared_Memory** shall be interruptible
37 by the delivery of a signal. **Masked_Signals** specifies the set of signals that shall be
38 added to the signal mask for the duration of this operation, as described in §3.3.6.

39 When a shared memory object is created, the state of the shared memory object, in-
40 cluding all data associated with the shared memory object, shall persist until the
41 shared memory object is unlinked and all other references are gone. It is unspeci-
42 fied whether the name and shared memory object state remain valid after a system
43 reboot.

44 NOTE: The `POSIX_IO.Close` operation, which is specified in §6.1.1, is used to close a
45 shared memory object.

12.4.1.3 Error Handling

If any of the following conditions occur, 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.

`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, but 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 occur, 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.

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 must be a constrained type and be 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 below.

```
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;
```

```

1      -- 12.5.1 Open Shared Memory
2      function Open_And_Map_Shared_Memory
3          (Name:          POSIX.POSIX_String;
4           Protection:    POSIX_Memory_Mapping.Protection_Options;
5           Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
6          return POSIX_IO.File_Descriptor;
7      function Open_Or_Create_And_Map_Shared_Memory
8          (Name:          POSIX.POSIX_String;
9           Protection:    POSIX_Memory_Mapping.Protection_Options;
10          Permissions:    POSIX_Permissions.Permission_Set;
11          Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
12          Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
13          return POSIX_IO.File_Descriptor;
14      -- 12.5.2 Access Shared Memory
15      function Access_Shared_Memory
16          (File: POSIX_IO.File_Descriptor)
17          return Shared_Access;
18      -- 12.5.3 Close Shared Memory
19      procedure Unmap_And_Close_Shared_Memory
20          (File: in POSIX_IO.File_Descriptor);
21      -- 12.5.5 Lock/Unlock Shared Memory
22      procedure Lock_Shared_Memory
23          (File: in POSIX_IO.File_Descriptor);
24      procedure Unlock_Shared_Memory
25          (File: in POSIX_IO.File_Descriptor);
26      end POSIX_Generic_Shared_Memory;

```

27 12.5.1 Open Shared Memory

28 12.5.1.1 Synopsis

```

29      function Open_And_Map_Shared_Memory
30          (Name:          POSIX.POSIX_String;
31           Protection:    POSIX_Memory_Mapping.Protection_Options;
32           Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
33          return POSIX_IO.File_Descriptor;
34      function Open_Or_Create_And_Map_Shared_Memory
35          (Name:          POSIX.POSIX_String;
36           Protection:    POSIX_Memory_Mapping.Protection_Options;
37           Permissions:    POSIX_Permissions.Permission_Set;
38           Options:        POSIX_IO.Open_Option_Set:= POSIX_IO.Empty_Set;
39           Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
40          return POSIX_IO.File_Descriptor;

```

41 12.5.1.2 Description

42 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.

48 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 that are defined elsewhere in this standard, using the supplied parameter values together with the listed implicit parameter values.

- (1) `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`.
- (2) `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 must specify a count of storage units sufficient to contain an Ada object of the generic actual type `Object_Type`.
- (3) `Map_Memory` to map the POSIX shared memory object into the process's address space. The `File` parameter corresponds to the value returned by `Open_Shared_Memory`, if that operation is successful. The `Length` parameter must 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.

- (1) `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`.
- (2) `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 must specify a count of storage units sufficient to contain an Ada object of the generic actual type `Object_Type`.
- (3) `Map_Memory` to map the POSIX shared memory object into the process's address space. The `File` parameter corresponds to the value returned by `Open_Or_Create_Shared_Memory`, if that operation is successful. The `First` and `Length` parameters must 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

1 be atomic with respect to other processes executing `Open_Or_Create_-`
2 `And_Map_Shared_Memory` naming the same shared memory object with
3 `Exclusive` set.

4 If any operation on the type `POSIX_IO.File_Descriptor` that is not specified in
5 §12.5 is applied to a file descriptor returned by `Open_And_Map_Shared_Memory`
6 or `Open_Or_Create_And_Map_Shared_Memory`, the effect of that and subsequent
7 operations on that file descriptor is undefined.

8 `Open_And_Map_Shared_Memory` and `Open_Or_Create_And_Map_Shared_Memory`
9 shall be interruptible by the delivery of a signal. `Masked_Signals` specifies the set
10 of signals that shall be added to the signal mask for the duration of this operation,
11 as described in §3.3.6.

12 If the operation propagates an exception, it shall have no residual effect. In partic-
13 ular, no new shared memory object shall exist, no new memory mapping shall exist,
14 and no new file descriptor shall be allocated.

15 **12.5.1.3 Error Handling**

16 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
17 with the corresponding error code:

18 **Filename_Too_Long**

19 The length in POSIX characters of the specified pathname exceeds `Pathname`
20 `Limit` or the length in POSIX characters of a component of the specified path-
21 name is greater than `Filename_Maximum` and the `Filename Truncation` option is
22 not supported for the pathname prefix of that component.

23 **File_Exists**

24 `Open_Or_Create_Shared_Memory` was called with `Exclusive` specified in
25 `Options` and the named shared memory object already exists.

26 **Interrupted_Operation**

27 The operation was interrupted by a signal.

28 **Invalid_Argument**

29 `Permissions` is the empty set.

30 The operation is not supported for the given name. The implementation
31 shall document under what circumstances this error may be returned.

32 **No_Such_File_Or_Directory**

33 `Open_Shared_Memory` was called and the named shared memory object
34 does not exist.

35 **No_Space_Left_On_Device**

36 There is insufficient space for the creation of the new shared memory object.

37 **Not_Enough_Space**

38 There is insufficient room in the address space to effect the mapping, or the
39 mapping could not be locked in memory, if required by `Lock_All`, because
40 it would require more space than the system is able to supply.

- 1 **Operation_Not_Implemented**
2 The operation is not supported by this implementation.
- 3 **Operation_Not_Supported**
4 The implementation does not support the combination of accesses requested
5 in the **Protection** argument.
- 6 **Permission_Denied**
7 The shared memory object exists and the operation requests a mode of ac-
8 cess that is denied, or the shared memory object does not exist and permis-
9 sion to create the shared memory object is denied, or **POSIX_IO.Truncate**
10 is specified and write permission is denied.
- 11 **Resource_Temporarily_Unavailable**
12 The mapping could not be locked in memory, if required by **Lock_All**, due
13 to a lack of resources.
- 14 **Too_Many_Open_Files**
15 Too many file descriptors are currently in use by this process.
- 16 **Too_Many_Open_Files_In_System**
17 Too many shared memory objects are currently open in the system.

18 **12.5.2 Access Shared Memory**

19 **12.5.2.1 Synopsis**

20 **function** **Access_Shared_Memory**
21 (**File**: **POSIX_IO.File_Descriptor**)
22 **return** **Shared_Access**;

23 **12.5.2.2 Description**

24 If **File** is an open file descriptor returned by a call to **Open_And_Map_Shared_**-
25 **Memory** or **Open_Or_Create_And_Map_Shared_Memory**, then a call to **Access_**-
26 **Shared_Memory** of the same generic instantiation shall return an access value that
27 designates the Ada object of type **Object_Type** within the shared memory object
28 designated by the file descriptor.

29 **12.5.2.3 Error Handling**

30 If any of the following conditions occur, the exception **POSIX_Error** shall be raised,
31 with the corresponding error code:

- 32 **Bad_File_Descriptor**
33 File is not the file descriptor for an open shared memory object defined by
34 this generic instantiation.

35 **12.5.3 Close Shared Memory**

36 **12.5.3.1 Synopsis**

37 **procedure** **Unmap_And_Close_Shared_Memory**
38 (**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 process's address space.
- (2) A call to `POSIX_IO.Close` to close the file descriptor.

12.5.3.3 Error Handling

If any of the following conditions occur, 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 below.

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.

1 NOTE: These operations do not provide locking in the sense of protection against concur-
2 rent access. The meaning of “lock” here is that the range of virtual addresses mapped by
3 the shared memory object are locked into main memory; they cannot be paged out onto
4 a secondary storage device.

5 **12.5.5.3 Error Handling**

6 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
7 with the corresponding error code:

8 **`Bad_File_Descriptor`**

9 File is not the file descriptor for a shared memory object defined by this
10 generic instantiation.

11 **`Operation_Not_Implemented`**

12 The implementation does not support this memory locking operation.

13 **`Resource_Temporarily_Unavailable`**

14 Some or all of the memory mapped to the shared memory object could not
15 be locked when the call was made.

16 **`Not_Enough_Space`**

17 Locking the pages mapped by the specified shared memory object would
18 exceed an implementation defined limit on the amount of memory that the
19 process can lock.

20 **`Operation_Not_Permitted`**

21 The calling process does not have the appropriate privilege to perform the
22 requested operation.

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

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, in the Rationale, 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.1b {3} and POSIX.1c {4}.

The Scheduling Contention Scope attribute of a task shall correspond to the scheduling contention scope of the associated thread of control in [POSIX.1c {4} 13]. 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.1c {4}.

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 below.

```
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
```

```

1      (Process:    POSIX_Process_Identification.Process_ID)
2      return Scheduling_Parameters;
3      procedure Set_Scheduling_Policy
4      (Process:    in POSIX_Process_Identification.Process_ID;
5       New_Policy: in Scheduling_Policy;
6       Parameters: in Scheduling_Parameters);
7      function Get_Scheduling_Policy
8      (Process: POSIX_Process_Identification.Process_ID)
9      return Scheduling_Policy;
10     -- 13.2.4 Process Yield CPU
11     procedure Yield;
12     -- 13.2.5 Get Scheduling Limits
13     function Get_Maximum_Priority (Policy: Scheduling_Policy)
14     return Scheduling_Priority;
15     function Get_Minimum_Priority (Policy: Scheduling_Policy)
16     return Scheduling_Priority;
17     function Get_Round_Robin_Interval
18     (Process: POSIX_Process_Identification.Process_ID)
19     return POSIX.Timespec;
20     private
21     implementation-defined
22     end POSIX_Process_Scheduling;

```

23 13.2.1 Scheduling Parameters

24 13.2.1.1 Synopsis

```

25     subtype Scheduling_Priority is Integer;
26     type Scheduling_Parameters is private;
27     function Get_Priority (Parameters: Scheduling_Parameters)
28     return Scheduling_Priority;
29     procedure Set_Priority
30     (Parameters: in out Scheduling_Parameters;
31      Priority:    in Scheduling_Priority);

```

32 13.2.1.2 Description

33 Values of the subtype `Scheduling_Priority` are used to specify the Scheduling Pri-
34 ority attribute of processes. The range of priority values that are valid for scheduling
35 processes is implementation defined.

36 Values of the type `Scheduling_Parameters` are used to specify the per-process
37 Scheduling Parameters required for each scheduling policy supported. This type shall
38 support at least one attribute, Scheduling Priority, of type `Scheduling_Priority`.

39 `Set_Priority` shall set the Scheduling Priority attribute of the object specified by
40 `Parameters` to the value specified by `Priority`.

41 `Get_Priority` shall return the Scheduling Priority attribute of the object specified by
42 `Parameters`.

43 Implementations may add other attributes, as permitted in §1.3.1.1, point (2).

44 NOTE: Adding attributes which may change the behavior of applications with respect
45 to this standard when those attributes are uninitialized also requires that the extension
46 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.1b {3} 13.2].

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.)

1 — For tasks with `System_Wide` Scheduling Contention Scope, these operations shall
2 have no effect on their scheduling.

3 — For tasks with `Within_Process` scheduling contention scope, scheduling with
4 respect to threads of control in other processes may be dependent on the schedul-
5 ing policy and scheduling parameters of their process, which is governed using
6 these functions.

7 `Get_Scheduling_Parameters` shall return the scheduling parameters of the pro-
8 cess specified by `PROCESS`, provided that such a process exists and the calling process
9 has permission.

10 `Get_Scheduling_Policy` shall return the Scheduling Policy of the process specified by
11 `PROCESS`, provided that such a process exists and the calling process has permission.

12 For `Set_Scheduling_Parameters` and `Set_Scheduling_Policy`:

13 — Implementations may require that the requesting process have permission to
14 set its own scheduling parameters or those of another process. Additionally,
15 implementation defined restrictions may apply as to the appropriate privilege
16 required to set a process's own Scheduling Policy, or another process's Scheduling
17 Policy, to a particular value.

18 — If the new Scheduling Policy of the process is `Sched_FIFO` or `Sched_RR`, only the
19 Scheduling Priority attribute is required to be supported. In this case, the Scheduling
20 Priority scheduling parameter is required to be within the range of valid priorities
21 for the new scheduling policy.

22 — If the new Scheduling Policy is `Sched_Other`, the affected scheduling parameters
23 are implementation defined.

24 — If the operation fails it shall propagate an exception, and the Scheduling Policy
25 and scheduling parameters of the specified process shall not have been affected.

26 NOTE: A call to `Set_Scheduling_Parameters` or `Set_Scheduling_Policy` may
27 result in preemption of the calling task before the call returns.

28 **13.2.3.3 Error Handling**

29 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
30 with the corresponding error code:

31 **Invalid_Argument**

32 One or more of the requested scheduling parameters is outside the range
33 defined for the specified `PROCESS`'s Scheduling Policy.

34 The value of the `Policy` parameter is invalid, or one or more of the at-
35 tributes of `Parameters` is outside the valid range for the specified schedul-
36 ing policy.

37 **Operation_Not_Implemented**

38 The operation is not supported by this implementation.

39 **Operation_Not_Permitted**

40 The requesting process does not have permission to perform the operation
41 for the specified process.

1 The requesting process does not have permission to invoke `Set_Scheduling_Parameters`.

3 `No_Such_Process`

4 No process can be found corresponding to that specified by `Process`.

5 **13.2.4 Process Yield CPU**

6 **13.2.4.1 Synopsis**

7 **procedure** `Yield`;

8 **13.2.4.2 Description**

9 The `Yield` operation shall cause the calling task to yield the processor to other tasks
10 of equal process or task priority. The specific effects are implementation defined.

11 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
12 where tasks are implemented via interleaved execution of a per-process kernel entity.

14 NOTE: See §13.3.4 for a task-level yield operation.

15 **13.2.4.3 Error Handling**

16 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
17 with the corresponding error code:

18 `Operation_Not_Implemented`

19 The `Yield` operation is not supported by this implementation.

20 **13.2.5 Get Scheduling Limits**

21 **13.2.5.1 Synopsis**

22 **function** `Get_Maximum_Priority` (`Policy`: `Scheduling_Policy`)
23 **return** `Scheduling_Priority`;
24 **function** `Get_Minimum_Priority` (`Policy`: `Scheduling_Policy`)
25 **return** `Scheduling_Priority`;
26 **function** `Get_Round_Robin_Interval`
27 (`Process`: `POSIX_Process_Identification.Process_ID`)
28 **return** `POSIX.Timespec`;

29 **13.2.5.2 Description**

30 `Get_Maximum_Priority` and `Get_Minimum_Priority` shall return the maximum
31 and minimum priority values that are valid for the policy specified by `Policy`.

32 `Get_Round_Robin_Interval` shall return the current *round-robin-interval* of the
33 process specified by `Process`. The value returned represents the quantum under the
34 `Sched_RR` policy at which a scheduling decision shall be made, when another process
35 at the same priority is ready to execute. (See [POSIX.1b {3} 13.2.2].)

13.2.5.3 Error Handling

If any of the following conditions occur, 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 [Ada RM {1} D.1] and the pragmas and package interfaces defined in [Ada RM {1} D.2-D.5].

13.3.1 Dynamic Priorities

NOTE: The initial base priority of a task may be specified via the `Priority` pragma, defined in [Ada RM {1} D.1]. Thereafter, the base priority may be modified by means of calls to the procedure `Set_Priority`, which is defined in `Ada.Dynamic_Priorities` [Ada RM {1} D.5].

Implementations of this standard that support child library packages [Ada RM {1} 10.1] shall support the package `Ada.Dynamic_Priorities` and provide a library-level renaming of that package as `Ada_Dynamic_Priorities`.

During transition from Ada83 (ISO/IEC 8652: 1987) to Ada95 (ISO/IEC 8652: 1995), 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 [Ada RM {1} D.2.2] 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.1c {4} 13], for some implementation defined correspondence of Ada tasks to POSIX threads.

1 NOTE: The effect of the `Task_Dispatching_Policy` pragma is to specify the policy
 2 by which all the tasks in a process are scheduled. In contrast, the effect of the `Task_`
 3 `Creation_Attributes` pragma is to specify the scheduling policy for tasks of just one
 4 type. Thus, although the rules for the Ada `FIFO_Within_Priorities` task dispatching
 5 policy are very similar to the rules for the POSIX `Sched_FIFO` task scheduling policy,
 6 one should not be confused with the other. The former is process-wide and the latter is
 7 task-specific.

8 **13.3.3 Task Creation Attributes Pragma**

9 The Ada implementation shall support the `Task_Creation_Attributes` pragma.
 10 The `Task_Creation_Attributes` pragma is allowed within a task specification, in
 11 which case it shall apply to all tasks of the specified task type. Only one occurrence
 12 of the pragma is allowed in each task specification.

13 The `Task_Creation_Attributes` pragma shall be recognized at least for compi-
 14 lations that have specified `POSIX_Task_Dispatching` as the partition-wide task
 15 dispatching policy. Otherwise, it may be rejected or not recognized.

16 The `Task_Creation_Attributes` pragma shall allow (at least) argument associ-
 17 ations with argument identifiers `Scheduling_Contention_Scope` and `Schedul-`
 18 `ing_Policy`, in any order. For the argument identifier `Scheduling_Contention_`
 19 `Scope`, at least the names `System_Wide` and `Within_Process` shall be recognized.
 20 For the argument identifier `Scheduling_Policy`, at least the names `Sched_FIFO`,
 21 `Sched_RR`, and `Sched_Other` shall be recognized.

22 An implementation may support implementation defined additional argument iden-
 23 tifiers for the `Task_Creation_Attributes` pragma, and additional scheduling pol-
 24 icy names. Scheduling policy names shall begin with “`Sched_`”.

25 **13.3.3.1 Error Handling**

26 If the `Task_Creation_Attributes` pragma is not supported, or the specified com-
 27 bination of arguments is not supported, either the pragma shall not be recognized,
 28 the compilation shall be rejected, or `POSIX_Error` shall be raised during the creation
 29 or activation of the task to which the pragma is applied.

30 If any of the following conditions are detected during the creation or activation of a
 31 task containing the `Task_Creation_Attributes` pragma, the exception `POSIX_`
 32 `Error` shall be raised, with the corresponding error code:

33 `Invalid_Argument`

34 The name or expression specified for a supported argument identifier is not
 35 valid.

36 `Operation_Not_Supported`

37 The combination of argument associations given in the pragma is not sup-
 38 ported.

39 **13.3.4 Task Yield CPU**

40 Execution of a delay statement with duration zero shall cause the calling task to
 41 yield the processor to other ready tasks of equal priority. The specific effects are
 42 implementation defined.

1 13.4 Synchronization Scheduling

2 This clause specifies the interactions between the synchronization operations of Sec-
3 tion 11 and priority task scheduling.

4 When a task owns a mutex with the `No_Priority_Inheritance` protocol attribute
5 (see §11.2.5), its priority and scheduling are not affected by its mutex ownership.

6 When a task is blocking higher priority tasks because of owning one or more mutexes
7 with the `Highest_Blocked_Task` protocol attribute, it inherits the active priorities
8 of all the tasks waiting on any of the mutexes owned by this task and initialized with
9 this protocol.

10 When a task owns one or more mutexes initialized with the `Highest_Ceiling_Pri-`
11 `ority` protocol, it inherits the ceiling priorities of all the mutexes owned by this task
12 and initialized with this attribute, regardless of whether other tasks are blocked on
13 any of these mutexes or not. The `Ceiling Priority` attribute defines the ceiling priority of
14 initialized mutexes, which is the minimum priority level at which the critical section
15 guarded by the mutex is executed. In order to avoid priority inversion, the `Ceiling Pri-`
16 `ority` of the mutex shall be set to a priority higher than or equal to the highest priority
17 of all the tasks that may lock that mutex.

18 When a task makes a call to `POSIX_Mutexes.Lock`, if the `Mutex Priority Inheritance`
19 option is supported and the mutex was initialized with the protocol attribute hav-
20 ing the value `Highest_Blocked_Task`, when the calling task is blocked because the
21 mutex is owned by another task, that owner task shall inherit the priority of the call-
22 ing task as long as it continues to own the mutex. The implementation shall update
23 its execution priority to the maximum of its assigned priority and all its inherited
24 priorities. Furthermore, if this owner task itself becomes blocked on another mutex,
25 the same priority inheritance effect shall be propagated to this other owner task, in
26 a recursive manner.

27 NOTE: If a task simultaneously owns several mutexes initialized with different proto-
28 cols, it will inherit all the priorities that it would have obtained by each of these proto-
29 cols.

Section 14: Clocks and Timers

1 This section describes the high-resolution clock and timer facilities available under
2 this standard.

3 *The functionality described in this section is optional. If the Timers option is not sup-*
4 *ported, the implementation may cause all calls to the explicitly declared operations*
5 *defined in this section to raise POSIX_Error. Otherwise, the behavior shall be as*
6 *specified below.*

7 14.1 Package POSIX_Timers

8 This package provides access to services that can be used to read and set the value
9 of a clock, obtain the resolution of a clock, create a timer, delete a timer, arm and
10 disarm a timer, and obtain the state of a timer.

```
11 with POSIX,  
12     POSIX_Signals;  
13 package POSIX_Timers is  
14     -- 14.1.1 Clock and Timer Types  
15     type Clock_ID is private;  
16     type Timer_ID is private;  
17     -- 14.1.2 Realtime Clock  
18     Clock_Realttime: constant Clock_ID;  
19     -- 14.1.3 Timer State and Timer Options  
20     type Timer_State is private;  
21     type Timer_Options is new POSIX.Option_Set;  
22     Absolute_Timer: constant Timer_Options := implementation-defined;  
23     procedure Set_Initial  
24         (State: in out Timer_State;  
25          Initial: in POSIX.Timespec);  
26     function Get_Initial  
27         (State: Timer_State) return POSIX.Timespec;  
28     procedure Set_Interval  
29         (State: in out Timer_State;  
30          Interval: in POSIX.Timespec);  
31     function Get_Interval  
32         (State: Timer_State) return POSIX.Timespec;  
33     -- 14.1.4 Clock Operations  
34     procedure Set_Time  
35         (Clock: in Clock_ID;  
36          Value: in POSIX.Timespec);  
37     function Get_Time (Clock: Clock_ID)  
38         return POSIX.Timespec;  
39     function Get_Resolution (Clock: Clock_ID)  
40         return POSIX.Timespec;  
41     -- 14.1.5 Create a Timer  
42     function Create_Timer  
43         (Clock: Clock_ID;  
44          Event: POSIX_Signals.Signal_Event)  
45         return Timer_ID;  
46     -- 14.1.6 Delete a Timer  
47     procedure Delete_Timer  
48         (Timer: in out Timer_ID);
```



```

1      -- 14.1.7 Timer Operations
2      procedure Arm_Timer
3          (Timer:      in  Timer_ID;
4            Options:   in  Timer_Options;
5            New_State: in  Timer_State;
6            Old_State: out Timer_State);
7      procedure Arm_Timer
8          (Timer:      in  Timer_ID;
9            Options:   in  Timer_Options;
10           New_State: in  Timer_State);
11     function Get_Timer_State
12         (Timer: Timer_ID) return Timer_State;
13     procedure Disarm_Timer
14         (Timer: in Timer_ID);
15     function Get_Timer_OVERRUNS
16         (Timer: Timer_ID) return Natural;
17 private
18     implementation-defined
19 end POSIX_Timers;

```

20 14.1.1 Clock and Timer Types

21 14.1.1.1 Synopsis

```

22     type Clock_ID is private;
23     type Timer_ID is private;

```

24 14.1.1.2 Description

25 Clocks are used to measure the passage of time. Timers are based on clocks, and can
 26 be used to trigger signals.

27 Objects of type `Clock_ID` are used to identify clocks in clock and timer operations.

28 Objects of type `Timer_ID` are used to identify timers in timer operations.

29 A value of type `Timer_ID` is *valid* if it has been returned by function `Create_Timer`
 30 (see §14.1.5) and not yet provided as a parameter to `Delete_Timer` (see §14.1.6).

31 14.1.2 Realtime Clock

32 14.1.2.1 Synopsis

```

33     Clock_Realtime: constant Clock_ID;

```

34 14.1.2.2 Description

35 Clocks measure the passage of time. A clock can measure real time or some other
 36 implementation defined concept of time, such as the processor time used by the pro-
 37 cess.

38 Every conforming implementation shall provide at least one system-wide realtime
 39 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: Fully Conforming applications should not 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 “`Clock_`”.

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 which includes 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 which 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 note in §14.1.7 implies the existence of another hidden attribute of timer state, namely, whether or not the option `Absolute_Timer` was specified in the `Arm_Timer` call which 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 non-negative 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 §2.2.2.39).

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 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, is implementation defined.

14.1.4.3 Error Handling

If any of the following conditions occur, 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 any of the following conditions 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;
```

1 **14.1.5.2 Description**

2 An application can create a timer, associated with a specific clock, which later can be
3 armed to provide one-shot or periodic notifications of timer expiration. Notifications
4 of timer expiration are provided to the application as signal events.

5 Each implementation

6 shall define a set of clocks, which shall include the system-wide realtime clock
7 `Clock_Realtime`, that can be used as timing bases for timers.

8 The `Create_Timer` function shall create a timer using the specified `Clock` as a
9 timing base and shall return the identifier of the timer which can be used in later
10 timer operations. This timer identifier shall be unique within the calling process
11 until the timer is deleted. The timer whose identification is returned shall be in a
12 disarmed state upon return from `Create_Timer`.

13 The `Event` parameter identifies a `Signal_Event` object which defines the asyn-
14 chronous notification that will occur when the timer expires. If the `Notification` at-
15 tribute of `Event` is `Signal_Notification`, then `Event` shall specify the signal
16 number and the application-specific data value to be sent upon timer expiration.
17 The `Event` parameter must not specify a reserved signal of the Ada language imple-
18 mentation. If the `Notification` attribute of `Event` is `No_Notification`, no notification
19 shall be sent upon timer expiration. The behavior for any other value of `Notification` is
20 implementation defined.

21 If the `Event` parameter specifies `Signal_Notification`, the `Signal` specified by
22 `Event` shall be generated upon timer expiration.

23 If the `Realtime Signals` option is supported, and if the signal is in the range `Real-`
24 `time_Signal`, and if signal queueing is enabled for the specified signal, the signal
25 shall be queued with the `Data` value specified by `Event`; otherwise, it is unspecified
26 whether the signal is queued and what `Data` value, if any, is queued with it.

27 Timers shall not be inherited by a child process created by a call to `POSIX_Unsafe_-`
28 `Process_Primitives.Fork`, or `POSIX_Process_Primitives.Start_Process`,
29 or `POSIX_Process_Primitives.Start_Process_Search` and shall be disarmed
30 and deleted by a call to `POSIX_Unsafe_Process_Primitives.Exec` or `POSIX_-`
31 `Unsafe_Process_Primitives.Exec_Search`.

32 The value returned is undefined if the `Create_Timer` function fails.

33 If the Ada implementation reserves any timers, the value returned by `Timers_-`
34 `Maximum` (see §4.5) shall be reduced to reflect the number of timers actually available
35 to the application.

36 **14.1.5.3 Error Handling**

37 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
38 with the corresponding error code:

39 `Resource_Temporarily_Unavailable`

40 The system lacks sufficient signal queueing resources to honor the request.

1 The calling process has already created all the timers it is allowed by this
2 implementation.

3 **Invalid_Argument**

4 Clock is not a valid clock identifier.

5 The application is attempting to create a timer that delivers a signal that is
6 reserved for the Ada implementation (see §2.2.2.117).

7 **Operation_Not_Supported**

8 The function `Create_Timer` is not supported by this implementation.

9 **14.1.6 Delete a Timer**

10 **14.1.6.1 Synopsis**

11 **procedure** `Delete_Timer`
12 (`Timer`: **in out** `Timer_ID`);

13 **14.1.6.2 Description**

14 A previously created timer can be deleted.

15 The `Delete_Timer` procedure shall delete the timer specified by `Timer`, previously
16 created by `Create_Timer`. If the specified timer is armed when `Delete_Timer` is
17 called, the behavior shall be as if the timer is automatically disarmed before dele-
18 tion. The disposition of pending signals (if any) generated by the deleted timer is
19 unspecified. An implementation may cause `Delete_Timer` to set `Timer` to some
20 invalid value. The effect of using `Timer` in subsequent timer operations, except in a
21 subsequent call to `Create_Timer`, is undefined.

22 **14.1.6.3 Error Handling**

23 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
24 with the corresponding error code:

25 **Invalid_Argument**

26 `Timer` is not a valid timer identifier.

27 **Operation_Not_Supported**

28 The procedure `Delete_Timer` is not supported by this implementation.

29 **14.1.7 Timer Operations**

30 **14.1.7.1 Synopsis**

31 **procedure** `Arm_Timer`
32 (`Timer`: **in** `Timer_ID`;
33 `Options`: **in** `Timer_Options`;
34 `New_State`: **in** `Timer_State`;
35 `Old_State`: **out** `Timer_State`);
36 **procedure** `Arm_Timer`
37 (`Timer`: **in** `Timer_ID`;
38 `Options`: **in** `Timer_Options`;
39 `New_State`: **in** `Timer_State`);

```
1      function Get_Timer_State
2      (Timer: Timer_ID) return Timer_State;
3      procedure Disarm_Timer
4      (Timer: in Timer_ID);
5      function Get_Timer_OVERRUNS
6      (Timer: Timer_ID) return Natural;
```

7 **14.1.7.2 Description**

8 Once created, a timer must be armed in order for signals to be generated upon timer
9 expiration.

10 The Arm_Timer procedure shall set the time until next expiration of the timer spec-
11 ified by Timer from the Initial attribute of New_State and shall arm the timer if this
12 value is positive.

13 If Absolute_Timer is specified by Options, then the timer specified by Timer shall
14 be armed as an absolute timer. The time of the next timer expiration shall be set to
15 the value specified by the Initial attribute of New_State, provided this value is not
16 zero. The timer shall expire when the clock on which the timer is based reaches the
17 time specified by the Initial attribute of New_State.

18 If the Initial attribute of New_State is zero, the implementation shall detect the con-
19 dition and the call to Arm_Timer shall fail.

20 If the Initial attribute of New_State is not zero, but is less than or equal to the current
21 value of the clock on which the specified timer is based, the call to Arm_Timer shall
22 succeed and immediate expiration notification shall be made.

23 If Absolute_Timer is not specified by Options, then the timer specified by Timer
24 shall be armed as a relative timer. The time of the next timer expiration shall be set
25 as specified by the Initial attribute of New_State, provided the value is positive. The
26 value of the Initial attribute of New_State shall be added to the current value of the
27 clock on which the specified timer is based in order to obtain the future value of time
28 at which the timer shall expire.

29 If the Initial attribute of New_State is zero or negative, the implementation shall
30 detect the condition and the call to Arm_Timer shall fail.

31 If the timer specified by Timer is already armed, Arm_Timer shall reset the time
32 until next expiration, as specified by Options and the Initial attribute of New_State.
33 The effect on pending expiration notifications of resetting the timer is unspecified.

34 The effect of the Interval attribute of New_State is to specify a reload value when the
35 timer expires. When this value is zero, a one-shot timer is armed. Upon expiration,
36 the timer shall not be automatically rearmed. When this value is positive, a periodic
37 timer is specified. Upon timer expiration, the value of the Interval attribute of New_
38 State shall be added to the current value of the clock on which the timer is based to
39 obtain the next expiration time of the timer, and the timer shall be rearmed.

40 Time values that are between two consecutive non-negative integer multiples of the
41 resolution of the underlying clock shall be rounded up to the next larger multiple of
42 the resolution. Quantization error shall not cause the timer to expire earlier than
43 the rounded-up time value.

1 If present, `Old_State` shall return the state of the timer prior to setting its new
2 state, the same as would be returned by a call to `Get_Timer_State`.

3 The function `Get_Timer_State` shall return the current state of the timer specified
4 by `Timer`. The `Initial` attribute of the return value shall return a `Timespec` value
5 representing either zero, if the specified timer is not armed, or the difference between
6 the next expiration time of `Timer` and the current time of the underlying clock, if the
7 timer is armed. The `Interval` attribute of the return value shall return a `Timespec`
8 value representing either zero, if the specified timer is one-shot, or the interval value
9 previously specified by a call to `Arm_Timer`, if the timer is periodic. The values of the
10 `Initial` and `Interval` attributes are subject to the resolution of the underlying clock.

11 NOTE: The value of the `Initial` attribute returns a relative amount of time until the next
12 timer expiration, even if the timer was armed as an absolute timer.

13 The `Disarm_Timer` procedure shall cancel a previous timer arming operation for
14 the timer specified by `Timer`. The effect of `Disarm_Timer` on pending expiration
15 notifications is unspecified.

16 NOTE: A future revision to this standard is expected to require the following behavior.
17 If the value of the realtime clock is set via `Set_Time`, the new value of the clock shall be
18 used to determine the expiration for any absolute timers based upon the realtime clock.
19 If the absolute time requested at the invocation of such a time service is before the new
20 value of the clock, the time service shall expire immediately in the same way as if the
21 clock had reached the requested time normally.

22 Setting the value of the realtime clock via `Set_Time` shall have no effect on tasks that
23 are blocked waiting for a relative time service based on this clock. Consequently, such
24 time services shall expire when the requested relative interval elapses, independently
25 of the new or old value of the clock.

26 The effect of setting a clock via `Set_Time` on armed timers associated with a clock other
27 than the realtime clock is implemented defined.

28 Only a single signal shall be queued to the process for a given timer at any point in
29 time. When a timer for which a signal is still pending expires, no additional signal
30 shall be queued, and a timer overrun shall occur.

31 When a timer expiration signal is accepted, if the `Realtime Signals` option is supported,
32 the `Get_Timer_Overruns` function shall return the timer expiration overrun count
33 for the timer specified by `Timer`. The overrun count returned shall be the number of
34 extra timer expirations which occurred between the time the timer expiration signal
35 was generated and when it was accepted, up to an implementation defined maxi-
36 mum of `Timer_Overruns_Maximum`. If the number of such extra timer expirations
37 exceeds `Timer_Overruns_Maximum`, then the overrun count shall be set to `Timer_-`
38 `Overruns_Maximum`. The value returned by `Get_Timer_Overruns` shall apply to
39 the most recent expiration signal acceptance for the specified timer. If no expiration
40 signal has been accepted for `Timer`, or the `Realtime Signals` option is not supported,
41 the meaning of the overrun count returned is undefined.

42 **14.1.7.3 Error Handling**

43 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
44 with the corresponding error code:

Invalid_Argument

Timer is not a valid timer identifier.

Upon Arm_Timer, either 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

1 This section describes the facilities for interprocess communication message passing
2 available under this standard.

3 *The functionality described in this section is optional. If the Message Queues option*
4 *is not supported, the implementation may cause all calls to the explicitly declared*
5 *operations defined in this section to raise POSIX_Error. Otherwise, the behavior*
6 *shall be as specified below.*

7 15.1 Package POSIX_Message_Queues

8 This package provides access to services that can be used to create and remove mes-
9 sages queue objects, send messages to a message queue, and receive messages from a
10 message queue.

```
11 with Ada_Streams,  
12     POSIX,  
13     POSIX_IO,  
14     POSIX_Permissions,  
15     POSIX_Signals;  
16 package POSIX_Message_Queues is  
17   -- 15.1.1 Message Queue Attributes  
18   type Message_Queue_Descriptor is private;  
19   type Attributes is private;  
20   type Message_Queue_Options is new POSIX.Option_Set;  
21   Non_Blocking: constant Message_Queue_Options  
22     := Message_Queue_Options(POSIX_IO.Non_Blocking);  
23   subtype Message_Priority is Integer range implementation-defined;  
24   procedure Set_Max_Messages  
25     (Attrs: in out Attributes;  
26      Value: Natural);  
27   function Get_Max_Messages  
28     (Attrs: Attributes) return Natural;  
29   procedure Set_Message_Length  
30     (Attrs: in out Attributes;  
31      Value: Natural);  
32   function Get_Message_Length  
33     (Attrs: Attributes) return Natural;  
34   procedure Set_Options  
35     (Attrs: in out Attributes;  
36      Value: Message_Queue_Options);  
37   function Get_Options  
38     (Attrs: Attributes)  
39     return Message_Queue_Options;  
40   function Get_Message_Count  
41     (Attrs: Attributes) return Natural;  
42   -- 15.1.2 Open a Message Queue  
43   function Open  
44     (Name:          POSIX.POSIX_String;  
45      Mode:          POSIX_IO.File_Mode;  
46      Options:       POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;  
47      Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)  
48     return Message_Queue_Descriptor;
```

```

1      function Open_Or_Create
2      (Name:          POSIX.POSIX_String;
3       Mode:          POSIX_IO.File_Mode;
4       Permissions:   POSIX_Permissions.Permission_Set;
5       Options:       POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
6       Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
7      return Message_Queue_Descriptor;
8      function Open_Or_Create
9      (Name:          POSIX.POSIX_String;
10     Mode:          POSIX_IO.File_Mode;
11     Permissions:   POSIX_Permissions.Permission_Set;
12     Options:       POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;
13     Attrs:         Attributes;
14     Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)
15     return Message_Queue_Descriptor;
16     -- 15.1.3 Close a Message Queue
17     procedure Close (MQ: in out Message_Queue_Descriptor);
18     -- 15.1.4 Remove a Message Queue
19     procedure Unlink_Message_Queue (Name: in POSIX.POSIX_String);
20     -- 15.1.5 Send a Message to a Message Queue
21     procedure Send
22     (MQ:          in Message_Queue_Descriptor;
23     Message:      in Ada_Streams.Stream_Element_Array;
24     Priority:      in Message_Priority;
25     Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
26     -- 15.1.6 Receive a Message from a Message Queue
27     procedure Receive
28     (MQ:          in Message_Queue_Descriptor;
29     Message:      out Ada_Streams.Stream_Element_Array;
30     Last:         out Ada_Streams.Stream_Element_Offset;
31     Priority:      out Message_Priority;
32     Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
33     -- 15.1.7 Generic Message Passing
34     generic
35     type Message_Type is private;
36     package Generic_Message_Queues is
37     procedure Send
38     (MQ:          in Message_Queue_Descriptor;
39     Message:      in Message_Type;
40     Priority:      in Message_Priority;
41     Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
42     procedure Receive
43     (MQ:          in Message_Queue_Descriptor;
44     Message:      out Message_Type;
45     Priority:      out Message_Priority;
46     Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
47     function Get_Error_Buffer return Ada_Streams.Stream_Element_Array;
48     end Generic_Message_Queues;
49     -- 15.1.8 Notify Process that a Message is Available
50     procedure Request_Notify
51     (MQ:  in Message_Queue_Descriptor;
52     Event: in POSIX_Signals.Signal_Event);
53     procedure Remove_Notify (MQ: in Message_Queue_Descriptor);

```

```

1      -- 15.1.9 Set/Query Message Queue Attributes
2      procedure Set_Attributes
3          (MQ:      in Message_Queue_Descriptor;
4            New_Attrs: in Attributes;
5            Old_Attrs: out Attributes);
6      procedure Set_Attributes
7          (MQ:      in Message_Queue_Descriptor;
8            New_Attrs: in Attributes);
9      function Get_Attributes
10         (MQ:      Message_Queue_Descriptor)
11         return Attributes;
12 private
13     implementation-defined
14 end POSIX_Message_Queues;

```

15.1.1 Message Queue Attributes

15.1.1.1 Synopsis

```

17      type Message_Queue_Descriptor is private;
18      type Attributes is private;
19      type Message_Queue_Options is new POSIX.Option_Set;
20      Non_Blocking: constant Message_Queue_Options
21          := Message_Queue_Options(POSIX_IO.Non_Blocking);
22      subtype Message_Priority is Integer range implementation-defined;
23      procedure Set_Max_Messages
24          (Attrs: in out Attributes;
25           Value: Natural);
26      function Get_Max_Messages
27          (Attrs: Attributes) return Natural;
28      procedure Set_Message_Length
29          (Attrs: in out Attributes;
30           Value: Natural);
31      function Get_Message_Length
32          (Attrs: Attributes) return Natural;
33      procedure Set_Options
34          (Attrs: in out Attributes;
35           Value: Message_Queue_Options);
36      function Get_Options
37          (Attrs: Attributes)
38          return Message_Queue_Options;
39      function Get_Message_Count
40          (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).

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:

- 1 **Max Messages**
2 The number of messages that can be held in the message queue, without
3 causing `Send` or `Generic_Message_Queue.Send` to fail or wait due to
4 lack of resources. This attribute may be queried, but can only be set at the
5 time the message queue is created.
- 6 **Message Length**
7 The maximum size in bytes (not bits) of each message in the message queue.
8 This attribute may be queried, but can only be set at the time the message
9 queue is created.
- 10 **Options**
11 Options specific to message queues which affect message queue operations
12 and the state of the message queue to which they are applied. This attribute
13 may be both queried and set (See §15.1.9).
- 14 **Message Count**
15 The number of messages currently on the queue. This attribute may be
16 queried, but cannot be set.
- 17 Implementations may add other attributes, as permitted in §1.3.1.1, point (2).
18 Adding attributes which may change the behavior of applications with respect to this
19 standard when those attributes are uninitialized, also requires that the extension be
20 enabled, as required by §1.3.1.1.
- 21 A function whose name is of the form `Get_X`, where `X` is the name of one of the at-
22 tributes of a message queue, shall return the value of that attribute for the specified
23 **Attributes** object.
- 24 A procedure whose name is of the form `Set_X`, where `X` is the name of one of the
25 attributes of a message queue, shall set the value of that attribute for the specified
26 **Attributes** object.
- 27 NOTE: The operation to set the Message Count attribute of an **Attributes** object is
28 intentionally omitted.
- 29 Values of type `Message_Queue_Options` are used to specify the **Options** attribute
30 of an **Attributes** object. The following values of `Message_Queue_Options` shall
31 be defined :
- 32 **Non_Blocking**
33 Calls to `Send`, `Generic_Message_Queue.Send`, `Receive`, and `Generic_`
34 `Message_Queue.Receive` shall fail if a message queue resource is tem-
35 porarily unavailable (such as if the message queue is full on a `Send`). If
36 **Non_Blocking** is not specified, calls to these operations shall wait until the
37 resource becomes available.
38 The constant `POSIX_Message_Queue.Non_Blocking` here is a homo-
39 graph of the constant `POSIX_IO.Non_Blocking` and must have the same
40 representation, though it is of a different type.
- 41 The interpretation of other values of `Message_Queue_Options` is unspecified.
- 42 The subtype `Message_Priority` defines the range of message priority levels sup-
43 ported by the implementation.

1 NOTE: Message_Priority'Last may be greater than Message Priority Maximum.

2 NOTE: Priority is an attribute of a message, not an attribute of a message queue.

3 **15.1.1.3 Error Handling**

4 If any of the following conditions occur, the exception POSIX_Error shall be raised,
5 with the corresponding error code:

6 Operation_Not_Implemented

7 The requested message queue operation is not supported by this implemen-
8 tation.

9 **15.1.2 Open a Message Queue**

10 **15.1.2.1 Synopsis**

```
11   function Open  
12       (Name:            POSIX.POSIX_String;  
13        Mode:            POSIX_IO.File_Mode;  
14        Options:         POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;  
15        Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)  
16       return Message_Queue_Descriptor;  
17   function Open_Or_Create  
18       (Name:            POSIX.POSIX_String;  
19        Mode:            POSIX_IO.File_Mode;  
20        Permissions:     POSIX.Permissions.Permission_Set;  
21        Options:         POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;  
22        Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)  
23       return Message_Queue_Descriptor;  
24   function Open_Or_Create  
25       (Name:            POSIX.POSIX_String;  
26        Mode:            POSIX_IO.File_Mode;  
27        Permissions:     POSIX.Permissions.Permission_Set;  
28        Options:         POSIX_IO.Open_Option_Set := POSIX_IO.Empty_Set;  
29        Attrs:            Attributes;  
30        Masked_Signals: POSIX.Signal_Masking:= POSIX.RTS_Signals)  
31       return Message_Queue_Descriptor;
```

32 **15.1.2.2 Description**

33 The Open and Open_Or_Create functions establish the connection between a process
34 and a message queue via a message queue descriptor. They create an open message
35 queue description that refers to that message queue, and return a message queue
36 descriptor that refers to the open message queue description. The open message
37 queue description contains information about the mode in which the process accesses
38 the message queue, as well as a reference to the message queue itself. The message
39 queue descriptor is used by other operations to refer to the open message queue
40 description.

41 The Open function requires that the message queue exist at the time of the call.
42 Open_Or_Create will create the message queue if it does not exist. The Name pa-
43 rameter 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_Queue.Receive` but not `Send` or `Generic_Message_Queue.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_Queue.Send` but not `Receive` or `Generic_Message_Queue.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 message queue's user ID shall be set to the process's effective user ID, and the message queue's group ID shall be set to the process's effective group ID.
- 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`.

- 1 NOTE: The blocking behavior associated with the new open message queue de-
 2 scription is taken from the `Options` parameter. The `Options` attribute of the `Attrs`
 3 parameter is ignored by `Open_Or_Create`.
- 4 — If the optional parameter `Attrs` is specified, and the calling process does not
 5 have the appropriate privilege on `Name`, the `Open_Or_Create` function shall fail
 6 without creating the message queue.
 - 7 — If the `Attrs` parameter is not specified, the message queue shall be created with
 8 implementation defined default message queue attributes.

9 The parameter `Options` shall specify additional options for the message queue. The
 10 meaning of the options is as follows :

11 Exclusive

12 `Open_Or_Create` shall fail if the message queue `Name` exists. The check for
 13 the existence of the message queue and the creation of the message queue
 14 if it does not exist shall be atomic with respect to other processes executing
 15 `Open_Or_Create` naming the same `Name` with `Exclusive` set. If this option
 16 is specified for `Open`, the result is undefined.

17 Non_Blocking

18 This option is associated with the open message queue description and de-
 19 termines whether calls to `Send`, `Generic_Message_Queue.Send`, `Re-`
 20 ceive, and `Generic_Message_Queue.Receive` shall fail if a message
 21 queue resource is temporarily unavailable (such as if the message queue is
 22 full on a `Send`). If `Non_Blocking` is not specified in `Options`, calls to these
 23 operations shall wait until the resource becomes available.

24 The constant `POSIX_IO.Non_Blocking` here is a homograph of the con-
 25 stant `POSIX_Message_Queue.Non_Blocking`, and shall have the same
 26 representation, though it is of a different type.

27 The effect of setting other values from `POSIX_IO.Open_Option_Set` in the `Options`
 28 parameter of `Open` or `Open_Or_Create` is undefined.

29 `Open` and `Open_Or_Create` may be interruptible by the delivery of a signal.
 30 `Masked_Signals` specifies the set of signals that shall be added to the signal mask
 31 for the duration of this operation, as described in §3.3.6.

32 The `Open` and `Open_Or_Create` functions shall not add or remove messages from
 33 the message queue.

34 15.1.2.3 Error Handling

35 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
 36 with the corresponding error code:

37 Permission_Denied

38 The message queue exists and the permissions specified by `Mode` are denied,
 39 or the message queue does not exist and permission to create the message
 40 queue is denied.

41 File_Exists

42 The `Exclusive` option is specified on a call to `Open_Or_Create` and the
 43 named message queue already exists.

- 1 **Interrupted_Operation**
2 A signal interrupted the `Open` or `Open_Or_Create` operation.
- 3 **Invalid_Argument**
4 The `Open` or `Open_Or_Create` operation is not supported for the given
5 name. The implementation shall document under what circumstances this
6 error may be returned.
7 The `Max Messages` or `Message Length` attribute of `Attrs` is less than or equal
8 to zero.
- 9 **Too_Many_Open_Files**
10 Too many message queue descriptors or file descriptors are currently in use
11 by this process.
- 12 **Filename_Too_Long**
13 The length in POSIX characters of the specified pathname exceeds `Pathname`
14 Limit or the length in POSIX characters of a component of the specified path-
15 name is greater than `Filename Maximum` and the `Filename Truncation` option is
16 not supported for the pathname prefix of that component.
- 17 **Too_Many_Open_Files_In_System**
18 Too many message queues are currently open in the system.
- 19 **No_Such_File_Or_Directory**
20 The `Open` operation was called and the named message queue does not exist.
- 21 **No_Space_Left_On_Device**
22 There is insufficient space for the creation of the new message queue.
- 23 **Operation_Not_Implemented**
24 The `Open` and `Open_Or_Create` operations are not supported by this im-
25 plementation.

26 **15.1.3 Close a Message Queue**

27 **15.1.3.1 Synopsis**

28 **procedure** `Close` (`MQ`: **in out** `Message_Queue_Descriptor`);

29 **15.1.3.2 Description**

30 The `Close` procedure shall remove the association between the message queue de-
31 scriptor `MQ` and its associated open message queue description. When all message
32 queue descriptors associated with an open message queue description are closed, the
33 open message queue description shall be freed. The results of using the message
34 queue descriptor value after a successful return from `Close` until the return of this
35 message queue descriptor value from a subsequent call to `Open` or `Open_Or_Create`
36 are undefined.

37 If the process has successfully attached a notification request to the message queue,
38 this attachment shall be removed and the message queue made available for another
39 process to attach for notification.

15.1.3.3 Error Handling

If any of the following conditions occur, 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 re-create 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 occur, 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.

`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, then 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 occur, 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.

- 1 **Bad_File_Descriptor**
- 2 The MQ argument is not a valid message queue descriptor open for writing.
- 3 **Interrupted_Operation**
- 4 A signal interrupted the call to Send.
- 5 **Invalid_Argument**
- 6 The value of Priority is outside of the valid range.
- 7 **Message_Too_Long**
- 8 Message'Length is greater than the Message Length attribute of the queue.
- 9 **Operation_Not_Implemented**
- 10 The Send operation is not supported by this implementation.

11 **15.1.6 Receive a Message from a Message Queue**

12 **15.1.6.1 Synopsis**

```
13     procedure Receive
14         (MQ:                 in   Message_Queue_Descriptor;
15          Message:           out   Ada_Streams.Stream_Element_Array;
16          Last:              out   Ada_Streams.Stream_Element_Offset;
17          Priority:           out   Message_Priority;
18          Masked_Signals: in   POSIX.Signal_Masking := POSIX.RTS_Signals);
```

19 **15.1.6.2 Description**

20 The **Receive** procedure removes the oldest of the highest priority messages from the
21 message queue specified by MQ. The message is stored in the **Message** parameter of
22 type **Ada_Streams.Stream_Element_Array**. The call shall fail if **Message'Length**
23 is less than the **Message Length** attribute of the message queue. Otherwise, the mes-
24 sage shall be removed from the queue and placed into **Message**, starting at the first
25 element of **Message**, and **Last** shall be set to the index of the last byte of the mes-
26 sage.

27 **Priority** shall be assigned the priority of the message removed from the message
28 queue.

29 If the specified message queue is empty and **Non_Blocking** is not specified in the
30 message queue description associated with MQ, **Receive** shall block until a message
31 is enqueued on the message queue or until interrupted by a signal. If the **Process**
32 **Scheduling** option is supported, then the task to be unblocked shall be selected in
33 a manner appropriate to the scheduling policies and parameters in effect for the
34 blocked tasks.

35 If the **Priority Process Scheduling** option is not supported, the selection of which task
36 receives the message is unspecified.

37 If the specified message queue is empty and **Non_Blocking** is specified in the open
38 message queue description associated with MQ, **Receive** shall fail.

39 **Receive** may be interruptible by the delivery of a signal. **Masked_Signals** specifies
40 the set of signals that shall be added to the signal mask for the duration of this
41 operation, as described in §3.3.6.

1 15.1.6.3 Error Handling

2 If any of the following conditions occur, the exception `POSIX_Error` shall be raised,
 3 with the corresponding error code:

4 `Resource_Temporarily_Unavailable`

5 `Non_Blocking` is specified in the open message queue description associ-
 6 ated with MQ and the specified message queue is empty.

7 `Bad_File_Descriptor`

8 The MQ argument is not a valid message queue descriptor open for reading.

9 `Message_Too_Long`

10 `Message'Length` is less than the `Message Length` attribute of the message
 11 queue.

12 `Interrupted_Operation`

13 A signal interrupted the call to `Receive`.

14 `Operation_Not_Implemented`

15 The `Receive` operation is not supported by this implementation.

16 If any of the following conditions is detected, the exception `POSIX_Error` shall be
 17 raised, with the corresponding error code:

18 `Bad_Message`

19 The implementation has detected a data corruption problem with the mes-
 20 sage.

21 15.1.7 Generic Message Passing

22 15.1.7.1 Synopsis

```

23 generic
24   type Message_Type is private;
25   package Generic_Message_Queues is
26     procedure Send
27       (MQ:           in Message_Queue_Descriptor;
28        Message:      in Message_Type;
29        Priority:      in Message_Priority;
30        Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
31     procedure Receive
32       (MQ:           in Message_Queue_Descriptor;
33        Message:      out Message_Type;
34        Priority:      out Message_Priority;
35        Masked_Signals: in POSIX.Signal_Masking:= POSIX.RTS_Signals);
36     function Get_Error_Buffer return Ada.Streams.Stream_Element_Array;
37   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 only specified 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: It is the application's responsibility when using the operations `Send` and `Receive` from an instantiation of `Generic_Message_Queues` to ensure that (1) 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` and (2) 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 occur, 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.

There is no raw data available on a call to `Get_Error_Buffer`. Either `Constraint_Error` has not been raised from a call to `Receive` or some 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`.

- 1 **Invalid_Argument**
 2 The value of **Priority** specified for the **Send** operation is outside of the
 3 valid range.
- 4 **Message_Too_Long**
 5 The size of **Message** in bytes is greater than the **Message Length** attribute of
 6 the queue on a **Send** or less than the **Message Length** attribute of the queue
 7 on a **Receive**.
- 8 **Operation_Not_Implemented**
 9 The **Send**, **Receive** and **Get_Error_Buffer** operations are not supported
 10 by this implementation.

11 If any of the following conditions is detected, the exception **POSIX_Error** shall be
 12 raised, with the corresponding error code:

- 13 **Bad_Message**
 14 The implementation has detected a data corruption problem with the mes-
 15 sage on a **Receive**.

16 In addition to the error conditions specified above, **Constraint_Error** may be
 17 raised by a call to **Receive** if the message received cannot be interpreted as a value
 18 of the specified type. In cases where the message was removed from the queue but
 19 the message was not returned due to **Constraint_Error** having been raised, **Get_-**
 20 **Error_Buffer** can be used to gain access to the raw message data for debugging and
 21 error logging purposes.

22 **15.1.8 Notify Process that a Message is Available**

23 **15.1.8.1 Synopsis**

24 **procedure** Request_Notify
 25 (MQ: **in** Message_Queue_Descriptor;
 26 Event: **in** POSIX_Signals.Signal_Event);
 27 **procedure** Remove_Notify (MQ: **in** Message_Queue_Descriptor);

28 **15.1.8.2 Description**

29 *The functionality described in this subclause is optional. If the Realtime Signals option*
 30 *is not supported, the implementation may cause all calls to the explicitly declared*
 31 *operations defined in this subclause to raise **POSIX_Error**. Otherwise, the behavior*
 32 *shall be as specified below.*

33 **Request_Notify** registers the calling process to be notified of message arrival at an
 34 empty message queue specified by **MQ**. The notification specified by **Event** shall be
 35 sent to the process when the message queue transitions to non-empty. At any time,
 36 only one process may be registered to receive notification for the specified message
 37 queue. If the calling process or any other process has already registered for noti-
 38 fication of message arrival at the specified message queue, subsequent attempts to
 39 register for that message queue shall fail.

The `Event` parameter identifies a `Signal_Event` object which defines the asynchronous notification that will occur when the message queue transitions to non-empty. If the `Notification` attribute of `Event` is `No_Notification`, no notification shall be sent when the message queue transitions to non-empty. 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 non-empty. 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 §2.2.2.117.)

`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 some 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 occur, 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);
```

```
1      procedure Set_Attributes
2      (MQ:      in Message_Queue_Descriptor;
3       New_Attrs: in Attributes);
4      function Get_Attributes
5      (MQ:      Message_Queue_Descriptor)
6      return Attributes;
```

7 **15.1.9.2 Description**

8 Set_Attributes is used to set the attributes associated with the open message
9 queue description referenced by the message queue descriptor specified by the MQ
10 parameter. Set_Attributes is overloaded to allow the previous attributes of the
11 message queue to be returned.

12 Only the Options attribute of the New_Attrs parameter is relevant to the Set_At-
13 tributes call. The values of the other attributes are ignored by Set_Attributes.

14 Get_Attributes is used to get status information and attributes of the message
15 queue and the open message queue description associated with the message queue
16 descriptor specified by MQ. The Max Messages and Message Length attributes of the
17 returned Attributes object shall have the values that were set when the message
18 queue was created. The Options attribute of the returned Attributes object shall
19 have the value associated with the open message queue description that was set
20 when the message queue was created or opened, and as modified by subsequent calls
21 to Set_Attributes. The Message Count attribute of the returned Attributes object
22 shall have a value that reflects the current state of the specified message queue.

23 **15.1.9.3 Error Handling**

24 If any of the following conditions occur, the exception POSIX_Error shall be raised,
25 with the corresponding error code:

26 **Bad_File_Descriptor**

27 The MQ argument is not a valid message queue descriptor.

28 **Operation_Not_Implemented**

29 The requested message queue operation is not supported by this implemen-
30 tation.

Section 16: Task Management

1 This section describes the services which this standard provides for task identifica-
2 tion.

3 **16.1 Package Ada_Task_Identification**

4 **package** Ada_Task_Identification **renames** Ada.Task_Identification;

5 During transition from Ada83 (ISO/IEC 8652: 1987) to Ada95 (ISO/IEC 8652: 1995)
6 the implementation is permitted to replace this renaming declaration with the fol-
7 lowing package specification.

```
8 package Ada_Task_Identification is  
9   -- Task Identification Type  
10  type Task_ID is private;  
11  Null_Task_ID: constant Task_ID;  
12  function Equals      (Left, Right: Task_ID) return Boolean;  
13  function Image       (T: Task_ID) return String;  
14  -- Identifying the Current Task  
15  function Current_Task return Task_ID;  
16  -- Aborting a Task  
17  procedure Abort_Task  (T: in out Task_ID);  
18  -- Inquiring the State of a Task  
19  function Is_Terminated (T: Task_ID) return Boolean;  
20  function Is_Callable  (T: Task_ID) return Boolean;  
21 private  
22   -- implementation-defined;  
23 end Ada_Task_Identification;
```

24 **16.1.1 Description**

25 The semantics of the types and operations defined in this package shall satisfy the
26 requirements of Ada95 (ISO/IEC 8652: 1995).

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Annex A (informative) Bibliography

1 The following documents are related to this standard in an informative manner, or
2 were used as references in its preparation.

3 {B1} Adams, Douglas, *The Hitchhikers Guide to the Galaxy*. New York: Crown
4 Books, 1989.

5 {B2} Blair, Byron E. (Ed.), *Time and Frequency: Theory and Fundamentals*. U.S.
6 Dept. of Commerce, National Bureau of Standards, May 1974.

7 {B3} *IEEE Std 1003.5-1992, IEEE Standard for Information Technology— POSIX*
8 *Ada Language Interfaces— Part 1: Binding for System API*.

9 {B4} ISO/IEC 8859-1: 1998¹, *Information technology— 8-bit single-byte coded*
10 *graphic character sets— Part 1: Latin alphabet No. 1*.

11 {B5} ISO/IEC JTC 1/SC22/WG9/Ada Rapporteur Group, *GET_LINE for interactive*
12 *devices*, Ada Issue AI-00172, August 1986.²

13 {B6} ISO/IEC JTC 1/SC22/WG9/Ada Rapporteur Group, *Data can be appended to*
14 *the end of an existing file*, Ada Issue AI-00278, June 1989.

15 {B7} ISO/IEC JTC 1/SC22/WG9/Ada Rapporteur Group, *NEW_PAGE can raise*
16 *USE_ERROR for certain FORMs of File*, Ada Issue AI-00886, January 1991.

17 {B8} McJones, Paul R. and Swart, Garret F., “Evolving the UNIX System Interface
18 to Support Multithreaded Programs,” Digital Equipment Corp., Palo Alto, CA,
19 DEC-SRC Report 21, September 1987.

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

22 ²Ada Issues can be obtained from the Ada Information Clearinghouse, c/o IIT Research Institute,
23 4500 Forbes Boulevard, Lanham, MD 20706-4312, USA.

This side is intentionally blank.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14519:1999

Annex B (informative) **Rationale and Notes**

B.1 General

This is the rationale 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 base standards: POSIX.1 {2}, POSIX.1b {3}, POSIX.1c {4}, and POSIX.1i. See those standards for that rationale.

In what follows, it is sometimes useful to distinguish between the original version of this standard, denoted POSIX.5 {B3}, and the realtime revision of this standard, denoted POSIX.5b {3}, which is a binding to POSIX.1b {3} and POSIX.1c {4}.

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 will 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 {3} revision, the rationale has been 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 POSIX.1 {2}, as amended by POSIX.1b {3} and POSIX.1c {4}, in the sense that the semantics defined in those documents 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.

This standard does *not* require that the system support any of the C-language interface defined in the base standards: POSIX.1 {2}, POSIX.1b {3}, and POSIX.1c {4}. (See §B.1.4.) Note, in particular, that the system is not required to support the POSIX.1c {4} C-language threads services. Ada already has multiple threads of control

1 within a process, and POSIX.5 {B3} already defines the interactions of tasks with
2 the facilities of POSIX.1 {2}. Support for tasks is required of all implementations of
3 the Ada language, and therefore is required for all implementations of this standard.
4 The developers of this standard have expended great effort to make it transparent
5 to the Ada application whether or not POSIX.1c {4} is supported by the underlying
6 system.

7 This standard is a binding to POSIX.1c {4}, in the sense that several changes have
8 been made from POSIX.5 {B3} to achieve semantic compatibility with an implemen-
9 tation in which both Ada tasks and POSIX.1c {4} threads map to the same type of
10 entity in the underlying operating system. Moreover, some new interfaces are pro-
11 vided for services defined by POSIX.1c {4} that are not defined for Ada tasks by
12 Ada RM {1}, but which should be implementable by an Ada runtime system, regard-
13 less of whether the system supports POSIX.1c {4}.

14 **B.1.3 Ada Language**

15 The original version of this standard, POSIX.5 {B3}, defined a binding to Ada83
16 (ISO/IEC 8652: 1987), since that was the recognized standard for the Ada language
17 at the time the binding was developed. Concurrent with the revision POSIX.5b {3},
18 the Ada language standard was also being revised. It was decided to retain compat-
19 ibility in the binding with Ada83 (ISO/IEC 8652: 1987), but also to support migra-
20 tion to Ada95 (ISO/IEC 8652: 1995). This decision was taken because production
21 implementations of Ada95 (ISO/IEC 8652: 1995) were not expected to be available
22 immediately, and there was an immediate demand for the revised POSIX Ada bind-
23 ing. Moreover, implementations of Ada83 (ISO/IEC 8652: 1987) were expected to
24 continue to be used for maintenance of existing applications.

25 There are some cases where new functionality of Ada95 (ISO/IEC 8652: 1995) con-
26 flicts with or overlaps functionality defined by POSIX.1c {4}. Such cases have been
27 resolved on a case-by-case basis, taking into account the degree of implementation
28 freedom given by the Ada language. The most extreme case is the model for thread
29 cancelation, which is too restrictive to support the new Ada semantics for abortion,
30 exception propagation, and object finalization. In this case, it was decided to regard
31 the POSIX.1c {4} feature as specific to the C language binding, and therefore to pro-
32 vide no Ada binding for it. In some other cases, such as thread scheduling, the Ada
33 language provides enough flexibility that the POSIX.1c {4} model can be provided as
34 a user-selectable alternative to the standard Ada facility, with the implementation
35 being left to define any interaction between the two.

36 There are some cases where new Ada95 (ISO/IEC 8652: 1995) features are needed to
37 make effective use of the POSIX interface. A good example is the extended support
38 for the type `Address` in package `System` and its new child package `System.Storage_Elements`.
39 In this case, to provide the capability during transition from Ada83
40 (ISO/IEC 8652: 1987) to Ada95 (ISO/IEC 8652: 1995), this standard defines the
41 package `System.Storage_Elements` to be a renaming of `System.Storage_Elements`,
42 and gives the implementation explicit permission to replace the renaming
43 with a package specification and body with similar functionality, as close as possible
44 to the Ada95 (ISO/IEC 8652: 1995) package, without depending on any new language
45 features.

1 **B.1.4 Implementation Model**

2 The POSIX Ada binding is intended to be implementable on a variety of systems,
3 in a variety of ways. In particular, the binding has been developed to allow for the
4 following implementation possibilities:

- 5 — An implementation on a bare machine, with no underlying operating system
6 support (e.g., corresponding to the Minimal Realtime Profile of POSIX.13).
- 7 — An implementation based on an underlying operating system conforming to PO-
8 SIX.1 {2}, as amended by POSIX.1b {3}, but not supporting POSIX.1c {4}.
- 9 — An implementation based on an underlying operating system conforming to PO-
10 SIX.1 {2} as amended by POSIX.1b {3} and POSIX.1c {4}.

11 The only means by which the application might distinguish between these possibil-
12 ities is the support or non-support for particular options. For example, blocking be-
13 havior might not be per-task in an implementation based on an underlying operating
14 system that does not support POSIX.1c {4}. Thus, Strictly Conforming applications
15 should be portable across all implementations.

16 **B.1.5 Level of Binding**

17 The developers of POSIX.5 {B3} faced a fundamental choice between a “direct” *ver-*
18 *sus* an “abstract” binding of the POSIX.1 {2} functions into Ada. A direct binding
19 maps the POSIX C functions as closely as possible to the corresponding Ada facil-
20 ities. Arguments in favor of a direct binding are that it is less expensive in time
21 and effort to produce and the implementation is more likely to be able to call the
22 underlying POSIX.1 {2} functions directly.

23 An abstract binding tries to group POSIX functions into abstract data types, oper-
24 ations on those types, and packages of logically related operations. Arguments in
25 favor of an abstract binding are that it allows a more natural style and bindings may
26 be omitted more easily for POSIX functions that duplicate already existing features
27 of Ada.

28 The abstract binding approach was chosen for POSIX.5 {B3}. Examples of ways in
29 which the Ada binding differs from a direct translation of C into Ada include:

- 30 — Using packages to organize declarations that are logically related.
- 31 — Using generics to generalize certain C functions.
- 32 — Using more expressive Ada names.
- 33 — Splitting multipurpose C functions into several Ada subprograms.
- 34 — Mapping C *errno* codes to the Ada exception mechanism.
- 35 — Omitting POSIX C functions that duplicate existing Ada language features.

36 The model is that there is a basic abstraction, consisting of the set of POSIX types,
37 values, and operations. The POSIX.1 {2} binding is one expression of the POSIX
38 abstraction in the C language. This standard represents another expression of the
39 same POSIX abstraction, in the Ada language. The approach used to develop the
40 Ada binding was to uncover the basic abstraction and then to express it as well as
41 possible in Ada.

1 The POSIX.5b {3} revision did not seriously reconsider this question. To do so would
2 have required a total rewrite of POSIX.5 {B3}. Besides, input from the balloting
3 group seemed to favor continuation of the abstract binding approach.

4 **B.1.6 Form of Document**

5 A related, but separate issue is the form of this document. Historically, POSIX stan-
6 dards are first expressed in the C language. Bindings of these standards to other
7 languages then face a choice between a “thin” document and a “thick” document. A
8 thin binding document is one that defines only the syntax of the operations in the
9 new language. Readers of the language binding are referred to the base standards
10 for an explanation of the required behavior. Arguments in favor of a thin binding
11 document are that it less expensive to produce and maintain and it is less likely to
12 introduce inadvertent discrepancies between the base standards and the language
13 binding.

14 A thick binding document, on the other hand, is self-contained. The required be-
15 havior of each interface in the new language is fully described, so that readers need
16 not refer to the base standards. Arguments in favor of a thick document are that
17 users can find all the needed information in one place and it is likely to be more
18 understandable.

19 The original version of this standard is a thick binding document. The required
20 behavior of the POSIX Ada interface is fully documented; users need not refer to
21 POSIX.1 {2}, and therefore need not be conversant with C.

22 At the time work started on POSIX.5b {3}, the IEEE Portable Applications Stan-
23 dards Committee (PASC) officially adopted the language-independent standards
24 paradigm promoted by ISO. It was believed that the development and maintenance of
25 multilingual interfaces for POSIX would be easier if there were a common language-
26 independent specification (LIS) of POSIX services. Language bindings then would be
27 thin specifications of syntax and would refer back to the LIS for the required behav-
28 ior of the interfaces. Considerable effort was expended within POSIX to make this
29 LIS paradigm work, but several difficulties were encountered.

30 First, the POSIX application program interfaces are all intended to comprise one
31 standard, but were being developed piecemeal. It was difficult to synchronize the
32 development of LIS documents with the development of extensions to C language
33 standards. Several extensions of POSIX.1 {2} were underway as C language bindings;
34 some of these were already in ballot. Conversion to thin bindings and LIS would
35 mean essentially going back and starting over.

36 Second, even for new work, it was difficult to find volunteers willing and able to cre-
37 ate LIS documents. Most of the POSIX work is based on established practice in the
38 C/UNIX market, is done by individuals who are experienced in C, and is funded by
39 organizations who regard C as the predominate language in their markets. Finding
40 people willing to develop LIS documents is much harder. The notation is unfamiliar,
41 and the connection to practice is tenuous. Thus, progress on LIS standards has been
42 very slow.

43 A final problem is end-user hostility to the LIS model. The combination of LIS and
44 thin language binding is much harder to read and understand than a single self-

1 contained document. To understand the behavior required by such a standard, the
2 user must master both the LIS notation and the specific programming language,
3 and mentally keep track of the correspondence of names and concepts between the
4 two documents— using one set of names for syntax and another set of names for
5 semantics.

6 The first balloted draft of POSIX.5b {3} was the result of a good-faith effort to develop
7 a thin Ada binding to an LIS. Since there was no LIS for POSIX.1b {3} or POSIX.1c-
8 {4}, the C- language interfaces were used in place of the LIS. The response to the
9 first ballot of POSIX.5b {3} was overwhelmingly against changing over to the thin
10 binding format. This coincided with a realization within IEEE PASC that the LIS
11 paradigm was an impediment to the development of needed standards, and so the
12 LIS requirement was deleted. Subsequent drafts of POSIX.5b {3} have converted to
13 the thick format consistent with POSIX.5 {B3}.

14 **B.1.7 Global Issues**

15 **B.1.7.1 Document Structure**

16 The document structure of the Ada binding closely follows that of the C-language
17 binding for the base standards. Each section consists of one or a few Ada package
18 specifications, containing declarations for logically related types, constants, and op-
19 erations. A few operations have been moved from the section where they appear in
20 the C interface to a different section in the Ada binding to achieve Ada packaging
21 consistency; these differences date back to POSIX.5 {B3}. To aid users and imple-
22 mentors in relating Ada binding entities to the C interface entities, annex C provides
23 a C-to-Ada cross-reference.

24 **B.1.7.2 Packaging**

25 The developers of POSIX.5 {B3} wanted to produce an object-oriented packaging
26 for POSIX, but could not because of the overriding decision to remain reasonably
27 consistent with the organization of the base standards.

28 Four packaging styles received consideration:

- 29 — Single package: specify the entire binding within a single package with no sub-
30 packaging.
- 31 — Multiple packages: create several separate packages, each centered around a
32 small set of logically related types and operations.
- 33 — Subpackaged: the same as “multiple packages,” except that the entire binding is
34 wrapped by a single outer package.
- 35 — Limited subpackaged: place nearly all of the binding in a single package, but
36 place a few selected portions (e.g., “unsafe” operations) in subpackages.

37 After considering the arguments for and against each alternative, the multiple pack-
38 ages approach was selected, primarily because it could be converted easily into one
39 of the other styles, should that become desirable.

40 Arguments in favor of multiple packages include: (a) clear identification of interface
41 usage, by means of examining the **with** clauses of an application; and (b) 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: (a) complicated visibility rules; (b) large and segmented name space, so that it is not obvious where types, objects, and operations are located unless fully dotted notation is used; (c) increased potential for elaboration order and compilation dependency problems; and (d) compartmentalization of implementation details, such that Ada visibility rules require unchecked conversion between private types in different packages that have the same underlying representation.

For POSIX.5b {3}, the availability of child packages in Ada95 (ISO/IEC 8652: 1995) offered another choice. The decision was made to take advantage of this new Ada language feature, but to preserve compatibility with POSIX.5 {B3} and Ada83 (ISO/IEC 8652: 1987). Therefore the policy is that wherever there is a package with a name of the form `POSIX_XXX`, if the implementation supports child packages, `POSIX_XXXX` should be declared as a renaming of `POSIX.XXXX`, a child package of `POSIX`. This has the effect of mitigating some of the disadvantages of the multiple packages approach.

B.1.8 Naming

The POSIX name space is very large. One of the goals of POSIX.5b {3} has been 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 {3} 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, there are some names used in POSIX.5 {B3} that do not follow these rules. They have been retained for compatibility, and so 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, etc). The names of such operations are always `Get_X` and `Set_X`, respectively, where `X` is the name of the attribute.

Any exceptions to consistent naming rules are carried over from POSIX.5 {B3}. In some cases (e.g., `Change_Owner_Restriction` does not conform to the naming rules for options), this was a conscious decision. If there are other cases, yet to be discovered, that do not conform to the rules, it was simple oversight.

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 {B3}. Programs may iterate over program argument lists or environment strings (Section 4), directories (Section 5), or group databases (Section 9). A form of iteration was desired that would be proper Ada style, easy to use, and consistent between 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 it is 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 which indicates the exact nature of the error may be retrieved from *errno*) or they return 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. This allows greater flexibility in

writing exception handlers to treat errors according to the application's needs and requires minimal implementation support over and above what Ada already provides. In particular, this allows the use of a single exception handler for all POSIX errors (including any implementation-defined errors), and the use of 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 function's return value 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 asynchronous I/O operation that might have terminated in error.

B.1.9.2 Extensible Types

The base 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. This is forbidden in §1.3.1.1, because (1) it introduces new visible names (thus possibly breaking applications that have a “**use**” clause for the package), and (2) a simple “**with**”-clause check for uses of implementation-dependent extensions is no longer adequate.

1 There was also concern that an application might depend on the enumeration order,
2 which would not be portable. In order to map enumeration values directly to the
3 representation of the underlying operating system, the ordering of the enumeration
4 type would have to agree with the ordering of the underlying representation (and
5 therefore would be implementation- dependent). A call to an underlying system ser-
6 vice might return a value that cannot be represented in the standard enumeration
7 type (that is, there might be “holes” in the standard values). Checking for this situa-
8 tion, and handling the error in a reasonable way, is a burden on implementors and a
9 performance penalty for users. For all these reasons, enumeration types are seldom
10 used in the POSIX/Ada binding.

11 Integer types are preferred in situations where the values may be used as indices into
12 arrays or for case statements. Also, integer types are convenient for values that must
13 be passed across language boundaries (e.g., from an Ada application to an underlying
14 C-based operating system). Thus, integer types are used for error codes, for signal
15 values, and for file descriptors. To provide some of the convenience of enumeration
16 types for such integer types, value and image functions are provided.

17 This approach has a small risk. For example consider `Error_Code`. The intention
18 is that the implementation-defined range includes all values of the type (both PO-
19 SIX defined and implementation defined), but only the names of the POSIX-defined
20 error codes are visible. This creates the risk of an applications program using an
21 integer value that either represents an implementation extension or a value that is
22 completely invalid (i.e., no meaning either in POSIX or in the implementation.) The
23 application programmer would have to do so using a facility other than a predefined
24 name (e.g., an integer literal). The current approach seems to provide reasonable
25 strong typing and compile-time checking while still permitting extensions by other
26 POSIX standards-development groups.

27 Private types are preferred for situations where no functionality of discrete types is
28 desired. For example, array indexing by `Process_ID` was not thought to be common,
29 so this type was made private. Also, the opaque and structure types in the POSIX C
30 interface are often mapped to private types in the POSIX/Ada binding. Thus, private
31 types are used for option sets and for the time types *timespec* and *itimerspec*.

32 **B.1.9.3 Private Types**

33 C structured types are usually mapped to private types in the POSIX/Ada inter-
34 face, rather than to explicit record types. This allows implementors to add extension
35 fields without changing the visible part of the standard package specification. Use
36 of private types prevents the application from using aggregate notation to represent
37 objects of the private types, and thereby insulates the user from implementation-
38 defined extensions or from implementation-dependent ordering of the components of
39 the structure.

40 The alternative of using explicit record types with record representation clauses to
41 map to the underlying C structures is unlikely to work in any case, since there is
42 no guarantee that the Ada records would have the same layout as the corresponding
43 C structures (even with the use of record representation specifications). Also, most
44 POSIX C structures do not specify the order of the components, and allow additional
45 implementation-defined components. This would require the record type definition

and record representation to be implementation dependent— thus defeating the goal of application portability.

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. This 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 POSIX C declarations 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 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:

- 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*.

1 — The capability of *sleep()* and *nanosleep()* are provided by the **delay** statement.

2 In the case of POSIX.1c {4}, most of the new C functions already have equivalent
3 facilities within Ada. Thus, the only C functions of POSIX.1c {4} which have binding
4 defined by this standard are those that apply to mutexes and condition variables and
5 certain thread scheduling functions.

6 **B.1.9.7 Handles**

7 The POSIX C interface makes extensive use of pointers. In general the POSIX/Ada
8 binding avoids use of Ada access types to correspond to C pointers. One reason is that
9 use of pointers is inherently unsafe; that is, there will always be a chance of dangling
10 references and storage leakage. But, more importantly, it unnecessarily constrains
11 implementations if “handles” are required to be implemented as access types.

12 Access types in Ada imply dynamic (heap) memory allocation. Realtime applications
13 avoid any use of dynamic, if possible, because of the possibility of runtime errors that
14 cannot be detected and removed by prior testing, and because unpredictable response
15 times may occur due to storage recovery being performed at unexpected times.

16 Realtime implementations of POSIX, especially those conforming to the Minimal Re-
17 realtime profile of POSIX.13, may avoid dynamic allocation by preallocating a limited
18 number of system resources in an array data structure. The C pointer would then
19 correspond to an index into the array in the Ada implementation.

20 Other implementations may choose to use dynamic allocation in order to avoid im-
21 posing such hard *a priori* limits on system resources. For these implementations, the
22 C pointer type would indeed correspond to an Ada access type.

23 To allow for both of these implementation choices (array indices and access types) for
24 “handles” to system objects, the Ada binding uses “descriptors” declared as private
25 types.

26 Every occurrence of a C pointer parameter potentially corresponds to two objects in
27 the Ada interface: the “handle” or “descriptor” itself, and the object being referenced
28 by the “handle” or “descriptor”. Sometimes, both must be made visible— as in the
29 case of semaphores. (See Section 7 for the reasons.)

```
30     type Semaphore is limited private;
31     type Semaphore_Descriptor is private;
32     function Descriptor_Of(Sem: Semaphore) return Semaphore_Descriptor;
```

33 If the underlying object is exposed (e.g., Semaphore), it is always an Ada limited
34 private type. The “descriptor” objects are private to allow copying. Given an object
35 (e.g., Semaphore), the function `Descriptor_Of` returns a “handle” that can be used
36 in subsequent operations to refer to the object.

37 At other times, only the “handle” is needed— as in the case of asynchronous I/O con-
38 trol blocks. Then the Ada binding omits the underlying object type declaration and
39 the function `Descriptor_Of`. Thus, for asynchronous I/O control blocks, only the
40 “descriptor” is visible in the interface:

```
41     type AIO_Descriptor is private;
```

42 Occasionally, as in the case of I/O buffers, the proper semantics are achieved by
43 mapping a usage of a C pointer type to an Ada access type, but this is 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, asynchronous I/O 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 the previous subclause.

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 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 base 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. These are enumerated in §1.3.1.1.

B.2 Terminology and General Requirements

B.2.1 Conventions

This clause is similar to the corresponding clause in the base standards, with typographical conventions changed to be appropriate for the Ada language.

B.2.2 Definitions

This clause is similar to the corresponding clause in the base standards. 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 the base standards, with several additions unique to Ada.

B.2.3.1 Process/Active Partition Equivalence

The developers of POSIX.5 {B3} were confronted with a fundamental choice. Which Ada construct corresponds most closely to a POSIX process? Two Ada83 (ISO/IEC 8652: 1987) constructs 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 {B3} was that all Ada tasks within an Ada program *appeared* to execute within a single POSIX process.

Note that 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.

Ada95 (ISO/IEC 8652: 1995) 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 {3} 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 {4} introduces the concept of multi-threaded POSIX processes. The obvious mapping is that an Ada task corresponds to a POSIX thread as defined by POSIX.1c {4}. 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. That is, given the implementation freedoms allowed by the Ada RM {1} and by POSIX, it is not possible to guarantee that Ada tasks having the behavior required and permitted by Ada95 (ISO/IEC 8652: 1995) can be implemented using the facilities for POSIX threads defined by POSIX.1c {4}.

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 lightweight concurrency within a process, and POSIX.1c {4} threads are the C-language expression of the same abstraction. It is clear then that any C threads functions which 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 Ada I/O operations defined in [Ada RM {1}Annex A] 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;
```

This 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. This is 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 to do time-slicing.

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 §2.6.1 and §2.5.1.) Corresponding to each limit and option there 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 there 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). This means that 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, say `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, and `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 base C-language standard. This is unlike the situation with signals, where certain signals are reserved for the implementation. This may mean that the Ada runtime system is prevented from using certain features internally on platforms that only support the minimum portable values. This still seems preferable to the alternative, which is to reserve an arbitrary amount of each limited system resource for possible use by the Ada runtime system. The amount of a resource needed by the Ada runtime is not known, and may well be zero in some 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 {3} 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”.

- 1 — Configurable functions for options have the same name as the corresponding
- 2 option, but with either “_Is_Supported” or “_Are_Supported” appended.
- 3 — Configurable functions for limits have the same name as the corresponding limit.
- 4 — Constants for the portable values of limits have the same name as the corre-
- 5 sponding limit, but with “Portable_” prepended.
- 6 — Blanks in the names of options and limits are replaced with underscores.

7 Unfortunately, POSIX.5 {B3} did not follow these naming rules in all cases, so that
8 this standard now has duplicate names for many constants, static subtypes, and
9 configurable functions. The old name was retained for compatibility with POSIX.5
10 {B3}, but is obsolescent, and the new name is provided for future use.

11 The packaging structure of POSIX.5 {B3} placed all constants and static subtypes for
12 limits and options in package `POSIX`. However, because of the many new limits and
13 options added by POSIX.5b {3}, it was decided to split all options and limits out into
14 two new packages, `POSIX_Options` and `POSIX_Limits`. POSIX.5b {3} retains the
15 old names in `POSIX` for compatibility with POSIX.5 {B3}, and places the new names
16 for options in `POSIX_Options` and the new names for limits in `POSIX_Limits`. PO-
17 SIX.5 {B3} already had placed the runtime callable functions for options and limits
18 in separate packages; `POSIX_Configurable_System_Limits` contains functions
19 for options and limits that are not pathname-specific, and `POSIX_Configurable_-`
20 `File_Limits` contains functions for options and limits that are pathname-specific.
21 POSIX.5b {3} merely adds the new names to these packages.

22 Thus, the placement rules for constants, static subtypes, and configuration functions
23 having to do with options and limits may be summarized as follows:

- 24 — Static subtypes for limits and constants for portable values of limits are in pack-
- 25 age `POSIX_Limits`.
- 26 — Static subtypes for options are in package `POSIX_Options`.
- 27 — Configuration functions for options and limits that are not pathname-specific are
- 28 in package `POSIX_Configurable_System_Limits`.
- 29 — Configuration functions for options and limits that are pathname-specific are in
- 30 package `POSIX_Configurable_File_Limits`.

31 **B.2.4 Package `POSIX`**

32 The package `POSIX` contains a common core of declarations that are used by the other
33 POSIX/Ada interface packages. For implementations that support child packages,
34 all other POSIX/Ada interface packages are permitted to be implemented as child
35 packages of package `POSIX`. (See §1.3.1.1.)

36 **B.2.4.1 Options and Limits**

37 The preferred and complete interfaces for obtaining information about whether an
38 implementation supports an optional feature are defined in the package `POSIX_-`
39 `Options`.

40 The preferred and complete interfaces for obtaining information about implementa-
41 tion limits are defined in the package `POSIX_Options`.

However, for upward compatibility with POSIX.5b {3}, redundant interfaces for obtaining information about optional features and limits defined in POSIX.1 {2} are 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, this 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 the Ada language standard [Ada95 (ISO/IEC 8652: 1995)1.1.3(6)] which 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 this 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 {B3} 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` is added by POSIX.5b {3} 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 {B3} defined blocking behavior for the following classes of operations:

- Ada Text I/O operations.
- POSIX synchronous I/O operations.
- File locking.
- Wait for child.

Note that POSIX.5b {3} adds only one new blocking behavior subtype `Realtime_Blocking_Behavior`, which applies to all the “realtime” operations added by POSIX.5b {3}. The alternative was to have individual blocking behaviors for the separate options (e.g., semaphores, message queues, etc). 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:

- 1 — **Tasks..Tasks:** Realtime operations may be relied upon to not block more than
2 the calling task.
- 3 — **Program..Program:** Some realtime operations may block all tasks within the
4 process; consult the conformance document for specific information.
- 5 NOTE: Some operations are required to support task blocking behavior, regardless
6 of the range of `Realtime_Blocking_Behavior`.
- 7 — **Tasks..Special:** Some realtime operations may block some or all tasks within
8 the process; consult the conformance document for specific information.
- 9 Other ranges are not anticipated, since they would imply that all realtime operations
10 block additional tasks as well as the calling task.

11 **B.2.4.3 Characters, Bytes, and I/O Units**

12 POSIX and C are really based on byte-oriented file systems. The C `char` type, bytes,
13 and the I/O units upon which input/output operations are performed are really all
14 the same in POSIX. File names and user names are arrays of `chars`, and file contents
15 are streams of `chars`. POSIX.5 {B3} defined a single type, `POSIX_Character` to rep-
16 resent all these objects. The intent was that the type `POSIX_Character` correspond
17 to the C `char` type, and that implementations would extend it to include all possible
18 byte values of the underlying system.

19 However, in the POSIX.5 {B3} ISO fast-track ballot, and in the ballot of PO-
20 SIX.5b {3}, objections were raised to the use of `POSIX_Character` in contexts where
21 the base standards use byte, mainly as a unit of untyped I/O data or or as a unit of
22 memory. These objections touched on several issues:

- 23 — The trend toward internationalization of character sets and the associated data
24 types implies that a graphic character is not always represented by a single byte
25 of data.
- 26 — Concerns that the mapping of `POSIX_Character` to a memory representation
27 might leave holes, i.e., bit patterns that do not correspond to characters.
- 28 — Concerns that an Ada implementation might impose some overhead on the con-
29 version of character values to other types of data.
- 30 — The provision in Ada95 (ISO/IEC 8652: 1995) of standard marshalling and un-
31 marshalling operations (`'Write` and `'Read`) for streams, which are based on
32 the data type `Stream_Element`. It is clearly desirable to be able to use these
33 operations in conjunction with the POSIX message passing and I/O operations,
34 especially when complex data objects are being transmitted.

35 The possibility was considered of just re-emphasizing the original intent of POSIX.5
36 {B3} that the values of the type `POSIX_Character` are just bytes – they must each
37 fit in one byte of storage and the range of values must cover all valid one-byte bit-
38 patterns. Conversion operations between `POSIX_String` and `Stream_Element_-`
39 `Array` could be added. But, this was unappealing because of the ample evidence from
40 balloters that the word “character” in the type name led them to think of this as a
41 conventional character type, rather than as a “byte” type. There was also concern
42 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 {3}. 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 Ada95 (ISO/IEC 8652: 1995). New I/O operations (e.g., message passing and asynchronous I/O) are defined only on the new type. The I/O operations on the `POSIX_String` type are retained, for compatibility with POSIX.5 {B3}, 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 Ada95 (ISO/IEC 8652: 1995) 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 Ada95 (ISO/IEC 8652: 1995) 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`, and then 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. This is a function, rather than a constant, to ensure that 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`) This permits the end user to write expressions of the form:

```
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, and 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 `Booleans` 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:

```
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`: Mask no signals other than those that already have been masked.

- 1 — **RTS_Signals**: Signals normally used by the Ada runtime system will be masked
- 2 during the operation and will not cause it to be interrupted.
- 3 — **All_Signals**: All signals, including implementation-defined signals, are
- 4 masked during the operation.

5 A **Masked_Signals** attribute is also provided for process creation templates, since
6 process creation can also involve interruptible operations.

7 This three-value enumeration was chosen instead of a signal set because it is not
8 intended as a substitute for the operations that block and unblock a set of signals.
9 The purpose is to cover those signals that cannot otherwise be masked, namely, the
10 reserved signals. The three choices provided are themselves adequate for most ap-
11 plications, and if an application needs finer-grained control (for signals that are not
12 reserved signals) it can use the facilities of package **POSIX_Signals**. Another reason
13 for not allowing the specification of a set of signals here is that it would require mov-
14 ing the declaration of the signal and signal set types from package **POSIX_Signals**
15 to package **POSIX**.

16 The POSIX/Ada operations that have the **Masked_Signals** parameter correspond
17 exactly to the POSIX C-language functions that can return with *errno* set to [EINTR].

18 **B.2.4.7 Time Types**

19 The C-language type *time_t* is mapped to the Ada type **Time** in package **Calendar**,
20 and the C-language type *clock_t* is mapped to the Ada type **Tick_Count** in several
21 places (see §4.2).

22 The POSIX time-zone **TZ** functionality included in Section 8 of POSIX.1 {2} is placed
23 in Section 4 of this standard. Since **TZ** should not be used to modify the behavior of
24 the predefined **Calendar** package in Ada, an analog to the Ada predefined package
25 **Calendar** is provided. This package provides the same functionality, but applies the
26 information contained in the **TZ** environment variable to a value of type **POSIX_-**
27 **Calendar.Time**. This package also provides operations to convert between values
28 of **Calendar.Time** and **POSIX_Calendar.POSIX_Time**.

29 The C-language structure *timespec* is mapped to the private type **Timespec** type
30 in the package **POSIX**. This type is placed in **POSIX** rather than in **POSIX_Timers**,
31 because it is used as the type of a parameter of function **Await_Signal_Or_Time-**
32 **out** in package **POSIX_Signals**. Since **POSIX_Timers** depends on **POSIX_Signals**,
33 putting **Timespec** in **POSIX_Timers** would create a circular compilation dependency.
34 Moreover, it is reasonable to expect that **Timespec** may find uses in future exten-
35 sions of other POSIX interfaces.

36 The idea of providing separate types for relative and absolute time, similar to
37 **Ada.Real_Time** in Ada95 (ISO/IEC 8652: 1995), was considered. The decision to
38 use a single type is based on simplicity and consistency with the base standards.

39 The possibility of making **Timespec** a visible record type, with explicit component
40 fields for **Seconds** and **Nanoseconds**, was considered. The C-language interface
41 specifies *timespec* as such a structure. This was rejected on the grounds that the
42 underlying system representation of time might not be the same as specified in PO-
43 SIX.1b {3}. 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. This was not done because of concern that it would impose too much overhead on non-realtime applications, 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.

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*. This 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. Lacking this, 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 {3}, 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- {3} and POSIX.1c {4}. The old interfaces are retained in *POSIX* for compatibility with POSIX.5 {B3}, 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 {3}, 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 {3} and POSIX.1c {4}. The old interfaces are retained in POSIX for compatibility with POSIX.5 {B3}, 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

This is a renaming of the Ada95 (ISO/IEC 8652: 1995) 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 POSIX.5b {3} (see Section 12), operations on the type `System.Address` are required, beyond those required by Ada83 (ISO/IEC 8652: 1987). Therefore, POSIX.5b {3} imposes some specific requirements on this data type from Ada95 (ISO/IEC 8652: 1995), whether or not the implementation fully supports Ada95 (ISO/IEC 8652: 1995).

B.2.9 Package System_Storage_Elements

Some interfaces defined in POSIX.5b {3}, such as memory management and low-level I/O operations, require facilities of Ada95 (ISO/IEC 8652: 1995) that are defined in the package `System.Storage_Elements`. An implementation of POSIX.5b {3} is required to support these features, whether or not it fully supports Ada95 (ISO/IEC 8652: 1995).

To provide a transition path from Ada83 (ISO/IEC 8652: 1987), 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, which provides 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, for use with the memory management operations for which the implementation is allowed to require page alignment. 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

1 and `Adjust_Length(X'Address, X'Size/System.Storage_Unit)` for the page-
2 aligned length of the region.

3 The exception `Program_Error` was chosen for errors in these operations, for con-
4 sistency with the Ada95 (ISO/IEC 8652: 1995) treatment of errors in address arith-
5 metic.

6 **B.2.11 Environment Description**

7 This section is very similar to Clause 2.6 of POSIX.1 {2}. It is important to maintain
8 a direct mapping of environment variables between C and Ada, since environment
9 variables are used to communicate between C and Ada programs, and between pro-
10 grams and shell commands.

11 **B.2.11.1 Error Codes and Exceptions**

12 The rationale for the general error reporting model is explained in §B.1.9.1.

13 In POSIX.5b {3}, linkage is made between the `Exception_Message` feature of
14 Ada95 (ISO/IEC 8652: 1995) and the POSIX error codes.

15 Note that implementations are free to raise `POSIX_Error` for implementation de-
16 fined reasons, and may also add additional error codes for implementation defined
17 errors.

18 In addition, there is no Ada name corresponding to EDOM (domain error), since that
19 is the C-language equivalent of the Ada exception `Constraint_Error`.

20 **B.2.11.2 System Identification**

21 These functions are a de-multiplexing of the C-language `uname()` function.

22 **B.3 Process Primitives**

23 **B.3.1 Package `POSIX_Process_Primitives`**

24 This standard specifies that an Ada95 (ISO/IEC 8652: 1995) active partition corre-
25 sponds to a POSIX process, so that process-level POSIX operations affect all tasks
26 within the active partition. This choice seems to cause the least conflict between Ada
27 language rules and POSIX, and it resolves several binding issues cleanly:

- 28 — `Exec` and `Start_Process` are well- defined as producing a new active partition
29 execution.
- 30 — `Exit_Process` is well-defined as ending an active partition execution.
- 31 — The current process ID is well-defined.
- 32 — The bindings of file descriptors (and the descriptors of other file-like objects, e.g.,
33 named semaphores and message queues) are well-defined over the entire active
34 partition.
- 35 — All process-specific state information is well- defined throughout a program, in-
36 cluding process environment variables and system accounting times.

Note that 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 {B8}.

Certainly, there will be some applications that 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, this corresponds to 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. This is done for increased safety – 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. Note that this means 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

Note also that 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 pre-check 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.

1 Ordinarily, exit from an Ada program is automatic due to normal completion or an
2 unhandled exception. Standard exit status values are defined for two cases, plus the
3 case of failed process creation. The choice of the value zero for `Normal_Exit` is tradi-
4 tional in UNIX systems. The choice of values for `Unhandled_Exception_Exit` and
5 `Failed_Creation_Exit` is arbitrary, subject to the constraint that it must not be
6 near zero or above 127 (but see *Hitchhikers Guide* {B1}). Values near zero are likely
7 to be already in use for special purposes. Extreme high values have the same problem
8 as low values (since they are equivalent to negative values near zero). Values above
9 127 are conventionally used by UNIX system C-language applications for processes
10 killed by signals. This practice is intentionally not followed by this standard in the
11 case that the signal is converted by the Ada language implementation into an excep-
12 tion, which propagates to the end of the active partition and causes it to terminate.
13 In such a case, the exception is no longer a signal. Following the C-language conven-
14 tion for processes terminated by signals would therefore be misleading and incorrect.
15 Moreover, since the same exception might be raised for other reasons, it would be an
16 excessive burden on Ada implementations to remember whether an exception was
17 caused by a signal and to split the exit processing into two cases accordingly. This
18 information would have to be stacked and restored in the case of exceptions raised
19 within exception handlers.

20 The requirement that main subprograms be parameterless procedures reflects estab-
21 lished Ada practice and concern that Ada compilers should not be required to support
22 as main subprograms functions returning values of type `Exit_Status`.

23 The procedure `Exit_Process` is provided to permit an application to terminate itself
24 immediately. This provides a capability that is otherwise not available in Ada—to
25 terminate all the tasks of a partition abruptly. The definition effect of `Exit_Process`
26 on tasks is intentionally not tied to Ada abortion or termination, since those terms
27 have established meanings that are irrelevant to this operation (which is performed
28 by the underlying operating system rather than the Ada language implementation).
29 Note that the effect of this operation is not consistent with semantics of **abort**, e.g.,
30 it does not recognize abort-deferral or perform finalization of controlled objects.

31 The underlying operating system can be relied upon to close any open file descriptors,
32 but cannot be relied upon to flush any output buffers that might be in use by the
33 application or the Ada implementation. There is no requirement that Ada output
34 buffers be flushed; this was considered to be impractical, since it would make the
35 implementations of all output packages too dependent on the implementation of the
36 POSIX/Ada interface, or vice versa. Also, flushing buffers is potentially too time-
37 consuming for situations where a fast process exit is desired. Strictly Conforming
38 applications should close or flush all open files before exiting, using the operations
39 provided for that purpose.

40 Note that if the Ada language implementation creates any “hidden” processes to
41 support execution of an Ada active partition, `Exit_Process` must terminate all of
42 these; this is a consequence of the rule that an Ada active partition execution must
43 appear to be a single POSIX process.

1 **B.3.4 Wait for Process Termination**

2 The operations that request the status of a child process could have been specified
3 as functions, returning a value of type `Termination_Status`. This alternative was
4 rejected because the type of the result is private, and typically more than one com-
5 ponent from the status value is required in subsequent programming, so the value
6 would have to be assigned to a variable anyway. Moreover, a procedure call is syn-
7 tactically a statement, and thus is similar to other blocking Ada forms, such as the
8 **delay** statement, the entry call, and the **accept**. This warns the programmer more
9 clearly about the side effects of this operation. The **out** parameter (which cannot
10 be omitted) is in the leading position, so that the other parameters can be omitted,
11 resulting in the default values being used.

12 The possibility of splitting these operations into separate procedures, rather than
13 using the parameters `Block` and `Trace_Stopped` was considered. The resulting
14 increase in clarity of expression was not considered sufficient to outweigh the loss
15 resulting from proliferation of procedures (from two to eight).

16 **B.3.5 Package `POSIX_Unsafe_Process_Primitives`**

17 It is clearly very difficult to define fully the semantics of the *fork()* and *exec()* func-
18 tions of the C- language binding in the presence of multiple Ada tasks in a way that
19 is useful and easy to implement. As a consequence, an application that uses these
20 operations directly will need to depend on special features of a particular Ada run-
21 time system and a particular POSIX/Ada implementation and therefore is not likely
22 to be portable. On the other hand, there are situations where nothing less general
23 than *fork()* or *exec()* will do. Therefore, the “unsafe” functions `Fork` and `Exec` (and
24 `Exec_Search`) are provided, but their use is made more conspicuous by putting them
25 into a separate package, labeled “unsafe”.

26 The text in §3.2 establishes conditions under which a Strictly Conforming POSIX.5
27 Application may use `Fork` and `Exec`.

28 **B.3.6 Package `POSIX_Signals`**

29 Signals are an especially troublesome feature of the POSIX interface because the be-
30 havior defined by the base standards does not map very directly to the Ada language.
31 It is necessary to reserve some POSIX signals for the Ada language implementation
32 in order to preserve Ada rules concerning the predefined exceptions `Program_Error`
33 and `Constraint_Error`. Also, there is no construct in Ada that corresponds safely
34 to the C-language asynchronous signal catching function (also called a signal han-
35 dler) – nor is one needed. Consequently, the POSIX/Ada binding restricts the view of
36 C signal facilities accessible to Ada applications. All operations deemed useful to Ada
37 applications are exposed, including the ability to take default signal actions and to
38 wait synchronously for signals. Also, the POSIX/Ada signal interface is intended to
39 be sufficient to achieve whatever level of signal interoperability is possible between
40 Ada and C portions of a mixed-language application, depending upon the support
41 provided by the underlying operating system.

1 **B.3.7 Signal Model**

2 The original signal model of POSIX.5 {B3} was based on the POSIX.1 {2} signal
3 model and the interrupt handling model of Ada83 (ISO/IEC 8652: 1987). Since then,
4 several things have changed:

- 5 — POSIX.1b {3} has added functions to synchronously wait for signals.
- 6 — POSIX.1c {4} has added multi-threaded processes and has redefined the process-
7 level signal masking operations to apply to the calling thread.
- 8 — Ada95 (ISO/IEC 8652: 1995) has made the use of task entries as interrupt han-
9 dlers obsolescent.
- 10 — It has become clear that signals are not interrupts and the POSIX/Ada binding
11 should not treat them as such.

12 Due to these changes, much of the rationale for the treatment of signals in POSIX.5
13 {B3} no longer holds.

14 At the heart of the problem is how to treat *sigaction()*. The POSIX C interface allows
15 an application to install C functions as asynchronous signal handlers via *sigaction()*.
16 The C application can use *longjmp()* out of a signal handler to an earlier *setjmp()*
17 point. This capability appears to be needed in some situations, where the desired
18 response to a signal is to abandon a computation abruptly. Because the Ada appli-
19 cation (or the Ada runtime system) is bound to need to execute some async-signal
20 unsafe operations after such a *longjmp()*, it is not safe in general to allow the Ada
21 application direct access to this capability. However, there are several Ada language
22 constructs that have closely related semantics. These are propagation of an excep-
23 tion, **abort** of a task, and the Ada95 (ISO/IEC 8652: 1995) asynchronous **select**
24 statement. A central goal of this binding, therefore, has been to allow the Ada im-
25 plementation to make use of signal handlers and *longjmp()* to implement these Ada
26 operations.

27 This standard is based on the assumption that asynchronous signal handlers are
28 used only by the Ada runtime system. Notification of synchronously generated sig-
29 nals to the Ada application is via Ada exceptions, raised by the runtime system.
30 Notification of other signals is via task-level blocking operations modeled after the
31 *sigwait()* interface of POSIX.1c {4} and the *sigwaitinfo()* interface of POSIX.1b {3}.
32 The rationale for this decision is discussed further in §B.3.13.

33 **B.3.8 Signal Masking and Related Concepts**

34 Originally, POSIX.5 {B3} was based on a model in which there is only one signal
35 mask, whose effect is process-wide. This agreed with existing practice in implemen-
36 tations of Ada under POSIX.1 {2} at that time, in which the Ada language implemen-
37 tation provided its own user-level kernel for scheduling multiple threads of control
38 within a process. Since then, some operating systems have begun to provide support
39 for multiple threads of control within a process, and POSIX.1c {4} has defined a stan-
40 dard for such services, with a C-language interface. The POSIX.1c {4} signal masking
41 model requires that there be an individual signal mask for each thread of control; it
42 defines some new operations for manipulating this mask, and specifies much of the
43 behavior of signal masking on a per-thread basis.

This standard cannot presume that the underlying operating system (if any) supports POSIX.1c{4} (see §B.1.2). Therefore the clear distinction between a thread-level signal mask and a process-level signal mask in the base standards 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. POSIX.5b {3} has assumed that process-level signal masks will not be supported by many underlying operating systems. 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 signal masking operations of POSIX.5 {B3} is now redefined to apply to the calling task (and possibly to other tasks as well).

Other changes to the signal masking model made by POSIX.1b {3} and POSIX.1c {4} 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 {4} (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. This 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 {B3} 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 {3} 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.1c {4} is supported, there are some difficulties:

- When *sigwait()* is called, all the signals which the thread is ready to accept must be masked, at least for the calling thread, or else the effect is undefined [POSIX.1c{4} 3.3.8.2]. 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 which the thread was prepared to accept is unspecified. In particular, if one wants to have the default action, one must call *sigaction()* to restore this. Still, unless the signal was masked before the call to *sigwait()* there will be a window of vulnerability in which arrival of a signal will have 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 {4} suggest 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 unmask 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 non-reserved signals (i.e., those which a user might want to accept) there are only two cases of signal masking that 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 non-realtime signals) from the parent process. This 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. This would eliminate a window of vulnerability during which a new process that expects to receive and handle signals cannot protect itself unwanted terminating (which is the default action) on early arriving signals, the window being between 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), and if the task attempts to execute *Await_Signal* the effect is undefined. The package body can prevent this by calling *Block_Signals* and then synchronizing (e.g., by rendezvous) with the task that uses the signal. The task that uses the signal must 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.

1 It is not desirable to initially block *all* the non-reserved signals in this fashion, since
2 one would then lose the ability to use the default signal actions for job control. Also,
3 there is the issue of compatibility with POSIX.5 {B3}. Both of these objections can be
4 met, however, if only the real-time signals are required to be blocked initially in the
5 environment task, since that leaves the initial masking of the job control signals un-
6 der control of the parent process, and the real-time signals are new for POSIX.5b {3}.

7 If a requirement for initial blocking of the realtime signals is added, it will need
8 to be stated in a form that does not impose any such requirement on non-Ada pro-
9 grams, and which is compatible with existing operating system implementations.
10 This means the blocking of the signals must be allowed to be done by the new pro-
11 cess, after creation. On the other hand, it must also be compatible with Ada-only
12 embedded implementations, where it may be preferable to block the realtime signals
13 from the first instant of the process creation. The signals must be blocked no later
14 than the elaboration of the body of package `POSIX_Signals` or any other package
15 that depends on it, since after that point the application may try to await a signal
16 or use an operation that causes a signal to be generated for event notification. This
17 leads to the question of whether a requirement for initially blocking realtime signals
18 should apply to all Ada active partitions, or just to applications that use the package
19 `POSIX_Signals`.

20 The suggestion of a requirement for initially blocking the realtime signals came late
21 in the ballot process for POSIX.5b {3}, after consensus had been achieved. There was
22 some concern that it would require several ballot iterations to determine whether
23 there was consent for such a change, and to work out an acceptable formation of the
24 details. Therefore, the issue was postponed until the next amendment or revision
25 to this standard. However, informative notes were added to the standard, to warn
26 implementors and users that they should be prepared for the imposition of such a
27 requirement in the future.

28 The signal masking and unmasking operations are required to guarantee the follow-
29 ing: A task starts with the condition that the signal(s) of interest are masked in all
30 other tasks. It unmask the signal (which might have the effect of unmasking the
31 signal for other tasks as well). Later, the task again masks the signal. It must be the
32 case that the signal is again masked in all tasks. In order to guarantee this behavior,
33 the implementation is allowed to impose a restriction that a task may not unblock a
34 signal that is already unblocked by another task in the same process. The task that
35 unblocks a signal becomes in effect the “signal manager” for that signal for the entire
36 process.

37 The possibility was considered of putting additional restrictions on signal blocking,
38 to further enforce the idea of each signal having only one “manager” task at a time.
39 For example, a task might not be allowed to block a signal unless the same task had
40 previously unblocked that signal. This restriction was not imposed, due to problems
41 dealing with details, such as the proper treatment of any signals that are already
42 unblocked when the process starts.

43 NOTE: The next revision of this standard should perhaps review this possibility, of defin-
44 ing further restrictions on signal blocking to enforce the concept of there being only one
45 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 {B3} already had the operations `Block_Signals` and `Unblock_Signals`. These operations might have been deleted from POSIX.5b {3}, since the original effect (of changing the process-wide signal mask) can no longer be required with POSIX.1c {4}. but that might break some existing applications, and it 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 such that a task that unmask 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.1c {4}. This is also consistent with a ballot objection of the P1003.4 Working Group which maintains that process-wide signal masking cannot be used with predictable effects in a multi-threaded 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 this by exercising discipline in the way it uses async-unsafe operations and the way it does task context-switches, but 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.)

A renaming is provided to provide both the long (Ada) names of signals and the conventional short (C) names. The use of the long identifier within Ada programs is strongly encouraged.

B.3.11 Standard Signals

The base standards define 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, ...) which 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 inter-process 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.

1 — Realtime signals may carry an application-defined data value which may be used
2 to distinguish between multiple occurrences of a signal.

3 — Notification for realtime signals is required to be in priority order (lowest signal
4 number first), so that some measure of predictable behavior is possible. There is
5 no notification order specified for other standard signals.

6 Constants are provided for values of the type `System.Address` corresponding to
7 each of the non-reserved non-realtime standard signals. These are to be used in
8 an address clause to bind a task entry to a particular signal. These were made con-
9 stants, rather than functions, because some Ada implementations might require that
10 the address given in an address clause be a compile-time constant. No constants are
11 provided for the reserved signals, since an application is not allowed to accept these
12 signals. Similarly, no constants are provided for `Signal_Kill` and `Signal_Stop`,
13 since these signals cannot be accepted, caught, or ignored. The function `Signal_-`
14 `Reference` returns an address for each realtime signal that may be used in a similar
15 manner.

16 In contrast, signal *names* are defined for the reserved signals and for other signals
17 that cannot be accepted, caught, or ignored, because an application might need to
18 send these signals to other (non-Ada) processes. The value `Signal_Null` is also
19 provided, even though it cannot be sent or received, because this value has other
20 uses. `Signal_Null` is a safe initial value for variables of type `Signal`, and may be
21 used as a parameter to `Send_Signal` for checking the validity of process and process
22 group identifiers.

23 **B.3.12 Signal Sets**

24 The type `Signal_Set` is a private type with operations to perform common set op-
25 erations. This type might be implemented as an integer bit mask or as a `Boolean`
26 array. Implicit initialization of objects of type `Signal_Set` is required in §3.3.7. This
27 can be implemented by making this a record type (e.g., containing a vector) or an
28 access type and using the default initialization provided by Ada for these types.

29 This standard is more restrictive in this matter than the C-language binding in two
30 ways:

- 31 (1) It does not allow signals that are not defined in the standard to be members of
32 the “empty” set.
- 33 (2) It does not allow uninitialized signal sets.

34 **B.3.13 Examine and Change Signal Action**

35 The use of *sigaction()* is reserved by this standard for the Ada runtime system (RTS).
36 The main reason is safety. There are several unsafe aspects to asynchronous sig-
37 nal handlers. First, the RTS needs to control the actions and masking of reserved
38 signals in order to implement certain runtime checks and tasking operations. Al-
39 lowing the application to interfere would result in undefined behavior. This does not
40 yet rule out the possibility of an interface that allows application handlers for those
41 signals that are not reserved for the RTS. The main reason for not allowing this
42 is that such an Ada procedure might unknowingly call an RTS routine that is not

1 async-signal safe, again resulting in undefined behavior. This is especially true with
2 the advent of POSIX.1c {4}, with many operations that are not async-signal safe.
3 Short of narrowly specifying a particular task implementation, which would prevent
4 implementors from taking advantage of the most efficient primitives of a particular
5 underlying operating system, there is no way to predict exactly which Ada source
6 constructs will result in calls to async-signal unsafe operations. POSIX.1c {4} also
7 provides another strong reason for not allowing async. signal handlers, in that the
8 interactions with *sigwait()* are undefined, and this standard provides an Ada binding
9 for *sigwait()*.

10 A less important reason for not allowing signal handler procedures is that Ada83
11 (ISO/IEC 8652: 1987) does not have procedure pointers. While Ada95 (ISO/IEC 8652:
12 1995) does introduce procedure pointers, this standard is intended to be compatible
13 with Ada83 (ISO/IEC 8652: 1987) as well as Ada95 (ISO/IEC 8652: 1995), so the use
14 of this new feature is ruled out.

15 The capability of blocking or ignoring certain signals was almost omitted from PO-
16 SIX.5 {B3} because it is difficult to reconcile with the Ada model of rendezvous.
17 However, this would leave the application with no way of limiting the rate of arrival
18 of troublesome signals. This would be especially poor since the default action for
19 most signals is to terminate the process.

20 Because an application that is not interested in a signal may want to ignore it, or
21 restore the default signal action, but direct access to *sigaction()* is not desirable, the
22 operations *Ignore_Signal* and *Restore_Default_Action* are provided.

23 For the *Ignore_Signal* operation, this standard requires dequeuing of any pend-
24 ing signals, since it preserves an important invariant for programming with signals.
25 Specifically, it should be true that a process can discard all the deliveries of a signal
26 generated prior to a given point in time by ignoring the signal while the signal is not
27 blocked. (For the reasons, see the next section.)

28 The flag bit *SA_NOCLDSTOP* of the C-language binding provides a way to specify
29 that the *Signal_Child* signal not be generated for a parent process whenever any
30 of its children stops. Since the Ada runtime system hides the equivalent of the *sigac-*
31 *tion()* function of the C-language binding, there is no direct way of translating this
32 feature. The ability to interrogate and turn this bit on and off is therefore provided
33 by means of separate operations, *Set_Stopped_Child_Signal* and *Stopped_-*
34 *Child_Signal_Enabled*.

35 **B.3.14 Signal Queuing**

36 The basic POSIX signal delivery mechanism is inherently unreliable for signal count-
37 ing. It only guarantees that, following posting of a signal by some process, that signal
38 will be delivered at least once some time after the posting. This mechanism has well-
39 defined properties that are quite useful as long as the user does not believe all signal
40 occurrences are delivered. Signals are not intended as event-counting mechanisms.
41 Any application that attempts to use them in that way is not portable and is probably
42 unreliable on most POSIX systems.

43 POSIX.1 {2} does permit implementations to queue signals, but even if queueing
44 is provided, it is not required to faithfully queue every signal. Hence, the portable

use for (non-realtime) 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 {3} adds 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

The base standards define two delivery modes for signals. In POSIX.1{2} notification provided only the signal number. POSIX.1b {3} 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 or not.
- `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 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 this is an inherently untyped interface. 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 this 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. The main change in this revision, to conform with POSIX.1c {4}, is loosening of the semantics to take into account the possibility of 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 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.1c {4} is supported.
- The interrupt entry mechanism is considered obsolescent for Ada95 (ISO/IEC 8652: 1995), and so use of the task entry mechanism to receive signal notification is discouraged.
- The attempt by POSIX.5 {B3} to identify POSIX signals with hardware interrupts was misguided in any case, due to semantic differences.

POSIX.1b {3} and POSIX.1c {4} create potential confusion by providing a non-orthogonal set of operations for waiting for signals, with overlapping semantics. These 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, we have chosen to designate `Await_Signal` 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 the POSIX.1c {4} C-language interface, 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 {B3}.

This leads to the question of whether the form of `Await_Signal_Or_Timeout` that returns a value of type `Signal` should be supported, in the case that the Realtime Signals option is not supported. POSIX.5b {3} requires support for the sake of orthogonality, and on the grounds that the functionality is already present in the C-language interface by using a timer to generate an alarm signal, but cannot be achieved in this manner for Ada applications because `Signal_Alarm` is reserved for the Ada implementation. Another supporting argument is that *sigwait()* duplicates the functionality of a simple accept statement on a task entry bound to a signal and POSIX.5 {B3} 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. Note that `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.1c {4}, 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, so a thread wanting to receive notification via a per-thread

signal sent by *pthread_kill()* would need to accept it synchronously, using *Await_Signal*, but then 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 Ada95 (ISO/IEC 8652: 1995) asynchronous transfer of control mechanism), with appropriate protection and cleanup mechanisms.

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 C- language 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, there is no *alarm()* function 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 *sigwait-info()* 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 without the extensions of POSIX.1c {4}, 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 POSIX.1c {4}, 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 {B3}. That interface was modeled after the interrupt entry interface of Ada83 (ISO/IEC 8652: 1987), which has been labeled obsolescent by Ada95

(ISO/IEC 8652: 1995). The signal entry interface is retained by this standard, for continued support of applications developed for POSIX.5 {B3}, but it is made optional to reflect the fact that it is based on a faulty analogy between signals and interrupts.

There is a known gap in the signal entry interface definition, 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 rectify this, 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 new Ada95 (ISO/IEC 8652: 1995) model for hardware interrupt handling, based on “protected procedures”, does not fit the POSIX.1c {4} 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. This is in contrast to the POSIX.1c {4} model, in which 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 {4}, 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 Ada95 (ISO/IEC 8652: 1995) 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 non-sharable 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. This is true regardless of what language those subsystems are written in. It is especially likely to be a problem where off-the-shelf subsystems (libraries) are linked with an Ada program. In the case that 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 time-outs). 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 was changed from POSIX.1 {2} to improve modularity. Specifically, the following changes were made:

- Clauses 4.1, 4.2, and 4.3 were combined into `POSIX_Process_Identification` to centralize handling IDs related to processes.
- Clause 4.4 was moved to package `POSIX` (Section 2) to centralize system inquiry operations in one place.
- Clause 4.6 and subclause 3.1.2.2 *argc*, *argv* were combined in `POSIX_Process_Environment` since together they define the strings that provide control of a command.
- Subclause 4.7.1 was moved to Section 7 to centralize controlling terminal functions.
- Subclause 4.7.2 was moved to Section 6 to centralize functions related to pathnames.
- Portions of clause 2.6 were moved into `POSIX_Process_Environment` to centralize environment handling.
- Clause 5.2 was moved to this section 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 separate types, although cursory reading of POSIX.1 {2} would suggest a single type was appropriate. If one carefully reads POSIX.1 {2}, 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

`Get_Groups` does not specify whether the effective group ID of the calling process is included in the list of supplementary groups. This follows the POSIX.1 {2} specification exactly. 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, 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 {2} 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 {2}, 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 hours.

This standard does not require systems to have any greater accuracy or range for timings than in the base document, even though this 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 to provide an accuracy and range that matches that which a system provides; no other definition for this type allows for that.

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. This allows additional data elements to be declared by an implementation, with access from a separate package supplied with the implementation. Although POSIX.1{2} 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 than 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 {2} precisely.

B.4.3.3 System Time

The C-language `time_t` type and the POSIX.1{2} `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 {2} 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 {2} does not specify or enforce that convention, so this standard also does not.

NOTE: As of Ada95 (ISO/IEC 8652: 1995), there is a standard Ada language-defined interface for argument lists, in the package `Ada.Command_Line` [Ada RM {1}A.15]. 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 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

1 for an application-specific format using an instance of the generic iterators. Similarly,
2 a value function that constructs an arbitrary environment can easily be constructed
3 from standard Ada loop and string input operations.

4 The character '=' is forbidden in environment variable names for safety, despite a
5 potential cost in performance; the implementation has to check each character in a
6 variable name. Allowing '=' in variable names could cause problems when building
7 an Ada binding on top of a POSIX system implemented in C, in which the (name,
8 value) pair is represented as a single string with '=' marking the start of the value.

9 This standard leaves the processing of multiple instances of environment variables
10 as undefined. The alternative, of specifying behavior for these (erroneous) environ-
11 ments, would be of benefit in only rare cases. Where efficiency or other concerns do
12 not dictate otherwise, it is suggested that the handling be as follows:

- 13 — Delete operations should ensure that all copies of a name are deleted.
- 14 — Set and access operations should reliably access the same (name, value) pair.
- 15 — Iteration operations should iterate over all copies of a name, with the first name
16 in a set of identical names being the same name retrieved by access functions.

17 The semantics of set and delete operations for environment variables were defined to
18 be similar to formal sets, in which adding or subtracting an item ignores the previous
19 state of the set. It was felt that these semantics were appropriate for typical uses of
20 environment variables.

21 Adding an additional parameter to the operations that would select alternative se-
22 mantics wherein exceptions are raised for setting an existing name or deleting a
23 nonexistent name was rejected. That approach was viewed as complicating the in-
24 terface with little benefit to the user. If an application truly requires these alternate
25 semantics, the user can test existence of an environment variable using `Is_Envi-`
26 `ronment_Variable` prior to the delete or set operation.

27 The effects of one task modifying the current environment while another iterates
28 over a value obtained from `Copy_From_Current_Environment` are undefined in
29 order to allow for efficient implementations.

30 Representing the type `Environment` presents some awkward choices. Suppose that
31 for compatibility with an underlying C implementation it is desirable to represent
32 an environment as an array of pointers, pointing to strings, where each string repre-
33 sents a (name, value) pair separated by '='.

34 Two ways to allocate space for the strings are

- 35 (1) Heap Allocation. The strings might be dynamically allocated in heap space, as
36 environment values are set and unset. This imposes no special limit on the size
37 of an environment, but poses problems for storage recovery since the storage of
38 such strings cannot be automatically recovered safely.
- 39 (2) Local Allocation. The strings might be allocated within a single contiguous block
40 of storage associated with each object of type `Environment`. This would impose
41 a limit on the size of an environment (as is permitted in POSIX.1 {2}), but would
42 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.

Some 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
    F00: constant POSIX_String:=
      POSIX_Process_Environment.Environment_Value_Of("Foo", Env);
    -- F00 is set null (by default) if it is not in Env
  begin
    -- Do processing on F00
    POSIX_Process_Environment.Clear_Environment(Env);
  end;
end;

```

If it is important to distinguish a null **F00** from **F00** not being defined in `Env` and if there is no appropriate string to use that is distinct from possible settings of **F00**, then `Is_Environment_Variable` can be used. This also eliminates the need for a local copy of the current environment, which eliminates 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. The approved Ada Commentary AI-00195 {B6} requires that the function `Calendar.Clock` “returns a value that reflects the time of day in the external environment.” It has been argued that this 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, a new `POSIX_Calendar` package is provided with the same functionality as the Ada predefined `Calendar`, but it 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 {3}. Old names are retained for compatibility with POSIX.5 {B3}.

B.5 Files and Directories

These packages are intended to provide the functionality of Section 5 of POSIX.1 {2}. Differences in organization from POSIX.1 {2} are noted below.

B.5.1 Organization

In general, this clause 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 data types manipulated, this means that Section 5 deals with files according to pathnames, rather than with `File_Descriptors`. 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 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 a file's `Status` value and the various functions in the package `POSIX_Configurable_File_Limits` that are overloaded to accept either a pathname or 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

1 POSIX.1 {2} to Section 4 of this standard, as the current working directory is an
2 aspect of the environment of a process.

3 **B.5.2 Package POSIX_Permissions**

4 **B.5.2.1 Permission Sets**

5 Permission_Set is defined as a Boolean array in order to allow the use of the logical
6 operators: **and**, **or**, and **not**, in the construction of permission sets. Constants of type
7 Permission_Set are defined corresponding to owner, group, and others access. A
8 constant is also provided for the union of owner, group, and others access. Finally,
9 constants are provided for setting the group ID and the user ID.

10 Note that this is a deviation from a stylistic convention followed nearly everywhere
11 else in this standard. This is an historical artifact, dating back to early work on
12 POSIX.5 {B3}. Extensions to this standard are encouraged not to emulate this case.
13 Instead, they should follow the predominant convention, which is to use a private
14 type in this sort of situation. For example, see the type Option_Set, which is used
15 in many similar situations.

16 The constant approach allows implementation extension packages to add elements
17 to the sets that are not necessarily contiguous. Thus, conforming applications can be
18 written that port to extended implementations with reasonably predictable results.
19 For example, consider an extended package, My_Permissions, that, in addition to
20 the user, group, and other permissions, adds in My_Special_User_Permissions
21 My_Special_Group_Permissions and My_Special_Other_Permissions. A pro-
22 gram to remove all “other” permissions could use the expression:

23 `all_but_other := arbitrary_set & (not Others_Permissions_Set);`

24 **B.5.2.2 Process Permission Set**

25 The *umask()* function in the C binding establishes the file mode creation mask of the
26 process to the value of its parameter and returns the prior value of the mask as its
27 result value. In Ada, this is considered poor style because a function should not have
28 side effects. Thus, the corresponding operation Set_Allowed_Process_Permissions
29 is not a function, but a pair of overloaded procedures where one returns the
30 old value and the other does not. Set_Allowed_Process_Permissions is over-
31 loaded with a version that returns the old permissions as an **out** parameter. Like
32 other operations, this operation is atomic in the face of tasking, but the user is still
33 responsible for coordinating the use of this operation by multiple tasks.

34 Future extensions to this standard should attempt to abide by this stylistic rule
35 against using functions for side-effects, even though it is apparently violated a few
36 places in this standard. In each of those cases there was some specific overriding
37 rationale. For example, the Open operation on files is a function. It might be argued
38 that such an operation is not called primarily for its side effect, even though it will
39 have a side-effect of creating a new open file description. The primary effect may be
40 viewed as to return a handle that can be used for I/O on the specified file. An analogy
41 can be drawn to the Ada **new** operation, which has the side-effect of allocating and
42 initializing storage, but which is still syntactically an expression. This argument was
43 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 {3}, there was opposition to continuing this precedent, for the open operations on new kinds of objects, such message queues. This 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 {2}.

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. To avoid these problems, and to simplify the application program, a function to interrogate the current setting, *Allowed_Process_Permissions*, has been added.

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 is shown below.

```

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;
end Example1;

```

1 **B.5.4 Package POSIX_File_Status**

2 The operations in this package are a straightforward mapping from the C operations
3 to retrieve attributes of a file object based on its pathname or file descriptor. First, an
4 object of the private type `Status` is retrieved. Then each attribute can be retrieved
5 from the `Status` object. New functions are provided for the “file-like” objects added
6 by POSIX.5b {3}: shared memory objects, message queues, and named semaphores.

7 **B.5.5 Package POSIX_Configurable_File_Limits**

8 The POSIX.5 {B3} package `POSIX_Configurable_File_Limits` has been made
9 dependent on `POSIX_IO` in order to support the POSIX.1 {2} `fpathconf()` capability,
10 which requires an argument of type `POSIX_IO.File_Descriptor`.

11 For each of the different limits or options associated with a file or directory, two
12 overloaded pairs of functions are provided. One pair of functions takes an open file
13 descriptor as a parameter. The other pair of functions takes a pathname as a param-
14 eter. The first function of each pair determines whether a specific limit exists, and
15 the second acquires the value of the limit if a limit exists. Using a separate function
16 name for each limit or option (rather than a single operation like `pathconf()`) provides
17 the advantage of allowing static type checking on the return type based on the limit
18 or option being acquired.

19 An alternative design approach considered was to have a single pair of subprograms
20 that return a variant record whose discriminant is the specific limit or option ac-
21 quired. This is the method used in POSIX.1 {2}, except that POSIX.1 {2} does not
22 have an explicit discriminant. This does allow arbitrary access to limits and op-
23 tions with the same subprogram signature, but eliminates static type checking on
24 the correspondence between the limit or option and the returned value. If there is
25 no explicit discriminant, then there is no type checking available at all. Moreover,
26 implementation extensions might require modification to the record type definition,
27 in the standard package specification, or else would have to provide duplicate opera-
28 tions on a different variant record type.

29 An alternative approach was also considered for determining whether the parameter
30 exists or not, specifically, the approach taken by POSIX.1 {2}, which uses a single
31 return parameter with a reserved value that is used to signal that there is no corre-
32 sponding limit. Such values may be either (1), as in C, or a very large value.

33 The chosen method is a combination of a `Boolean` return value and the use of a re-
34 served value. If a particular application needs to distinguish the difference between
35 a very large value and the absolute lack of a limit, two calls may be required. How-
36 ever, it is expected that the distinction is typically unnecessary, and only a call to
37 acquire the limit will be used.

38 The special character originally in this package, `Disable_Terminal_Special_-`
39 `Characters`, was moved to Section 7 because it is applicable only to terminal de-
40 vices.

41 Again, there are duplicate names of some of the functions in package `POSIX_Con-`
42 `figurable_File_Limits`. New names are added to conform to the naming rules

established by POSIX.5b {3}, and old names are retained to be compatible with POSIX.5 {B3}.

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`). This left the package `POSIX_Directories` with the single operation `For_Every_Directory_Entry`, so it was moved into `POSIX_Files`.

B.6 Input and Output Primitives

The operations in this section 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 asynchronous I/O interface added by POSIX.5b {3} 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 {3} is that the Ada95 (ISO/IEC 8652: 1995) 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 8 bits, or that the transfer of one byte of data means that only 8 bits are transferred. Additional parity bits or other “non-data” bits may be transferred.

In POSIX.5 {B3}, 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 confusing, interpreting it as a requirement or permission for these operations to perform some sort of character set conversion. This was never intended, so POSIX.5b {3} 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 {B3} are being phased out in favor of the type `Ada_Streams.Stream_Element_Array`, which is defined in Ada95 (ISO/IEC 8652: 1995). For compatibility with POSIX.5 {B3}, 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_Strings` 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 unmarshalling operations (`'Read` and `'Write`) defined in [Ada95 (ISO/IEC 8652: 1995) 13.13], this also provides 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, such as “+”, 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:

- 1 — A `File_Descriptor` is a value that is returned by `Open` and used as a handle
- 2 by `Read`, `Write`, `Close`, and other subprograms in the Ada binding.
- 3 — An open file description is an object (not directly visible to the application) that
- 4 is traditionally managed inside operating system kernel memory, although it
- 5 might be implemented differently.
- 6 — An external file is the actual file and its contents on some storage media such as
- 7 a disk.

8 It is important to understand the difference between these three objects. In the
9 simplest case, after a successful `Open`, there is one file descriptor, one file description,
10 and one external file. Writing a byte using the file descriptor as handle sends a
11 byte to the system, advances the read/write pointer by 1 in the file description, and
12 eventually writes the byte to disk. In slightly more complicated examples, the same
13 open file description may be referenced by more than one `File_Descriptor`, and
14 the same external file may be referenced by more than one open file description.

15 The case of more than one `File_Descriptor` referencing the same file descrip-
16 tion occurs after `Duplicate`, or `Exec` when the file is not closed upon exec. When
17 two `File_Descriptors` reference the same file description, they share the same
18 read/write pointer, mode (permissions), and flags (options and access modes). For ex-
19 ample, `Read` or `Seek` changes the read/write pointer for both file descriptors. Reading
20 a byte at position 5 on descriptor 1, then seeking to position 10 on descriptor 2, then
21 reading another byte on descriptor 1 results in reading the tenth byte.

22 This standard also allows two file descriptions to reference the same external file.
23 The case of more than one file description referencing the same external file occurs
24 when two separate `Open` calls are made, whether these two calls are made by the
25 same process or by different processes. If two file descriptions reference the same
26 external file and then one is removed (i.e., deleted) with `POSIX_Files.Unlink`, the
27 other still remains open, and the file continues to exist until the last file description
28 is closed. At that time the external file really is removed from the system. I/O
29 operations will be interleaved by the system. This is desirable if both files are being
30 read, or if both files were opened with the `Append` option and are being written. Or it
31 may be desirable if read and write locks are used. Otherwise, it may be a dangerous
32 situation, but this standard does nothing to prevent it.

33 File access modes (`Read_Only`, `Read_Write`, `Write_Only`) and file options (e.g.,
34 `Append`, `Non_Blocking`, `Truncate`, `Exclusive`) are properties of the open file de-
35 scription. They apply to all file descriptors referring to the open file description.
36 Some of these options (e.g., `Non_Blocking`, `Append`) can be read and/or set by the
37 `..._File_Control` operations. When multiple file descriptors refer to the same
38 open file description, when an option (e.g., `Append`) is changed using one file descrip-
39 tor, the change applies to I/O operations on all file descriptors referring to that open
40 file description.

41 Conversely, the same process can open a file twice and obtain two different file de-
42 scriptors referring to two distinct open file descriptions. The file can be open with
43 the `Non_Blocking` option specified for one file description and not specified for the
44 other. Then blocking operations that could not complete immediately would return
45 error code `Resource_Temporarily_Unavailable` on one file descriptor, but block
46 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

There are some combinations of options to `Open` and `Open_Or_Create` that 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 {2} 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, with more added by POSIX.5b {3}. Also, implementations may define still more options. Using these options should be straightforward, using expressions such as:

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 say, -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 {B3} is *not* followed for some close operations added by POSIX.5b {3}, 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 it 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 0 is the same as `dup()`, and with `Bound` greater than 0 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.

1 **B.6.1.3 Read and Write**

2 Read and Write use the actual length of the Buffer parameter as the I/O request
3 length and use an **out** parameter Last (rather than an **out** parameter Length) to
4 return the count of bytes actually transferred. This mirrors the parameter declara-
5 tions in Text_IO.Get_Line. Generic procedures Generic_Read and Generic_
6 Write are provided as a user convenience. An instantiation on an unconstrained
7 type may be rejected by the implementation, as per the Ada rules on generics. The
8 model is that bytes are written, and when the file is read back, bytes equal to the
9 size of Item are read back and put into Item via Unchecked_Conversion. No type
10 checking or reconstruction of unconstrained types occurs, and no array descriptors
11 or record structure information are written into the external file.

12 The error behavior of Generic_Read and Generic_Write is therefore closer to PO-
13 SIX.1 {2} *fread()* and *fwrite()*, respectively, than to POSIX.1 {2} *read()* and *write()*.

14 The fact that Read can raise End_Error is different from POSIX.1 {2} semantics—in
15 POSIX.1 {2} *read()* will just return 0 and *errno* will not be set. This standard reflects
16 the behavior of the language-defined Ada I/O packages in the face of an end-of-file
17 condition.

18 **B.6.1.4 Seek**

19 In early versions of the UNIX system, the C operation *lseek()* was called *seek()*. As
20 file sizes grew beyond 64 K bytes, the parameter changed from the C-language type
21 *int* to type *long*, and *seek()* was renamed *lseek()*. Here it is renamed back to *Seek*,
22 which accurately describes the operation. The C-language operation *lseek()* is split
23 into three subprograms for convenience: *Seek*, *File_Position*, and *File_Size*.
24 *File_Size* might be implemented with several calls to *lseek()* or one call to *fstat()*.

25 **B.6.1.5 File Control**

26 The C *fcntl()* operations for getting and setting modes and flags become more ratio-
27 nal when separated into several functions. This means the Ada caller does not have
28 to worry about separating bits in the result that is returned from *fcntl()*. The file
29 record locking operations from C *fcntl()* are implemented by the separate file locking
30 facilities in package *POSIX_File_Locking*, and the *F_DUPFD* operations are sup-
31 ported by *Duplicate_And_Close*. The current design of *Open_Option_Set* will
32 support use of the file control operations to set and query file options defined by fu-
33 ture POSIX standards. The operation *Get_File_Control* is a procedure with two
34 **out** parameters to keep the two types *File_Mode* and *Open_Option_Set* separate.

35 **B.6.1.6 Update File Status Information**

36 Although *fchmod()* is in Section 5 of POSIX.1b {3}, this binding puts the operation
37 into *POSIX_IO*, to preserve the invariant that only the operations that operate on
38 files by name are placed in *POSIX_Files*, and the operations that operate on files
39 via file descriptors are in Section 6. To do otherwise would also have created a new
40 dependence, of *POSIX_Files* on *POSIX_IO*. These factors were felt to outweigh the
41 functional affinity with *chmod()*.

1 **B.6.1.7 Truncate a File to A Specified Length**

2 Although *ftruncate()* is in Section 5 of POSIX.1b {3}, this binding puts the operation
3 into `POSIX_IO`, to preserve the invariant that only the operations that operate on
4 files by name are placed in `POSIX_Files`, and the operations that operate on files
5 via file descriptors are in Section 6. It would also have created a new dependence, of
6 `POSIX_Files` on `POSIX_IO`.

7 **B.6.1.8 File Synchronization**

8 The interfaces for file synchronization are a very direct mapping to the corresponding
9 interfaces defined in POSIX.1b {3}.

10 **B.6.2 Package `POSIX_File_Locking`**

11 **B.6.2.1 File Locking**

12 POSIX file record locking is advisory. A lock held by one process keeps another pro-
13 cess from setting a conflicting lock. But a lock does not prevent other operations,
14 such as open, read, or write.

15 Locks may be read locks or write locks. They may cover a whole file or a portion of a
16 file. A lock may start or extend beyond the current end of a file, but may not extend
17 before the beginning of a file.

18 The operations of package `POSIX_File_Locking` in the Ada binding split out the
19 behavior of the C function *fcntl()* having to do with file locking. The use of three
20 separate subprograms is believed to be clearer than the corresponding multipurpose
21 C function.

22 This interface does not follow the convention of using private types to hide C struc-
23 tured types, which is followed elsewhere in this standard, as it makes use of a visible
24 discriminated record type `File_Lock`. This visible record has a discriminant that is
25 used to define two different variants, corresponding to whether the lock covers the
26 entire file or only a portion of the file. The rationale for this deviation from the gen-
27 eral practice of this standard (which is to avoid visible record types) dates back to
28 POSIX.5 {B3}, was not documented, and has been forgotten.

29 In C, a file lock of length 0 is used to mean the whole file. In this standard, a variant
30 record is used instead. The C-language record has the process ID as a field, but since
31 this is only used in `Get_Lock`, it has been made into an **out** parameter.

32 Implementors of `Get_Lock` should make a local copy of the `Lock` parameter before
33 using it to guard against aliasing problems if the caller passes the same actual pa-
34 rameter 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. This means that 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 asynchronous I/O. 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, this may result in over-writing arbitrary memory, which 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 asynchronous I/O control block, which is not visible. Creation of the reference and allocation of the control block is performed using `Create_AIO_Control_Block`. Destruction of the reference and deallocation of the control block is 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. This 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 programmer’s responsibility 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 without kernel involvement.

B.6.3.2 Return Status and Error Status

For asynchronous I/O, 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 or not. It is primarily to support use of this function that the error codes

1 `Operation_In_Progress`, `Operation_Completed`, and `Operation_Canceled`
2 were added to package `POSIX`.

3 The `Get_Bytes_Transferred` function binds directly to `aio_return()` and has the
4 limitations documented in the C-language interface: the return value is undefined
5 if the I/O operation is still in progress and `Get_Bytes_Transferred` may be called
6 exactly once for a given asynchronous operation. Therefore `Get_AIO_Status` or
7 `Get_AIO_Error_Code` should be invoked to ensure that a given asynchronous oper-
8 ation is no longer in progress before invoking `Get_Bytes_Transferred`.

9 **B.6.3.3 File Status Flags**

10 In POSIX.1b {3}, `O_DSYNC` or `O_SYNC` is passed to `aio_fsync()` to specify the kind
11 of synchronization desired. There are constants corresponding to `O_DSYNC` and `O_-`
12 `SYNC` in `POSIX_IO`, where they are used as option sets (i.e., flags). However, these
13 constants are not used as flags when they are passed as parameters to the `aio_-`
14 `fsync()` operation. Inconsistent type usage has been avoided by splitting this into two
15 operations.

16 **B.6.3.4 Asynchronous I/O Priority**

17 The `Integer` type is used for the parameter of `Set_Priority_Reduction` and
18 the return value of `Get_Priority_Reduction`. This value is subtracted from the
19 process priority to obtain the priority of the I/O operation. The type is therefore
20 required to be the same as for process priorities, which is `Integer`. Since the value
21 is required to be non-negative, the subtype `Natural` is specified. No upper bound is
22 specified, because the range of allowed priority reductions may not be determinable
23 until run time.

24 **B.6.3.5 Buffers**

25 Buffers for asynchronous I/O operations are specified as explicit Ada access types to
26 the type `Stream_Element_Array`, which is an array type. Where the C-language
27 interface specifies a pointer to an array and a length as separate in-parameters, they
28 are replaced here by a pointer to an Ada unconstrained array; the `Length` attribute
29 of the array actual parameter specifies the length. Note that the actual number of
30 bytes transferred is obtainable via the function `Get_Bytes_Transferred`.

31 **B.6.3.6 AIO Descriptor List**

32 `AIO_Descriptor_List` is defined to be an array of `AIO_Descriptors`. This is the
33 type corresponding to the C `aio_cb` pointer array. It was necessary to split the function
34 `aio_suspend()` into two forms, so as to handle the case of a single control block pointer.
35 The C- language interface only specifies a form for an array of pointers, assuming
36 that a single pointer can be treated as if it were an array of length one. In the Ada
37 binding, two distinct types are involved.

38 Also, it is not specified what happens if the *nent* parameter of `aio_suspend()` is zero
39 or negative. This issue is avoided in the Ada interface by specifying the index range
40 of `AIO_Descriptor_List` to be positive.

B.6.3.6.1 Constraint Error

Because asynchronous I/O operations may require an unpredictable length of time to complete, it is particularly important to initiate asynchronous I/O 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. While this standard does not require it, an implementor can add a measure of safety by marking or invalidating the value of type `AIO_Descriptor` when the AIO Control Block is successfully destroyed.

`POSIX_Error` with an error code of `Invalid_Argument` was selected to correspond to the detection of an attempt to initiate an asynchronous I/O operation on an invalid value of type `AIO_Descriptor`, but implementations may substitute `Constraint_Error` since that may be raised by the runtime 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 asynchronous I/O operation on another copy has an unspecified result.

B.7 Device- and Class-Specific Functions

This section follows very closely the same section in POSIX.1 {2}. 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 comprise 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 section 4.7.1 of POSIX.1 {2}, has also been included in this section as function `Get_Controlling_Terminal_Name`, since this section contains a discussion of the controlling terminal concept.

The only major change from POSIX.1 {2} is to treat the terminal characteristics as a private type and supply the appropriate access and modification functions. Initially, one might feel this will cause much lower performance, but an inline implementation will have the same performance as a structure reference. An unsuccessful attempt was made to develop structure modification operators (“+” and “-”) for manipulating the `Terminal_Characteristics` type. This 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 {2}. 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 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 Sections 5, 6, and 7 of this standard. Herein, these are generally referred to as “POSIX I/O.”
- (2) The Ada I/O operations that are made available to the applications by Annex A of the Ada RM {1}. Herein, these are generally referred to as “Ada I/O.”

The two I/O systems are not intended to have identical objectives. The POSIX I/O system described in Sections 5, 6, and 7 has the objective of making the POSIX I/O model available to the Ada programmer. The Ada I/O system described in this section has the objective of rationalizing the Ada I/O model within the context of POSIX.1 {2}. The Ada I/O system 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 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 binding specified in Sections 5, 6, and 7.

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 the following dimensions or combinations thereof:

- 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:

- 1 – Ada input operations can read data written by Ada output operations.
- 2 – POSIX input operations can read data written by Ada output operations.
- 3 – Ada input operations can read data written by POSIX. output operations
- 4 – Access to POSIX standard input, output, and error files can be achieved via
- 5 Ada I/O operations.

6 In order to circumvent some of the shortcomings of `Text_IO`, the idea of providing a
7 `POSIX_Text_IO` package, which would serve as a functional replacement for `Text_`
8 `IO` and which would incorporate additional or modified functionality, was considered.
9 This concept was rejected because of the expense it would place upon implementors.
10 They would still be required to implement standard Ada `Text_IO` as well as `PO-`
11 `SIX_Text_IO` and the POSIX I/O facilities described in Sections 5 and 6. Requiring
12 the implementation of three I/O systems seemed excessive in a language binding.
13 Moreover, reliance on yet another set of I/O facilities would reduce portability of Ada
14 code to other (non-POSIX) systems.

15 It remains possible to perform formatted I/O on POSIX strings by exploiting the
16 character set mapping defined in Section 2. The functions `POSIX.To_String` and
17 `POSIX.To_POSIX_String` will convert between POSIX strings and the correspond-
18 ing Ada strings. The Ada strings can be processed using normal `Text_IO`.

19 The implementation is required to close Ada files (files opened using `Text_IO` or
20 any instantiation of `Direct_IO` or `Sequential_IO`) upon termination of the Ada
21 program, so that the external files corresponding to the Ada file objects are in a
22 predictable state. POSIX.1 {2} already required that POSIX files be closed when a
23 POSIX process terminates. Because an Ada active partition corresponds to a PO-
24 SIX process, this additional requirement probably amounts only to flushing Ada I/O
25 buffers upon partition termination. Note that this requirement is consistent with
26 the Ada Uniformity Issue UI-0023 {B?}.

27 It is intended that open Ada file objects must be closed when a nonerroneous Ada ap-
28 plication terminates by reaching the end of the main subprogram or by terminating
29 due to an unhandled exception. It is desirable, but not required, that the imple-
30 mentation also close the file objects when an erroneous Ada application terminates
31 or when the corresponding POSIX process dies because it does not catch a POSIX
32 signal.

33 This standard requires that the flushing of output buffers and the closing of open
34 file objects must occur after the termination of the main subprogram of the partition
35 and after the termination of the library units, but prior to the termination of the
36 POSIX process. The reason for this additional standardization is to ensure that the
37 external files corresponding to Ada file objects are left in a predictable state whenever
38 possible.

39 In addition to the flushing required at application termination, the package `POSIX_`
40 `Supplement_to_Ada_IO` provides a mechanism for the application explicitly to flush
41 output files. This mechanism was provided with some reluctance because of the bur-
42 den that it places upon implementors to deal with private types outside of the pack-
43 ages in which the private types were defined. Nevertheless, it is necessary because
44 the execution of `POSIX_Process_Primitives.Start_Process` may cause some
45 ambiguity regarding the status of data that is contained within Ada output buffers,