STANDARDS PROJECT

Draft Standard for Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application Program Interface (API) — Amendment x: Advanced Realtime Extensions [C Language]

Sponsor

Portable Application Standards Committee
of the
IEEE Computer Society

Work Item Number: JTC1 22.21.04.01.01

Abstract: IEEE Std. P1003.1j-199x is part of the POSIX series of standards for applications and user interfaces to open systems. It defines the applications interface to system services for synchronization, memory management, time management, and thread management. This standard is stated in terms of its C binding.

Keywords: API, application portability, C (programming language) data processing, information interchange, open systems, operating system, portable application, POSIX, programming language, realtime, system configuration computer interface

P1003.1j / D10
September 1999

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

September 1999 *SH XXXXX*

Editor's Notes

In addition to your paper ballot, you are also asked to e-mail any balloting comments: please read the balloting instructions document.

This section will not appear in the final document. It is used for editorial comments concerning this draft. Please consult the balloting instructions document and the cover letter for the ballot that accompanied this draft for information on how the balloting process is accomplished.

This draft uses small numbers or letters in the right margin in lieu of change bars. "A" denotes changes from Draft 9 to Draft 10. Trivial informative (i.e., non-normative) changes and purely editorial changes such as grammar, spelling, or cross references are not diff-marked. Changes of function names are not diff-marked either. Since this is a recirculation draft, only normative text marked with "A" is open for comments in this ballot. Revision indicators prior to "8" have been removed from this draft.

Since 1998 there is a new backwards compatibility requirement on the amendments to the base POSIX.1 standard, which states that the base standard will not be changed in such a way as to cause implementations or strictly conforming applications to no longer conform. The implications of this requirement are that no new interface specifications can be included that are not under an option; and that names for new interfaces must begin or end with one of the reserved prefixes or suffixes, including those defined in POSIX.1a. This document incorporates the required changes since draft 7.

Until draft 7, the rationale text for all the sections had been temporarily moved from Annex B and interspersed with the appropriate sections. This co-location of rationale with its accompanying text was done to encourage the Technical Reviewers to maintain the rationale text, as well as provide explanations to the reviewers and balloters. However, in order to better match the final document, all rationale subclauses have been moved back to Annex B in the last recirculations.

Please report typographical errors to:

Michael González Harbour
Dpto. de Electrónica y Computadores
Universidad de Cantabria
Avenida de los Castros s/n
39005 - Santander
SPAIN
TEL: +34 942 201483
FAX: +34 942 201402
Email: mgh@ctr.unican.es

(Electronic mail is preferred.)

The copying and distribution of IEEE balloting drafts is accomplished by the Standards Office. To report problems with reproduction of your copy, or to request additional copies of this draft, contact:

Tracy Woods
IEEE Computer Society,
1730 Massachusetts Avenue, NW,
Washington DC 20036-1992.
Phone: +1-202-371-1013
Fax: +1-202-728-0884
E-mail: twoods@computer.org
Web page: http://www.computer.org/standard/draftstd.htm

POSIX.1j Change History

This section is provided to track major changes between drafts.

Draft 10 [September 1999] Third recirculation of new ballot.

— Changed the treatment of typed memory objects by *fstat*() to make it like the treatment of shared memory objects.

— Various editorial changes, including removal of notes marking text that was conditional on the approval of P1003.1d, because this project has already been approved by the IEEE-SA Standards Board.

Draft 9 [July 1999] Second recirculation of new ballot.

— Because of the Backwards Compatibility requirement, the "Otherwise" clauses in those functions whose names do not start with the "posix_" reserved prefix, were deleted. See "Stubs and Support for Optional Features" in this Editor's Notes, for further information.

— Aligned the text used to describe optional features with the text used in POSIX.1b, POSIX.1c, and POSIX.1d. Option symbols are now used, instead of the associated option names.

— Text was added to specify that copies of synchronization objects cannot be used for synchronization. Only the original objects may be used.

— The two reader/writer-lock unlock operations were collapsed into a single unlock function, to match existing practice in the Single UNIX Specification.

— The clock attribute for condition variables is now under the same option as the *clock_nanosleep*() function; the option has been renamed to {_POSIX_CLOCK_SELECTION}.

Draft 8 [May 1999] First recirculation of new ballot.

— Annex I (Thread Management Considerations), Annex J (Synchronized Clock), and Annex K (Balloting Instructions) were removed from the draft.

— Moved all rationale text into Annex B, where it belongs.

— Moved the "Conventions" and "Normative References" subclauses into these editor's notes, because no amendment to the equivalent subclauses in POSIX.1 was intended.

— Changed the behavior of reader/writer locks when a signal is received, to align it with the current specification for mutexes.

— Changed relative timeouts to absolute, for consistency with the new POSIX.1d timeouts. As a consequence, the amendments to relative timeouts were omitted.

Draft 7 [October 1998] Reballot with new ballot group.

— Added the new backwards compatibility requirement and changed draft accordingly.

Draft 6 [November 1997] First recirculation.

— Merged the process and thread spin locks, and changed all the names of the barrier and reader/writer lock functions to follow the pthreads model.

— Changed the requirements on stubs to resolve balloting objections requesting consistency with POSIX.1c.

— Deleted the Typed Memory Access Management option.

— Moved the typed memory allocation flags out of *mmap*(), into *posix_typed_mem_open*().

— Moved the Thread Abortion chapter to an informative annex.

— Moved the Synchronized Clock to an informative annex.

Draft 5 [May 1995] First balloting round.

— Minor editorial changes, and deletion of all diff marks.

— Changed the requirements on stubs to follow the new SICC policy.

Draft 4 [Apr 1995]

— Added the Monotonic Clock, the Synchronized Clock, and the *nanosleep_rel*() function. Changed relative timeouts to depend on the Monotonic Clock, if present. Added an initialization attribute to condition variables, to specify the clock that shall be used for the timeout service in *pthread_cond_timedwait*().

— Added the Synchronized Clock.

— Added initialization attributes objects to barriers and reader/writer locks, and made some changes to the synchronization functions.

— Some minor changes to typed memory.

Draft 3 [Nov 1994]

— Some changes to the Synchronization Chapter.

— Added the barrier wait, reader/writer lock and spin lock calls to the list of blocking routines that are not cancellation points.

Draft 2 [Sep 1994]

— Added the Thread Abortion Chapter.

— Specified the effects of changing the time of CLOCK_REALTIME.

— Minor technical changes to the synchronization chapter.

Draft 1 [Jul 1994]

— Added new options and definitions to Sections 1 and 2, related to the Synchronization Section.

— Added the Synchronization Section.

— Deleted the placeholder for the Message Passing Section.

Draft 0 [Apr 1994]

— Preliminary draft, prior to PAR approval. Not reviewed by the Working Group.

Stubs and Support for Optional Features

Drafts of POSIX.1j previous to Draft 9 had required that implementations not supporting a specific option must either not provide a function named under the option, or provide that function exactly as specified in the standard. This was stated in the "Otherwise" clause that appeared in every optional function; among other things, this requirement prevented the implementation from providing stubs.

However, Draft 9 has removed the requirement for functions with names which do not begin with the "posix_" reserved prefix. This was done because such names not already specified by POSIX.1 are not reserved for the POSIX standard, and currently conforming implementations may already be providing, as extensions, functions with the same names but different functionality. This is the case, for example, with the reader/writer lock functions defined in the Single UNIX Specification, which are similar, but not identical, to the functions defined in P1003.1j.

If we were to retain the requirement from earlier drafts, such implementations would no longer conform to the POSIX standard, once P1003.1j is approved. But the P1003.1j scope prohibits us from breaking conforming implementations, and thus the requirement had to be removed. The requirement was retained only for those functions with the "posix_" prefix because, since this prefix is reserved for the POSIX standard (by P1003.1a), no conforming implementation can provide a function with such a name.

As a consequence, any new objection requesting that we restore the "Otherwise" clauses for those optional functions not starting with the "posix_" prefix, would be against the scope of the P1003.1j standards project, and would have to be considered as "unresponsive". Please note that the inconsistencies that exist in the POSIX standard with regard to optional functions and stubs will be harmonized during the POSIX revision process currently underway.

Normative References

NOTE: This standard does not amend subclause 1.2, Normative References, of ISO/IEC 9945-1:1996. However, the Normative References of ISO/IEC 9945-1:1996 are repeated here for information. In addition, since IEEE P1003.1j modifies ISO/IEC 9945-1:1996 as amended by IEEE 1003.1d:1999 (and by IEEE P1003.1a, if approved before this standard), we have included the latter

two among this informal list of references.

The following standards contain provisions which, through references in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

{1} ISO/IEC 9899: 1995[1)], *Information processing systems—Programming languages—C.*

{2} ISO/IEC 9945-1: 1996 (IEEE Std 1003.1-1996), *Information technology—Portable operating system interface (POSIX)—Part 1: System application program interface (API) [C Language].*

{3} IEEE Draft Std. P1003.1a, Draft 14, January 1998, *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] — Amendment*

{4} IEEE Std 1003.1d:1999, *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] — Amendment x: Additional Realtime Extensions*

{5} IEEE Std 610-1990, *IEEE Standard Computer Dictionary — A Compilation of IEEE Standard Computer Glossaries*

Conventions

NOTE: This standard does not amend subclause 2.1, Conventions, of ISO/IEC 9945-1:1996. However, we repeat this subclause here for information.

This document uses the following typographic conventions:

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

— Cross references to defined terms within 1.3, 2.2.1, and 2.2.2; symbolic parameters that are generally substituted with real values by the application

— C language data types and function names (except in function Synopsis subclauses)

— Global external variable names

— Function families; references to groups of closely related functions (such as *directory*, *exec*, *sigsetops*, *sigwait*, *stdio*, and *wait*)

---

1) ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

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

PATH

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

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

— For C language data types and function names within function Synopsis subclauses

— To illustrate examples of system input or output where exact usage is depicted

— For references to utility names and C language headers

— For names of attributes in attributes objects

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

[ERRNO]

See 2.4.

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

{LIMIT}

See 2.8 and 2.9.

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

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

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

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

POSIX.1j Technical Reviewers

The individuals denoted in Table i are the Technical Reviewers for this draft. During balloting they are the subject matter experts who coordinate the resolution process for specific sections, as shown.

Table i — POSIX.1j Technical Reviewers

Section	Description	Reviewer
	Ballot Coordinators	Joe Gwinn and Jim Oblinger
11.5-7	*Synchronization*	Karen Gordon and Michael González
5,6,8,12	*Typed Memory*	Frank Prindle
3,6.7,11.2-4,14,15	*Monotonic Clock and Nanosleep*	Michael González

Contents

FIGURES

TABLES

Introduction

(This introduction is not a normative part of P1003.1j, Draft Standard for Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application Program Interface (API) — Amendment x: Advanced Realtime Extensions [C Language], but is included for information only.)

Editor's Note: This Introduction consists of material that will eventually be integrated into the base POSIX.1 standard's introduction (and the portion of Annex B that contains general rationale about the standard). The Introduction contains text that was previously held in either the Foreword or Scope. As this portion of the standard is for information only (nonnormative), specific details of the integration with POSIX.1 are left as an editorial exercise.

The purpose of this document is to supplement the base standard with interfaces and functionality for applications having realtime requirements or special efficiency requirements in tightly coupled multitasking environments.

This standard will not change the base standard which it amends (including any existing amendments) in such a way as to cause implementations or strictly conforming applications to no longer conform.

This standard defines systems interfaces to support the source portability of applications with realtime requirements. The system interfaces are all extensions of or additions to ISO/IEC 9945-1:1996, as amended by IEEE-1003.1d:1999 (and by IEEE 1003.1a, if approved before this standard). Although rooted in the culture defined by ISO/IEC 9945-1: 1990, they are focused upon the realtime application requirements, and the support of multiple threads of control within a process, which were beyond the scope of ISO/IEC 9945-1: 1990. The interfaces included in this standard were the set required to make ISO/IEC 9945-1: 1990 efficiently usable to realtime applications or applications running in multiprocessor systems with requirements that were not covered by the realtime or threads extensions specified in IEEE-1003.1b, IEEE-1003.1c, and IEEE-1003.1d. The scope is to take existing realtime or multiprocessor operating system practice and add it to the base standard.

The definition of *realtime* used in defining the scope of this standard is:

> Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time.

The key elements of defining the scope are a) defining a sufficient set of functionality to cover the realtime application program domain in the areas not covered by IEEE-1003.1b, IEEE-1003.1c, and IEEE-1003.1d; b) defining a sufficient set of functionality to cover efficient synchronization in multiprocessors that allows applications to achieve the performance benefits of such architectures; c) defining sufficient performance constraints and performance-related functions to allow a realtime application to achieve deterministic response from the system; and d) specifying changes or additions to improve or complete the definition of the facilities specified in the previous real-time or threads extensions IEEE-1003.1b, IEEE-1003.1c, and IEEE-1003.1d.

Wherever possible, the requirements of other application environments were included in the interface definition. The specific areas are noted in the scope overviews of each of the interface areas given below.

The specific functional areas included in this standard and their scope include:

- Synchronization: new synchronization primitives that allow multiprocessor applications to achieve the performance benefits of their hardware architecture.
- Memory management: a facility to allow programs to allocate or access different kinds of physical memory that are present in the system, and allow separate application programs to share portions of this memory.
- Clocks and Timers: the addition of the Monotonic Clock, the specification of the effects of setting the time of a clock on other timing services, and the addition of functions to support relative or absolute suspension based upon a clock specified by the application.

This standard has been defined exclusively at the source code level, for the C programming language. Although the interfaces will be portable, some of the parameters used by an implementation may have hardware or configuration dependencies.

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:[2)]

(1) Additional System Application Program Interfaces in C Language

(2) Ada language bindings to this standard

(3) Shell and utility facilities

(4) Verification testing methods

(5) Tracing facilities

(6) Fault tolerance

(7) Checkpoint/restart facilities

(8) Resource limiting facilities

(9) Network interface facilities

(10) System administration

(11) Profiles describing application- or user-specific combinations of Open Systems standards

(12) An overall guide to POSIX-based or related Open Systems standards and profiles

Extensions are approved as "amendments" or "revisions" to this document, following the IEEE and ISO/IEC Procedures.

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

If you have 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, and ask to have this forwarded to the chairperson of the appropriate PASC working group. If you have interest in participating in this work at the international level, contact your ISO/IEC national body.

---

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

P1003.1j was prepared by the System Services Working Group—Realtime, sponsored by the Portable Application Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the System Services Working Group was as follows:

Portable Application Standards Committee (PASC)

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

System Services Working Group—Realtime: Officials

Chair:	Joe Gwinn
	Susan Corwin (until 1995)
Editor:	Michael González
Secretary:	Karen Gordon
	Lee Schemerhorn (until 1995)

Ballot Coordinators

Joe Gwinn Jim Oblinger

Technical Reviewers

Michael González Karen Gordon Frank Prindle

Working Group

to be supplied to be supplied to be supplied

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

Institutional Representatives *<To be filled in>*

Individual Balloters *<To be filled in>*

When the IEEE Standards Board approved this standard on <*date to be provided*>, it had the following membership:

(to be pasted in by IEEE)

Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application Program Interface (API) — Amendment x: Advanced Realtime Extensions [C Language]

Section 1: General

1.3 Conformance

1.3.1 Implementation Conformance

⇒ **1.3.1.3 Conforming Implementation Options** *Add to the table of implementation options that warrant requirement by applications or in specifications:*

{_POSIX_BARRIERS}	Barriers option in (2.9.3)
{_POSIX_CLOCK_SELECTION}	Clock Selection option (in 2.9.3)
{_POSIX_MONOTONIC_CLOCK}	Monotonic Clock option (in 2.9.3)
{_POSIX_READER_WRITER_LOCKS}	Reader/Writer Locks option in (2.9.3)
{_POSIX_SPIN_LOCKS}	Spin Locks option (in 2.9.3)
{_POSIX_TYPED_MEMORY_OBJECTS}	Typed Memory Objects option (in 2.9.3)

Section 2: Terminology and General Requirements

2.2 Definitions

2.2.2 General Terms

⇒ **2.2.2 General Terms** *Modify the definition of* "memory object" *replacing it with the following text:*

2.2.2.63 memory object: Either a file, a shared memory object, or a typed memory object.

When used in conjunction with *mmap*(), a memory object will appear in the address space of the calling process.

⇒ **2.2.2 General Terms** *Modify the contents of subclause 2.2.2, General Terms, to add the following definitions in the correct sorted order [disregarding the subclause numbers shown here].*

2.2.2.133 barrier: A synchronization object that allows multiple threads to synchronize at a particular point in their execution.

2.2.2.134 clock jump: The difference between two successive distinct values of a clock, as observed from the application via one of the "get time" operations.

2.2.2.135 monotonic clock: A clock whose value cannot be set via *clock_settime*() and which cannot have negative clock jumps.

2.2.2.136 reader/writer lock: A synchronization object that allows a group of threads, called "readers", simultaneous read access to a resource and another group, called "writers", exclusive write access to the resource. All readers exclude any writers and a writer excludes all readers and any other writers.

2.2.2.137 spin lock: A synchronization object used to allow multiple threads to serialize their access to shared data.

2.2.2.138 typed memory namespace: A system-wide namespace that contains the names of the typed memory objects present in the system. It is configurable for a given implementation.

2.2.2.139 typed memory object: A combination of a typed memory pool and a typed memory port. The entire contents of the pool shall be accessible from the port. The typed memory object is identified through a name that belongs to the typed memory namespace.

2.2.2.140 typed memory pool: An extent of memory with the same operational characteristics. Typed memory pools may be contained within each other.

2.2.2.141 typed memory port: A hardware access path to one or more typed memory pools.

2.5 Primitive System Data Types

⇒ **2.5 Primitive System Data Types** *Add the following text at the end of the first paragraph, starting "Some data types used by..."*

Support for some primitive data types is dependent on implementation options (see Table 2-2). Where an implementation option is not supported, the primitive data types for that option need not be found in the header <sys/types.h>.

⇒ **2.5 Primitive System Data Types** *In the second paragraph, replace* "All of the types listed in Table 2-1 ..." *by the following:*

"All of the types listed in Table 2-1 and Table 2-2 ..."

⇒ **2.5 Primitive System Data Types** *Add the following datatypes to the list of types for which there are no defined comparison or assignment operations:*

pthread_barrier_t, *pthread_barrierattr_t*, *pthread_rwlock_t*, *pthread_rwlockattr_t*, *pthread_spinlock_t*.

⇒ **2.5 Primitive System Data Types** *Add the following paragraphs after the paragraph starting* "There are no defined comparison...":

An implementation need not provide the types *pthread_barrier_t* and *pthread_barrierattr_t* unless the Barriers option is supported (see 2.9.3).

An implementation need not provide the types *pthread_rwlock_t* and *pthread_rwlockattr_t* unless the Reader Writer Locks option is supported (see 2.9.3).

An implementation need not provide the type *pthread_spinlock_t* unless the Spin Locks option is supported (see 2.9.3).

⇒ **2.5 Primitive System Data Types** *Add the following table, and renumber subsequent tables in this Section accordingly:*

Table 2-2 – Optional Primitive System Data Types

Defined Type	Description	Implementation Option
pthread_barrier_t	Used to identify a barrier	Barriers option
pthread_barrierattr_t	Used to define a barrier attributes object	Barriers option
pthread_rwlock_t	Used to identify a reader/writer lock	Reader Writer Locks option
pthread_rwlockattr_t	Used to define a reader/writer lock attributes object	Reader Writer Locks option
pthread_spinlock_t	Used to identify a spin lock	Spin Locks option

2.7 C Language Definitions

2.7.3 Headers and Function Prototypes

⇒ **2.7.3 Headers and Function Prototypes** *Add the following text after the sentence* "For other functions in this part of ISO/IEC 9945, the prototypes or declarations shall appear in the headers listed below.":

Presence of some prototypes or declarations is dependent on implementation options. Where an implementation option is not supported, the prototype or declaration need not be found in the header.

⇒ **2.7.3 Headers and Function Prototypes** *Modify the contents of subclause 2.7.3 to add the following optional functions, at the end of the current list of headers and functions.*

If the Typed Memory Objects option is supported:

`<sys/mman.h>`	*posix_typed_mem_open*(), *posix_mem_offset*(), *posix_typed_mem_get_info*()

If the Spin Locks option is supported:

`<pthread.h>`	*pthread_spin_init*(), *pthread_spin_destroy*(), *pthread_spin_lock*(), *pthread_spin_trylock*(), *pthread_spin_unlock*()

If the Barriers option is supported:

`<pthread.h>`	*pthread_barrierattr_init*(), *pthread_barrierattr_destroy*(), *pthread_barrierattr_getpshared*(), *pthread_barrierattr_setpshared*(), *pthread_barrier_init*(), *pthread_barrier_destroy*(), *pthread_barrier_wait*()

If the Reader/Writer Locks option is supported:

`<pthread.h>`	*pthread_rwlockattr_init*(), *pthread_rwlockattr_destroy*(), *pthread_rwlockattr_getpshared*(), *pthread_rwlockattr_setpshared*(), *pthread_rwlock_init*(), *pthread_rwlock_destroy*(), *pthread_rwlock_rdlock*(), *pthread_rwlock_tryrdlock*(), *pthread_rwlock_timedrdlock*(), *pthread_rwlock_wrlock*(), *pthread_rwlock_trywrlock*(), *pthread_rwlock_timedwrlock*(), *pthread_rwlock_unlock*()

If the Clock Selection option is supported:

`<time.h>`	*clock_nanosleep*()
`<pthread.h>`	*pthread_condattr_setclock*(), *pthread_condattr_getclock*()

2.8 Numerical Limits

2.8.7 Maximum Values

⇒ **2.8.7 Maximum Values** *In Table 2-7a, replace the description of* {_POSIX_CLOCKRES_MIN}, *currently reading* "The CLOCK_REALTIME clock resolution, in nanoseconds", *with the following:*

The resolution of the clocks CLOCK_REALTIME and CLOCK_MONOTONIC (if supported), in nanoseconds

2.9 Symbolic Constants

2.9.3 Compile-Time Symbolic Constants for Portability Specifications

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** *Change the first words in the first paragraph, currently saying* "The constants in Table 2-10 may be used..." *to the following:*

The constants in Table 2-10 and Table 2-11 may be used...

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** *Add the following sentence at the end of the first paragraph:*

If any of the constants in Table 2-11 is defined, it shall be defined with the value shown in that Table. This value represents the version of the associated option that is supported by the implementation.

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** *Add* Table 2-11, *shown below, after Table 2-10 renumbering all subsequent tables accordingly.*

NOTE: (Editor's note) The value `199ymmL` corresponds to the date of approval of IEEE P1003.1j.

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** *Add the following paragraphs:*

If the symbol {_POSIX_BARRIERS} is defined, then the symbols {_POSIX_THREADS} and {_POSIX_THREAD_SAFE_FUNCTIONS} shall also be defined. If the symbol {_POSIX_READER_WRITER_LOCKS} is defined, then the symbols {_POSIX_THREADS} and {_POSIX_THREAD_SAFE_FUNCTIONS} shall also be defined. If the symbol {_POSIX_SPIN_LOCKS} is defined, then the symbols {_POSIX_THREADS} and {_POSIX_THREAD_SAFE_FUNCTIONS} shall also be defined.

If the symbol {_POSIX_MONOTONIC_CLOCK} is defined, then the symbol {_POSIX_TIMERS} shall also be defined.

If the symbol {_POSIX_CLOCK_SELECTION} is defined, then the symbol {_POSIX_TIMERS} shall also be defined.

Table 2-11 – Versioned Compile-Time Symbolic Constants

Name	Value	Description
{_POSIX_BARRIERS}	199ymmL	If this symbol is defined with the shown value, the implementation supports the Barriers option.
{_POSIX_READER_WRITER_LOCKS}	199ymmL	If this symbol is defined with the shown value, the implementation supports the Reader/Writer Locks option.
{_POSIX_SPIN_LOCKS}	199ymmL	If this symbol is defined with the shown value, the implementation supports the Spin Locks option.
{_POSIX_TYPED_MEMORY_OBJECTS}	199ymmL	If this symbol is defined with the shown value, the implementation supports the Typed Memory Objects option.
{_POSIX_MONOTONIC_CLOCK}	199ymmL	If this symbol is defined with the shown value, the implementation supports the Monotonic Clock option.
{_POSIX_CLOCK_SELECTION}	199ymmL	If this symbol is defined with the shown value, the implementation supports the Clock Selection option.

Section 3: Process Primitives

3.1 Process Creation and Execution

3.1.2 Execute a File

⇒ **3.1.2.2 Execute a File—Description** *Add the following paragraph after the paragraph starting* "If the Memory Mapped Files or Shared Memory Objects option ..."

If the Typed Memory Objects option is supported, blocks of typed memory that were mapped in the calling process are unmapped, as if *munmap*() was implicitly called to unmap them.

3.2 Process Termination

3.2.2 Terminate a Process

⇒ **3.2.2.2 Terminate a Process—Description** *Add the following list item after item number (11), and renumber the subsequent items accordingly:*

(12) If the Typed Memory Objects option is supported, blocks of typed memory that were mapped in the calling process are unmapped, as if *munmap*() was implicitly called to unmap them.

3.3 Signals

3.3.8 Synchronously Accept a Signal

⇒ **3.3.8.2 Synchronously Accept a Signal—description** *Add the following text at the end of the paragraph starting* "The function *sigtimedwait*() behaves the same as ..."

If the Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to measure the time interval specified by the *timeout* argument.

Section 4: Process Environment

4.8 Configurable System Variables

4.8.1 Get Configurable System Variables

⇒ **4.8.1.2 Get Configurable System Variables— Description** *Add the following text after the sentence* "The implementation shall support all of the variables listed in Table 4-2 and may support others"*, in the second paragraph:*

Support for some configuration variables is dependent on implementation options (see Table 4-3). Where an implementation option is not supported, the variable need not be supported.

⇒ **4.8.1.2 Get Configurable System Variables— Description** *In the second paragraph, replace the text* "The variables in Table 4-2 come from ..." *by the following:*

"The variables in Table 4-2 and Table 4-3 come from ..."

⇒ **4.8.1.2 Get Configurable System Variables— Description** *Add the following table:*

Table 4-3 – Optional Configurable System Variables

Variable	*name* Value
{_POSIX_BARRIERS}	{_SC_BARRIERS}
{_POSIX_READER_WRITER_LOCKS}	{_SC_READER_WRITER_LOCKS}
{_POSIX_SPIN_LOCKS}	{_SC_SPIN_LOCKS}
{_POSIX_TYPED_MEMORY_OBJECTS}	{_SC_TYPED_MEMORY_OBJECTS}
{_POSIX_MONOTONIC_CLOCK}	{_SC_MONOTONIC_CLOCK}
{_POSIX_CLOCK_SELECTION}	{_SC_CLOCK_SELECTION}

Section 5: Files and Directories

5.6 File Characteristics

5.6.1 File Characteristics: Header and Data Structure

⇒ **5.6.1.1 <sys/stat.h> File Types** *Add the following text and macro after S_TYPEISSHM:*

If the Typed Memory Objects option is supported, the implementation may implement typed memory objects as distinct file types, and the following macro shall test whether a file is of the specified type:

S_TYPEISTMO(*buf*) Test macro for a typed memory object

5.6.2 Get File Status

⇒ **5.6.2.2 Get File Status—Description** *Replace the text "*If the Shared Memory Objects option is supported and *fildes* references a shared memory object,*" by the following:*

If the Shared Memory Objects option is supported and *fildes* references a shared memory object or the Typed Memory Objects option is supported and *fildes* references a typed memory object,

5.6.4 Change File Modes

⇒ **5.6.4.2 Change File Modes—Description** *Add the following paragraph before the paragraph starting "*If the calling process does not have appropriate privileges...*":*

If {_POSIX_TYPED_MEMORY_OBJECTS} is defined and *fildes* references a typed memory object, the behavior of *fchmod*() is unspecified.

Section 6: Input and Output Primitives

6.3 File Descriptor Deassignment

6.3.1 Close a File

⇒ **6.3.1.2 Close a File—Description** *In the close() function, replace the paragraph starting* "If a memory object remains referenced..." *by the following:*

> If a shared memory object or a memory mapped file 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 shared memory object or a memory mapped file and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object shall be removed.

6.4 Input and Output

6.4.1 Read from a File

⇒ **6.4.1.2 Read from a File—description** *Add the following text at the end of the description of the read() function:*

If the Typed Memory Objects option is supported:

> If *fildes* refers to a typed memory object, the result of the *read*() function is unspecified.

6.4.2 Write to a File

⇒ **6.4.2.2 Write to a File—Description** *Add the following text at the end of the description of the write() function:*

If the Typed Memory Objects option is supported:

If *fildes* refers to a typed memory object, the result of the *write*() function is unspecified.

6.5 Control Operations on Files

6.5.2 File Control

⇒ **6.5.2.2 File Control—Description** *Add the following text at the end of the description of the fcntl() function:*

If the Typed Memory Objects option is supported and *fildes* refers to a typed memory object, the result of the *fcntl*() function is unspecified.

6.5.3 Reposition Read/Write File Offset

⇒ **6.5.3.2 Reposition Read/Write File Offset—Description** *Add the following text at the end of the description of the lseek() function:*

If the Typed Memory Objects option is supported and *fildes* refers to a typed memory object, the result of the *lseek*() function is unspecified.

6.7 Asynchronous Input and Output

6.7.8 Wait for an Asynchronous I/O Request

⇒ **6.7.8.2 Wait for an Asynchronous I/O Request—Description** *In the description of the aio_suspend() function, add the following text at the end of the paragraph starting* "If the time interval indicated in ...":

If {_POSIX_MONOTONIC_CLOCK} is defined, the clock that shall be used to measure this time interval shall be the CLOCK_MONOTONIC clock. 9

Section 8: Language-Specific Services for the C Language

8.2 C Language Input/Output Functions

8.2.2 Open a Stream on a File Descriptor

⇒ **8.2.2.2 Open a Stream on a File Descriptor—Description** *Add the following text at the end of the description of the fdopen() function:*

If the Typed Memory Objects option is supported and *fildes* refers to a typed memory object, the result of the *fdopen*() function is unspecified.

Section 11: Synchronization

11.4 Condition Variables

11.4.1 Condition Variable Initialization Attributes

⇒ **11.4.1.1 Condition Variable Initialization Attributes—Synopsis** *Add the following function synopses:*

```
int pthread_condattr_getclock(const pthread_condattr_t *attr,
                              clockid_t *clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
                              clockid_t clock_id);
```

⇒ **11.4.1.2 Condition Variable Initialization Attributes—Description** *Add the following text before the "*Otherwise*" clause:*

If {_POSIX_CLOCK_SELECTION} is defined, the implementation shall provide the `clock` attribute and the associated functions *pthread_condattr_setclock*() and *pthread_condattr_getclock*(). The `clock` attribute is the clock id of the clock that shall be used to measure the timeout service of *pthread_cond_timedwait*(). The default value of the `clock` attribute shall refer to the system clock.

The *pthread_condattr_setclock*() function is used to set the `clock` attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock*() is called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail. The *pthread_condattr_getclock*() function obtains the value of the `clock` attribute from the attributes object referenced by *attr*.

⇒ **11.4.1.2 Condition Variable Initialization Attributes—Description** *Add the pthread_condattr_getclock() and pthread_condattr_setclock() functions to the "*Otherwise*" list.*

⇒ **11.4.1.3 Condition Variable Initialization Attributes—Returns** *Add the pthread_condattr_setclock() function to the list of functions appearing in the first paragraph. In addition, add the following paragraph:*

If successful, the *pthread_condattr_getclock*() function shall return zero and store the value of the `clock` attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error number shall be returned to indicate the error.

⇒ **11.4.1.4 Condition Variable Initialization Attributes—Errors** *Add the pthread_condattr_setclock() and pthread_condattr_getclock() functions to the list of functions for which the error value* [EINVAL] *is returned if the implementation detects that the value specified by attr is invalid. In addition, add the following text at the end of this subclause:*

For each of the following conditions, if the condition is detected, the *pthread_condattr_setclock*() function shall return the corresponding error number:

[EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-time clock.

⇒ **11.4.1.5 Condition Variable Initialization Attributes—Cross-References** *Add the following cross-reference:*

pthread_cond_timedwait(), 11.4.4.

11.4.4 Waiting on a Condition

⇒ **11.4.4.2 Waiting on a Condition—Description** *add the following text after the sentence starting "*The *pthread_cond_timedwait* function is the same as ...*":*

If {_POSIX_CLOCK_SELECTION} is defined, the condition variable shall have a `clock` attribute which specifies the clock that shall be used to measure the time specified by the *abstime* argument.

⇒ **11 Synchronization** *Add these subclauses:*

11.5 Barriers

11.5.1 Barrier Initialization Attributes

Functions: *pthread_barrierattr_init*(), *pthread_barrierattr_destroy*(), *pthread_barrierattr_getpshared*(), *pthread_barrierattr_setpshared*().

11.5.1.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *attr,
                                   int *pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                   int pshared);
```

11.5.1.2 Description

If {_POSIX_BARRIERS} is defined:

The function *pthread_barrierattr_init*() initializes a barrier attributes object *attr* with the default value for all of the attributes defined by the implementation.

The results are undefined if *pthread_barrierattr_init*() is called specifying an already initialized barrier attributes object.

After a barrier attributes object has been used to initialize one or more barriers, any function affecting the attributes object (including destruction) does not affect any previously initialized barrier.

The *pthread_barrierattr_destroy*() function destroys a barrier attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to *pthread_barrierattr_init*(). An implementation may cause *pthread_barrierattr_destroy*() to set the object referenced by *attr* to an invalid value.

If {_POSIX_THREAD_PROCESS_SHARED} is defined, the implementation shall provide the attribute `process-shared` and the associated functions *pthread_barrierattr_getpshared*() and *pthread_barrierattr_setpshared*(). The `process-shared` attribute is set to PTHREAD_PROCESS_SHARED to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the `process-shared` attribute is

PTHREAD_PROCESS_PRIVATE, the barrier shall only be operated upon by threads created within the same process as the thread that initialized the barrier; if threads of different processes attempt to operate on such a barrier, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE. Both constants PTHREAD_PROCESS_SHARED and PTHREAD_PROCESS_PRIVATE are defined in <pthread.h>.

The *pthread_barrierattr_setpshared*() function is used to set the process-shared attribute in an initialized attributes object referenced by *attr*. The *pthread_barrierattr_getpshared*() function obtains the value of the process-shared attribute from the attributes object referenced by *attr*.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation defined.

11.5.1.3 Returns

If successful, the *pthread_barrierattr_init*(), *pthread_barrierattr_destroy*(), and *pthread_barrierattr_setpshared*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_barrierattr_getpshared*() function shall return zero and store the value of the process-shared attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

11.5.1.4 Errors

If any of the following conditions occur, the *pthread_barrierattr_init*() function shall return the corresponding error value:

[ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

For each of the following conditions, if the condition is detected, the *pthread_barrierattr_destroy*(), *pthread_barrierattr_getpshared*(), and *pthread_barrierattr_setpshared*() functions shall return the corresponding error value:

[EINVAL] The value specified by *attr* is invalid.

For each of the following conditions, if the condition is detected, the *pthread_barrierattr_setpshared*() function shall return the corresponding error value:

[EINVAL] The new value specified for the process-shared attribute is not one of the legal values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

11.5.1.5 Cross-References

pthread_barrier_init(), 11.5.2.

11.5.2 Initialize/Destroy a Barrier

Functions: *pthread_barrier_init*(), *pthread_barrier_destroy*().

11.5.2.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_barrierattr_t *attr,
                         unsigned int count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

11.5.2.2 Description

If {_POSIX_BARRIERS} is defined:

The *pthread_barrier_init*() function shall allocate any resources required to use the barrier referenced by *barrier* and initializes the barrier with attributes referenced by *attr*. If *attr* is NULL, the default barrier attributes are used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if *pthread_barrier_init*() is called when any thread is blocked on the barrier (that is, has not returned from the *pthread_barrier_wait*() call). The results are undefined if a barrier is used without first being initialized. The results are undefined if *pthread_barrier_init*() is called specifying an already initialized barrier.

The *count* argument specifies the number of threads that must call *pthread_barrier_wait*() before any of them successfully return from the call. The value specified by *count* must be greater than zero.

If the *pthread_barrier_init*() function fails, the barrier is not initialized and the contents of *barrier* are undefined.

Only the object referenced by *barrier* may be used for performing synchronization. The result of referring to copies of that object in calls to *pthread_barrier_destroy*() or *pthread_barrier_wait*() is undefined.

The *pthread_barrier_destroy*() function destroys the barrier referenced by *barrier* and releases any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is re-initialized by another call to *pthread_barrier_init*(). An implementation may use this function to set *barrier* to an invalid value. The results are undefined if *pthread_barrier_destroy*() is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

11.5.2.3 Returns

Upon successful completion, the *pthread_barrier_init*() and *pthread_barrier_destroy*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.5.2.4 Errors

If any of the following conditions occur, the *pthread_barrier_init*() function shall return the corresponding value:

[EAGAIN] The system lacks the necessary resources to initialize another barrier.

[EINVAL] The value specified by *count* is equal to zero.

[ENOMEM] Insufficient memory exists to initialize the barrier.

For each of the following conditions, if the condition is detected, the *pthread_barrier_init*() function shall return the corresponding value:

[EBUSY] The implementation has detected an attempt to reinitialize a barrier while it is in use (for example, while being used in a *pthread_barrier_wait*() call) by another thread.

[EINVAL] The value specified by *attr* is invalid.

For each of the following conditions, if the condition is detected, the *pthread_barrier_destroy*() function shall return the corresponding value:

[EBUSY] The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a *pthread_barrier_wait*() call) by another thread.

[EINVAL] The value specified by *barrier* is invalid.

11.5.2.5 Cross-References

pthread_barrier_wait(), 11.5.3.

11.5.3 Synchronize at a Barrier

Functions: *pthread_barrier_wait*().

11.5.3.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

11.5.3.2 Description

If {_POSIX_BARRIERS} is defined:

The *pthread_barrier_wait*() function synchronizes participating threads at the barrier referenced by *barrier*. The calling thread blocks (that is, does not return from the *pthread_barrier_wait*() call) until the required number of threads have called *pthread_barrier_wait*() specifying the barrier.

When the required number of threads have called *pthread_barrier_wait*() specifying the barrier, the constant PTHREAD_BARRIER_SERIAL_THREAD is returned to one unspecified thread and zero is returned to each of the remaining threads. At this point, the barrier is reset to the state it had as a result of the most recent *pthread_barrier_init*() function that referenced it.

The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in <pthread.h> and its value is distinct from any other value returned by *pthread_barrier_wait*().

The results are undefined if this function is called with an uninitialized barrier.

If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the required number of threads have not arrived at the barrier during the execution of the signal handler); otherwise, the thread shall continue as normally from the completed barrier wait. Until the thread in the signal handler returns from it, it is unspecified whether other threads may proceed past the barrier once they have all reached it.

A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources shall be determined by the scheduling policy. See 13.2 for full details.

11.5.3.3 Returns

Upon successful completion, the *pthread_barrier_wait*() function shall return PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the barrier and zero for each of the other threads. Otherwise, an error number shall be returned to indicate the error.

11.5.3.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_barrier_wait*() function shall return the corresponding value:

[EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

11.5.3.5 Cross-References

pthread_barrier_init(), 11.5.2; *pthread_barrier_destroy*(), 11.5.2.

11.6 Reader/Writer Locks

Some of the synchronization primitives defined in this section provide exclusive access to a resource. An application may also want to allow a group of threads, called readers, simultaneous read access to a resource and another group of threads, called writers, exclusive write access to the resource. To do so, another synchronization primitive called a multiple reader/single writer, or reader/writer, lock can be used.

One or more readers acquire read access to the resource by performing a read lock operation on the associated reader/writer lock. A writer acquires exclusive write access by performing a write lock operation. Basically, all readers exclude any writers and a writer excludes all readers and any other writers.

A thread that has blocked on a reader/writer lock (that is, has not yet returned from a *pthread_rwlock_rdlock*() or *pthread_rwlock_wrlock*() call) shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources shall be determined by the scheduling policy. See 13.2 for full details.

11.6.1 Reader/Writer Lock Initialization Attributes

Functions: *pthread_rwlockattr_init*(), *pthread_rwlockattr_destroy*(), *pthread_rwlockattr_getpshared*(), *pthread_rwlockattr_setpshared*().

11.6.1.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
                                  int *pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared);
```

11.6.1.2 Description

If {_POSIX_READER_WRITER_LOCKS} is defined:

The function *pthread_rwlockattr_init*() initializes a reader/writer lock attributes object *attr* with the default value for all of the attributes defined by the implementation.

The results are undefined if *pthread_rwlockattr_init*() is called specifying an already initialized reader/writer lock attributes object.

After a reader/writer lock attributes object has been used to initialize one or more reader/writer locks, any function affecting the attributes object (including destruction) does not affect any previously initialized reader/writer lock.

The *pthread_rwlockattr_destroy*() function destroys a reader/writer lock attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to *pthread_rwlockattr_init*(). An implementation may cause *pthread_rwlockattr_destroy*() to set the object referenced by *attr* to an invalid value.

If {_POSIX_THREAD_PROCESS_SHARED} is defined, the the implementation shall provide the attribute `process-shared` and the associated functions *pthread_rwlockattr_getpshared*() and *pthread_rwlockattr_setpshared*(). If this option is not supported, then the `process-shared` attribute and these functions are not supported. The `process-shared` attribute is set to PTHREAD_PROCESS_SHARED to permit a reader/writer lock to be operated upon by any thread that has access to the memory where the reader/writer lock is allocated. If the `process-shared` attribute is PTHREAD_PROCESS_PRIVATE, the reader/writer lock shall only be operated upon by threads created within the same process as the thread that initialized the reader/writer lock; if threads of different processes attempt to operate on such a reader/writer lock, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE.

The *pthread_rwlockattr_setpshared*() function is used to set the `process-shared` attribute in an initialized attributes object referenced by *attr*. The *pthread_rwlockattr_getpshared*() function obtains the value of the `process-shared` attribute from the attributes object referenced by *attr*.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation defined.

11.6.1.3 Returns

If successful, the *pthread_rwlockattr_init*(), *pthread_rwlockattr_destroy*(), and *pthread_rwlockattr_setpshared*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_rwlockattr_getpshared*() function shall return zero and store the value of the `process-shared` attribute of *attr* into the object

referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

11.6.1.4 Errors

If any of the following conditions occur, the *pthread_rwlockattr_init*() function shall return the corresponding error number:

[ENOMEM] Insufficient memory exists to initialize the reader/writer lock attributes object.

For each of the following conditions, if the condition is detected, the *pthread_rwlockattr_destroy*(), *pthread_rwlockattr_getpshared*(), and *pthread_rwlockattr_setpshared*() functions shall return the corresponding error number:

[EINVAL] The value specified by *attr* is invalid.

For each of the following conditions, if the condition is detected, the *pthread_rwlockattr_setpshared*() function shall return the corresponding error number:

[EINVAL] The new value specified for the `process-shared` attribute is not one of the legal values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

11.6.1.5 Cross-References

pthread_rwlock_init(), 11.6.2.

11.6.2 Initialize/Destroy a Reader/Writer Lock

Functions: *pthread_rwlock_init*(), *pthread_rwlock_destroy*().

11.6.2.1 Synopsis

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

int pthread_rwlock_init(pthread_rwlock_t *lock,
                        const pthread_rwlockattr_t *attr);

int pthread_rwlock_destroy(pthread_rwlock_t *lock);
```

11.6.2.2 Description

If {_POSIX_READER_WRITER_LOCKS} is defined:

The *pthread_rwlock_init*() function shall allocate any resources required to use the reader/writer lock referenced by *lock* and initializes the lock to an unlocked state with attributes referenced by *attr*. If *attr* is NULL, the default reader/writer lock attributes are used; the effect is the same as passing the address of a default reader/writer lock attributes object. The results are undefined if *pthread_rwlock_init*() is called specifying an already initialized reader/writer lock. The results are undefined if a reader/writer lock is used without first being initialized.

If the *pthread_rwlock_init*() function fails, the lock is not initialized and the contents of *lock* are undefined.

Only the object referenced by *lock* may be used for performing synchronization. The result of referring to copies of that object in calls to *pthread_rwlock_destroy*(), *pthread_rwlock_rdlock*(), *pthread_rwlock_timedrdlock*(), *pthread_rwlock_tryrdlock*(), *pthread_rwlock_wrlock*(), *pthread_rwlock_timedwrlock*(), *pthread_rwlock_trywrlock*(), or *pthread_rwlock_unlock*() is undefined.

The *pthread_rwlock_destroy*() function destroys the reader/writer lock referenced by *lock* and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is re-initialized by another call to *pthread_rwlock_init*(). An implementation may use this function to set the lock to an invalid value. The results are undefined if *pthread_rwlock_destroy*() is called when any thread holds the lock, or if this function is called with an uninitialized reader/writer lock.

11.6.2.3 Returns

Upon successful completion, the *pthread_rwlock_init*() and *pthread_rwlock_destroy*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.6.2.4 Errors

If any of the following conditions occur, the *pthread_rwlock_init*() function shall return the corresponding value:

[EAGAIN] The system lacks the necessary resources to initialize another reader/writer lock.

[ENOMEM] Insufficient memory exists to initialize the lock.

For each of the following conditions, if the condition is detected, the *pthread_rwlock_init*() function shall return the corresponding value:

[EBUSY] The implementation has detected an attempt to reinitialize a reader/writer lock while it is in use (for example, while being used in a *pthread_rwlock_rdlock*() call) by another thread.

[EINVAL] The value specified by *attr* is invalid.

For each of the following conditions, if the condition is detected, the *pthread_rwlock_destroy*() function shall return the corresponding value:

[EBUSY] The implementation has detected an attempt to destroy a reader/writer lock while it is in use (for example, while being used in a *pthread_rwlock_rdlock*() call) by another thread.

[EINVAL] The value specified by *lock* is invalid.

11.6.2.5 Cross-References

pthread_rwlock_rdlock(), 11.6.3; *pthread_rwlock_timedrdlock*(), 11.6.3; *pthread_rwlock_tryrdlock*(), 11.6.3; *pthread_rwlock_wrlock*(), 11.6.4; *pthread_rwlock_timedwrlock*(), 11.6.4; *pthread_rwlock_trywrlock*(), 11.6.4; *pthread_rwlock_unlock*(), 11.6.5.

11.6.3 Apply a Read Lock

Functions: *pthread_rwlock_rdlock*(), *pthread_rwlock_timedrdlock*(), *pthread_rwlock_tryrdlock*().

11.6.3.1 Synopsis

```
#include <sys/types.h>
#include <time.h>
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *lock,
                               const struct timespec *abs_timeout);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
```

11.6.3.2 Description

If {_POSIX_READER_WRITER_LOCKS} is defined:

The *pthread_rwlock_rdlock*() function applies a read lock to the reader/writer lock referenced by *lock*. The calling thread shall acquire the read lock if a writer does not hold the lock, and there are no writers blocked on the lock. If {_POSIX_THREAD_PRIORITY_SCHEDULING} is defined, and the threads involved in the lock are executing with the scheduling policies SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise the calling thread shall acquire the lock. If {_POSIX_THREAD_PRIORITY_SCHEDULING} is not defined, it is implementation-defined whether the calling thread acquires the lock when a writer does not hold the lock and there are writers blocked on the lock. If

a writer holds the lock the calling thread shall not acquire the read lock. If the lock is not acquired, the calling thread blocks (that is, does not return from the *pthread_rwlock_rdlock*() call) until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds a write lock on *lock*.

The maximum number of simultaneous read locks that an implementation guarantees can be applied to a reader/writer lock shall be implementation-defined. The *pthread_rwlock_rdlock*() function may fail if this maximum would be exceeded.

The *pthread_rwlock_tryrdlock*() function applies a read lock as in the *pthread_rwlock_rdlock*() function, with the exception that the function fails if the equivalent *pthread_rwlock_rdlock*() call would have blocked the calling thread. In no case does the *pthread_rwlock_tryrdlock*() function ever block; it always either acquires the lock or fails and returns immediately.

The results are undefined if any of these functions is called with an uninitialized reader/writer lock.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a reader/writer lock via a call to *pthread_rwlock_rdlock*(), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

If {_POSIX_READER_WRITER_LOCKS} and {_POSIX_TIMEOUTS} are both defined:

The *pthread_rwlock_timedrdlock*() function applies a read lock to the reader/writer lock referenced by *lock* as in the *pthread_rwlock_rdlock*() function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait shall be terminated when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call. If {_POSIX_TIMERS} is defined, the timeout is based on the CLOCK_REALTIME clock; if {_POSIX_TIMERS} is not defined, the timeout is based on the system clock as returned by the *time*() function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* datatype is defined as a structure in the header <time.h>. Under no circumstances shall the function fail with a timeout if the lock can be acquired immediately. The validity of the *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a reader/writer lock via a call to *pthread_rwlock_timedrdlock*(), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on *lock*. The results are undefined if this function is called with an uninitialized reader/writer lock.

11.6.3.3 Returns

Upon successful completion, the *pthread_rwlock_rdlock*(), *pthread_rwlock_timedrdlock*(), and *pthread_rwlock_tryrdlock*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.6.3.4 Errors

If any of the following conditions occur, the *pthread_rwlock_tryrdlock*() function shall return the corresponding value:

[EBUSY] A writer holds the lock, or a writer with appropriate priority is blocked on the lock.

If any of the following conditions occur, the *pthread_rwlock_timedrdlock*() function shall return the corresponding value:

[ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

For each of the following conditions, if the condition is detected, the *pthread_rwlock_rdlock*(), *pthread_rwlock_timedrdlock*(), and *pthread_rwlock_tryrdlock*() functions shall return the corresponding value:

[EINVAL] The value specified by *lock* does not refer to an initialized reader/writer lock object, or the *abs_timeout* nanosecond value is less than zero or greater than or equal to 1000 million.

For each of the following conditions, if the condition is detected, the *pthread_rwlock_rdlock*() and *pthread_rwlock_timedrdlock*() functions shall return the corresponding value:

[EDEADLK] The calling thread already holds a write lock on *lock*.

For each of the following conditions, if the condition is detected, the *pthread_rwlock_rdlock*(), *pthread_rwlock_tryrdlock*(), and *pthread_rwlock_timedrdlock*() functions shall return the corresponding value:

[EAGAIN] The read lock could not be acquired because the maximum number of read locks for *lock* would be exceeded.

11.6.3.5 Cross-References

pthread_rwlock_init(), 11.6.2; *pthread_rwlock_destroy*(), 11.6.2; *pthread_rwlock_wrlock*(), 11.6.4; *pthread_rwlock_timedwrlock*(), 11.6.4; *pthread_rwlock_trywrlock*(), 11.6.4; *pthread_rwlock_unlock*(), 11.6.5.

11.6.4 Apply a Write Lock

Functions: *pthread_rwlock_wrlock*(), *pthread_rwlock_timedwrlock*(), *pthread_rwlock_trywrlock*().

11.6.4.1 Synopsis

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

int pthread_rwlock_wrlock(pthread_rwlock_t *lock);

int pthread_rwlock_timedwrlock(pthread_rwlock_t *lock,
                        const struct timespec *abs_timeout);

int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
```

11.6.4.2 Description

If {_POSIX_READER_WRITER_LOCKS} is defined:

The *pthread_rwlock_wrlock*() function applies a write lock to the reader/writer lock referenced by *lock*. The calling thread acquires the write lock if no thread (reader or writer) holds the reader/writer lock. Otherwise, the thread blocks (that is, does not return from the *pthread_rwlock_wrlock*() call) until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds the reader/writer lock.

The *pthread_rwlock_trywrlock*() function applies a write lock as in the *pthread_rwlock_wrlock*() function, with the exception that the function fails if the equivalent *pthread_rwlock_wrlock*() call would have blocked the calling thread. In no case does the *pthread_rwlock_trywrlock*() function ever block; it always either acquires the lock or fails and returns immediately.

The results are undefined if any of these functions is called with an uninitialized reader/writer lock.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a reader/writer lock via a call to *pthread_rwlock_wrlock*(), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

If {_POSIX_READER_WRITER_LOCKS} and {_POSIX_TIMEOUTS} are both defined:

The *pthread_rwlock_timedwrlock*() function applies a write lock to the reader/writer lock referenced by *lock* as in the *pthread_rwlock_wrlock*() function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait shall be terminated when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if

the absolute time specified by *abs_timeout* has already been passed at the time of the call. If {_POSIX_TIMERS} is defined, the timeout is based on the CLOCK_REALTIME clock; if {_POSIX_TIMERS} is not defined, the timeout is based on the system clock as returned by the *time*() function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* datatype is defined as a structure in the header <time.h>. Under no circumstances shall the function fail with a timeout if the lock can be acquired immediately. The validity of the *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a reader/writer lock via a call to *pthread_rwlock_timedwrlock*(), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

The calling thread may deadlock if at the time the call is made it holds the reader/writer lock. The results are undefined if this function is called with an uninitialized reader/writer lock.

11.6.4.3 Returns

Upon successful completion, the *pthread_rwlock_wrlock*(), *pthread_rwlock_timedwrlock*(), and *pthread_rwlock_trywrlock*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.6.4.4 Errors

If any of the following conditions occur, the *pthread_rwlock_trywrlock*() function shall return the corresponding value:

[EBUSY] A reader or writer holds the lock.

If any of the following conditions occur, the *pthread_rwlock_timedwrlock*() function shall return the corresponding value:

[ETIMEDOUT]
The lock could not be acquired before the specified timeout expired.

For each of the following conditions, if the condition is detected, the *pthread_rwlock_wrlock*(), *pthread_rwlock_timedwrlock*(), and *pthread_rwlock_trywrlock*() functions shall return the corresponding value:

[EINVAL] The value specified by *lock* does not refer to an initialized reader/writer lock object, or the *abs_timeout* nanosecond value is less than zero or greater than or equal to 1000 million.

For each of the following conditions, if the condition is detected, the *pthread_rwlock_wrlock*() and *pthread_rwlock_timedwrlock*() functions shall return the corresponding value:

[EDEADLK] The calling thread already holds the reader/writer lock.

11.6.4.5 Cross-References

pthread_rwlock_init(), 11.6.2; *pthread_rwlock_destroy*(), 11.6.2; *pthread_rwlock_rdlock*(), 11.6.3; *pthread_rwlock_timedrdlock*(), 11.6.3; *pthread_rwlock_tryrdlock*(), 11.6.3; *pthread_rwlock_unlock*(), 11.6.5.

11.6.5 Unlock a Reader/Writer Lock

Function: *pthread_rwlock_unlock*(). 9

11.6.5.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_rwlock_unlock(pthread_rwlock_t *lock); 9
```

11.6.5.2 Description

A

If {_POSIX_READER_WRITER_LOCKS} is defined: 9

The *pthread_rwlock_unlock*() function releases the lock on the reader/writer lock referenced by *lock* that was locked by the calling thread via one of the *pthread_rwlock_rdlock*(), *pthread_rwlock_timedrdlock*(), *pthread_rwlock_tryrdlock*(), *pthread_rwlock_wrlock*(), *pthread_rwlock_timedwrlock*(), or *pthread_rwlock_trywrlock*() functions. The results are undefined if a lock on *lock* is not held by the calling thread. If a read lock is released by this call, and at the time of the call the released lock is the last read lock to be held on *lock*, the reader/writer lock shall become available. If a write lock is released by this call, the reader/writer lock shall become available. 9

If there are threads blocked on the lock when it becomes available, the scheduling policy is used to determine which thread(s) shall acquire the lock. If {_POSIX_THREAD_PRIORITY_SCHEDULING} is defined, when threads executing with the scheduling policies SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on the lock, they will acquire the lock in priority order when the lock becomes available. For equal priority threads, write locks take precedence over read locks. If {_POSIX_THREAD_PRIORITY_SCHEDULING} is not defined, it is implementation defined whether write locks take precedence over read locks. 9

The results are undefined if any of these functions are called with an uninitialized reader/writer lock.

9

11.6.5.3 Returns

Upon successful completion, the *pthread_rwlock_unlock*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.6.5.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_rwlock_unlock*() function shall return the corresponding value:

[EINVAL] The value specified by *lock* does not refer to an initialized reader/writer lock object.

[EPERM] The calling thread does not hold a lock on the reader/writer lock.

11.6.5.5 Cross-References

pthread_rwlock_init(), 11.6.2; *pthread_rwlock_destroy*(), 11.6.2; *pthread_rwlock_rdlock*(), 11.6.3; *pthread_rwlock_timedrdlock*(), 11.6.3; *pthread_rwlock_tryrdlock*(), 11.6.3; *pthread_rwlock_wrlock*(), 11.6.4; *pthread_rwlock_timedwrlock*(), 11.6.4; *pthread_rwlock_trywrlock*(), 11.6.4.

11.7 Spin Locks

11.7.1 Initialize/Destroy a Spin Lock

Functions: *pthread_spin_init*(), *pthread_spin_destroy*().

11.7.1.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

11.7.1.2 Description

If {_POSIX_SPIN_LOCKS} is defined:

The *pthread_spin_init*() function allocates any resources required to use the spin lock referenced by *lock* and initializes the lock to an unlocked state.

If {_POSIX_THREAD_PROCESS_SHARED} is defined:

641 If the value of *pshared* is PTHREAD_PROCESS_SHARED the imple-
642 mentation shall permit the spin lock to be operated upon by any
643 thread that has access to the memory where the spin lock is allo-
644 cated, even if it is allocated in memory that is shared by multiple
645 processes. If the value of *pshared* is PTHREAD_PROCESS_PRIVATE,
646 the spin lock shall only be operated upon by threads created within
647 the same process as the thread that initialized the spin lock; if
648 threads of differing processes attempt to operate on such a spin lock,
649 the behavior is undefined.

650 Otherwise:

651 The lock may only be operated upon by threads contained in the pro-
652 cess containing the thread that initialized the lock, independently of
653 the value of *pshared*. If threads of different processes attempt to
654 operate on such a lock, the behavior is undefined.

655 The results are undefined if *pthread_spin_init*() is called specifying an
656 already initialized spin lock. The results are undefined if a spin lock is used
657 without first being initialized.

658 If the *pthread_spin_init*() function fails, the lock is not initialized and the
659 contents of *lock* are undefined.

660 Only the object referenced by *lock* may be used for performing synchroniza- 9
661 tion. The result of referring to copies of that object in calls to 9
662 *pthread_spin_destroy*(), *pthread_spin_lock*(), *pthread_spin_trylock*(), or 9
663 *pthread_spin_unlock*() is undefined. 9

664 The *pthread_spin_destroy*() function destroys the spin lock referenced by
665 *lock* and releases any resources used by the lock. The effect of subsequent
666 use of the lock is undefined until the lock is re-initialized by another call to
667 *pthread_spin_init*(). The results are undefined if *pthread_spin_destroy*() is
668 called when a thread holds the lock, or if this function is called with an
669 uninitialized thread spin lock.
670 8

671 9

672 **11.7.1.3 Returns**

673 Upon successful completion, the *pthread_spin_init*() and *pthread_spin_destroy*()
674 functions shall return zero. Otherwise, an error number shall be returned to indi-
675 cate the error.

676 **11.7.1.4 Errors**

677 If any of the following conditions occur, the *pthread_spin_init*() function shall
678 return the corresponding value:

679 [EAGAIN] The system lacks the necessary resources to initialize another
680 spin lock.

[ENOMEM] Insufficient memory exists to initialize the lock.

For each of the following conditions, if the condition is detected, the *pthread_spin_init*() and *pthread_spin_destroy*() functions shall return the corresponding value:

[EBUSY] The implementation has detected an attempt to initialize or destroy a spin lock while it is in use (for example, while being used in a *pthread_spin_lock*() call) by another thread.

[EINVAL] The value specified by *lock* is invalid.

11.7.1.5 Cross-References

pthread_spin_lock(), 11.7.2; *pthread_spin_trylock*(), 11.7.2; *pthread_spin_unlock*(), 11.7.3.

11.7.2 Lock a Spin Lock

Functions: *pthread_spin_lock*(), *pthread_spin_trylock*().

11.7.2.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

11.7.2.2 Description

If {_POSIX_SPIN_LOCKS} is defined: 9

The *pthread_spin_lock*() function locks the spin lock referenced by *lock*. The calling thread acquires the lock if it is not held by another thread. Otherwise, the thread spins (that is, does not return from the *pthread_spin_lock*() call) until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made.

The *pthread_spin_trylock*() function locks the spin lock referenced by *lock* if it is not held by any thread. Otherwise, the function fails.

The results are undefined if any of these functions is called with an uninitialized spin lock.

9

11.7.2.3 Returns

Upon successful completion, the *pthread_spin_lock*() and *pthread_spin_trylock*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.7.2.4 Errors

If any of the following conditions occur, the *pthread_spin_trylock*() function shall return the corresponding value:

[EBUSY] A thread currently holds the lock.

For each of the following conditions, if the condition is detected, the *pthread_spin_lock*() function shall return the corresponding value:

[EDEADLK] The calling thread already holds the lock.

For each of the following conditions, if the condition is detected, the *pthread_spin_lock*() and *pthread_spin_trylock*() functions shall return the corresponding value:

[EINVAL] The value specified by *lock* does not refer to an initialized spin lock object.

11.7.2.5 Cross-References

pthread_spin_init(), 11.7.1; *pthread_spin_destroy*(), 11.7.1; *pthread_spin_unlock*(), 11.7.3.

11.7.3 Unlock a Spin Lock

Function: *pthread_spin_unlock*().

11.7.3.1 Synopsis

```
#include <sys/types.h>
#include <pthread.h>
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

11.7.3.2 Description

If {_POSIX_SPIN_LOCKS} is defined: 9

The *pthread_spin_unlock*() function releases the spin lock referenced by *lock* which was locked via the *pthread_spin_lock*() or *pthread_spin_trylock*() functions. The results are undefined if the lock is not held by the calling thread. If there are threads spinning on the lock when *pthread_spin_unlock*() is called, the lock becomes available and an unspecified spinning thread shall acquire the lock.

The results are undefined if this function is called with an uninitialized thread spin lock.

9

11.7.3.3 Returns

Upon successful completion, the *pthread_spin_unlock*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.7.3.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_spin_unlock*() function shall return the corresponding value:

[EINVAL] An invalid argument was specified.

[EPERM] The calling thread does not hold the lock.

11.7.3.5 Cross-References

pthread_spin_init(), 11.7.1; *pthread_spin_destroy*(), 11.7.1; *pthread_spin_lock*(), 11.7.2; *pthread_spin_trylock*(), 11.7.2.

Section 12: Memory Management

⇒ **12 Memory Management** *Replace the first paragraph with:*

This section describes the process memory locking, memory mapped files, shared memory facilities, and typed memory facilities available under this part of ISO/IEC 9945-1.

⇒ **12 Memory Management** *Add the following new paragraphs after the paragraph that begins with "*An *unlink*() of a file...*" and ends with "*...of the memory object mapped.*":*

Implementations may support the Typed Memory Objects option without supporting the Memory Mapped Files option or the Shared Memory Objects option. Typed memory objects are implementation-configurable named storage pools accessible from one or more processors in a system, each via one or more ports such as backplane busses, LANs, I/O channels, etc. Each valid combination of a storage pool and a port is identified through a name that is defined at system configuration time, in an implementation-defined manner; the name may be independent of the file system. Using this name, a typed memory object can be opened and mapped into process address space. For a given storage pool and port, it is necessary to support both dynamic allocation from the pool as well as mapping at an application-supplied offset within the pool; when dynamic allocation has been performed, subsequent deallocation must be supported. Lastly, accessing typed memory objects from different ports requires a method for obtaining the offset and length of contiguous storage of a region of typed memory (dynamically allocated or not); this allows typed memory to be shared among processes and/or processors while being accessed from the desired port.

12.2 Memory Mapping Functions

12.2.1 Map Process Addresses to a Memory Object

⇒ **12.2.1.2 Map Process Addresses to a Memory Object—Description** *Replace the first paragraph with:*

If at least one of {_POSIX_MAPPED_FILES}, {_POSIX_SHARED_MEMORY_OBJECTS}, or {_POSIX_TYPED_MEMORY_OBJECTS} is defined:

⇒ **12.2.1.2 Map Process Addresses to a Memory Object—Description** *In the paragraph beginning with* "The *mmap*() function establishes..." *and ending* "...object represented by *fildes*.", *replace the last sentence (beginning* "The range of bytes starting...") *with:*

The range of bytes starting at *off* and continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the file, shared memory object, or typed memory object represented by *fildes*. If *fildes* represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the memory object to be mapped shall be that portion of the typed memory object allocated by the implementation as specified below. In this case, if *off* is non-zero, the behavior of *mmap*() is undefined. If *fildes* refers to a valid typed memory object that is not accessible from the calling process, *mmap*() shall fail.

⇒ **12.2.1.2 Map Process Addresses to a Memory Object—Description** *Add the following new paragraph after the paragraph that begins with* "MAP_SHARED and MAP_PRIVATE describe..." *and ends with* "...is retained across *fork*().":

When *fildes* represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, *mmap*() shall, if there are enough resources available, map *len* bytes allocated from the corresponding typed memory object which were not previously allocated to any process in any processor that may access that typed memory object. If there are not enough resources available, the function shall fail. If *fildes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allocated bytes shall be contiguous within the typed memory object. If *fildes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of non-contiguous fragments within the typed memory object. If *fildes* represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the POSIX_TYPED_MEM_ALLOCATE flag, *len* bytes starting at offset *off* within the typed memory object are mapped, exactly as when mapping a file or shared memory object. In this case, if two processes map an area of typed memory using the same *off* and *len* values and using file descriptors that refer to the same memory pool (either from the same port or from a different port), both

processes shall map the same region of storage.

⇒ **12.2.1.4 Map Process Addresses to a Memory Object—Errors** *Add to the description of* [ENOMEM] *the following additional paragraph:*

Not enough unallocated memory resources remain in the typed memory object designated by *fildes* to allocate *len* bytes.

⇒ **12.2.1.4 Map Process Addresses to a Memory Object—Errors** *Add to the description of* [ENXIO] *the following additional paragraph:*

The *fildes* argument refers to a typed memory object that is not accessible from the calling process.

⇒ **12.2.1.5 Map Process Addresses to a Memory Object—Cross-References** *Add the following cross-reference:*

posix_typed_mem_open(), 12.4.2.

12.2.2 Unmap Previously Mapped Addresses

⇒ **12.2.2.2 Unmap Previously Mapped Addresses—Description** *Replace the first paragraph with:*

If at least one of {_POSIX_MAPPED_FILES}, {_POSIX_SHARED_MEMORY_-OBJECTS}, or {_POSIX_TYPED_MEMORY_OBJECTS} is defined:

⇒ **12.2.2.2 Unmap Previously Mapped Addresses—Description** *Add the following new paragraphs after the paragraph which begins with* "Any memory locks..." *and ending with* "...an appropriate call to *munlock*()."*:*

If a mapping removed from a typed memory object causes the corresponding address range of the memory pool to be inaccessible by any process in the system except through allocatable mappings (i.e., mappings of typed memory objects opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall become deallocated and may become available to satisfy future typed memory allocation requests.

A mapping removed from a typed memory object opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of that typed memory for allocation.

⇒ **12.2.2.5 Unmap Previously Mapped Addresses—Cross-References** *Add the following cross-reference:*

posix_typed_mem_open(), 12.4.2.

12.2.4 Memory Object synchronization

⇒ **12.2.4.2 Memory Object synchronization—Description** *Change the sentence* "The effect of *msync*() on shared memory objects is unspecified." *to:*

The effect of *msync*() on a shared memory object or a typed memory object is unspecified.

⇒ **12 Memory Management** *Add the following clause:*

12.4 Typed Memory Functions

12.4.1 Data Definitions

If {_POSIX_TYPED_MEMORY_OBJECTS} is defined, the header <sys/mman.h> shall define the memory information structure *posix_typed_mem_info*, which shall include at least the following member:

Member Type	Member Name	Description
size_t	posix_tmi_length	Maximum length which may be allocated from a typed memory object.

12.4.2 Open a Typed Memory Object

Function: *posix_typed_mem_open*()

12.4.2.1 Synopsis

```
#include <sys/mman.h>
int posix_typed_mem_open(const char *name, int oflag, int tflag);
```

12.4.2.2 Description

If {_POSIX_TYPED_MEMORY_OBJECTS} is defined:

The *posix_typed_mem_open*() function establishes a connection between the typed memory object specified by the string pointed to by *name* and a file descriptor. It creates an open file description that refers to the typed memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that typed memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument shall conform to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *posix_typed_mem_open*() with the same value of *name* shall refer to the same typed memory object. 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.

Each typed memory object supported in a system is identified by a *name* which specifies not only its associated typed memory pool, but also the path or port by which it is accessed. That is, the same typed memory pool accessed via several different ports has several different corresponding names. The binding between *name*s and typed memory objects is established in an implementation-defined manner. Unlike shared memory objects, there is ordinarily no way for a program to create a typed memory object.

The value of *tflag* determines how the typed memory object behaves when subsequently mapped by calls to *mmap*(). At most one of the following flags defined in <sys/mman.h> may be specified:

Symbolic Constant	Description
POSIX_TYPED_MEM_ALLOCATE	Allocate on *mmap*().
POSIX_TYPED_MEM_ALLOCATE_CONTIG	Allocate contiguously on *mmap*().
POSIX_TYPED_MEM_MAP_ALLOCATABLE	Map on *mmap*(), without affecting allocatability.

If *tflag* has the flag POSIX_TYPED_MEM_ALLOCATE specified, any subsequent call to *mmap*() using the returned file descriptor shall result in allocation and mapping of typed memory from the specified typed memory pool. The allocated memory may be a contiguous previously unallocated area of the typed memory pool or several non-contiguous previously unallocated areas (mapped to a contiguous portion of the process address space). If *tflag* has the flag POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to *mmap*() using the returned file descriptor shall result in allocation and mapping of a single contiguous previously unallocated area of the typed memory pool (also mapped to a contiguous portion of the process address space). If *tflag* has none of the flags POSIX_TYPED_MEM_ALLOCATE or POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to

mmap() using the returned file descriptor shall map an application-chosen area from the specified typed memory pool such that this mapped area becomes unavailable for allocation until unmapped by all processes. If *tflag* has the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE specified, any subsequent call to *mmap*() using the returned file descriptor shall map an application-chosen area from the specified typed memory pool without an effect on the availability of that area for allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The appropriate privilege to specify the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag is implementation defined.

If successful, *posix_typed_mem_open*() returns a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other processes. It is unspecified whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The behavior of *msync*(), *ftruncate*(), and all file operations other than *mmap*(), *posix_mem_offset*(), *posix_typed_mem_get_info*(), *fstat*(), *dup*(), *dup2*(), and *close*(), is unspecified when passed a file descriptor connected to a typed memory object by this function.

The file status flags of the open file description shall be set according to the value of *oflag*. Applications shall specify exactly one of the three access mode values described below and defined in the header `<fcntl.h>`, as the value of *oflag*.

O_RDONLY Open for read access only.

O_WRONLY Open for write access only.

O_RDWR Open for read or write access.

Otherwise:

Either the implementation shall support the *posix_typed_mem_open*() function as described above or this function shall not be provided.

12.4.2.3 Returns

Upon successful completion, the *posix_typed_mem_open*() function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return -1 and set *errno* to indicate the error.

12.4.2.4 Errors

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

[EACCES] The typed memory object exists and the permissions specified by *oflag* are denied.

[EINTR] The *posix_typed_mem_open*() operation was interrupted by a signal.

[EINVAL] The flags specified in *tflag* are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE, POSIX_TYPED_MEM_ALLOCATE_CONTIG, or POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).

[EMFILE] Too many file descriptors are currently in use by this process.

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

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

[ENOENT] The named typed memory object does not exist.

[EPERM] The caller lacks the appropriate privilege to specify the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE in argument *tflag*.

12.4.2.5 Cross-References

close(), 6.3.1; *dup*(), 6.2.1; *exec*, 3.1.2; *fcntl*(), 6.5.2; <fcntl.h>, 6.5.1; *umask*(), 5.3.3; *mmap*(), 12.2.1; <sys/mman.h>, 12.1.1.2; *posix_mem_offset*(), 12.4.3.

12.4.3 Find Offset and Length of a Mapped Typed Memory Block

Function: *posix_mem_offset*()

12.4.3.1 Synopsis

```
#include <sys/mman.h>
int posix_mem_offset(const void *addr, size_t len, off_t *off,
                     size_t *contig_len, int *fildes);
```

12.4.3.2 Description

If {_POSIX_TYPED_MEMORY_OBJECTS} is defined:

The *posix_mem_offset*() function returns in the variable pointed to by *off* a value that identifies the offset (or location), within a memory object, of the memory block currently mapped at *addr*. The function shall return in the variable pointed to by *fildes*, the descriptor used (via *mmap*()) to establish the mapping which contains *addr*. If that descriptor was closed since the mapping was established, the returned value of *fildes* shall be -1. The *len* argument specifies the length of the block of the memory object the user wishes the offset for; upon return, the value pointed to by *contig_len* shall

equal either *len*, or the length of the largest contiguous block of the memory object that is currently mapped to the calling process starting at *addr*, whichever is smaller.

If the memory object mapped at *addr* is a typed memory object, then if the *off* and *contig_len* values obtained by calling *posix_mem_offset*() are used in a call to *mmap*() with a file descriptor that refers to the same memory pool as *fildes* (either through the same port or through a different port), and that was opened with neither the POSIX_TYPED_MEM_ALLOCATE nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the typed memory area that is mapped shall be exactly the same area that was mapped at *addr* in the address space of the process that called *posix_mem_offset*().

If the memory object specified by *fildes* is not a typed memory object, then the behavior of this function is implementation defined.

Otherwise:

Either the implementation shall support the *posix_mem_offset*() function as described above or this function shall not be provided.

12.4.3.3 Returns

Upon successful completion, the *posix_mem_offset*() function shall return zero. Otherwise, the corresponding error status value shall be returned.

12.4.3.4 Errors

If any of the following conditions occur, the *posix_mem_offset*() function shall return the corresponding error value:

[EACCES] The process has not mapped a memory object supported by this function at the given address *addr*.

12.4.3.5 Cross-References

mmap(), 12.2.1; <sys/mman.h>, 12.1.1.2; *posix_typed_mem_open*(), 12.4.2.

12.4.4 Query Typed Memory Information

Function: *posix_typed_mem_get_info*()

12.4.4.1 Synopsis

```
#include <sys/mman.h>
int posix_typed_mem_get_info(int fildes,
                             struct posix_typed_mem_info *info);
```

12.4.4.2 Description

If {_POSIX_TYPED_MEMORY_OBJECTS} is defined:

The *posix_typed_mem_get_info*() function returns, in the *posix_tmi_length* field of the *posix_typed_mem_info* structure pointed to by *info*, the maximum length which may be successfully allocated by the typed memory object designated by *fildes*. This maximum length shall take into account the flag POSIX_TYPED_MEM_ALLOCATE or POSIX_TYPED_MEM_ALLOCATE_CONTIG specified when the typed memory object represented by *fildes* was opened. The maximum length is dynamic; therefore, the value returned is valid only while the current mapping of the corresponding typed memory pool remains unchanged.

If *fildes* represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE flag nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag specified, the returned value of *info.posix_tmi_length* is unspecified.

The *posix_typed_mem_get_info*() function may return additional implementation-defined information in other fields of the *posix_typed_mem_info* structure pointed to by *info*.

If the memory object specified by *fildes* is not a typed memory object, then the behavior of this function is undefined.

Otherwise:

Either the implementation shall support the *posix_typed_mem_get_info*() function as described above or this function shall not be provided.

12.4.4.3 Returns

Upon successful completion, the *posix_typed_mem_get_info*() function shall return zero. Otherwise, the corresponding error status value shall be returned.

12.4.4.4 Errors

If any of the following conditions occur, the *posix_typed_mem_get_info*() function shall return the corresponding error value:

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

[ENODEV] The *fildes* argument is not connected to a memory object supported by this function.

12.4.4.5 Cross-References

mmap(), 12.2,1; *posix_typed_mem_open*(), 12.4.2; <sys/mman.h>, 12.1.1.2.

Section 14: Clocks and Timers

14.1 Data Definitions for Clocks and Timers

14.1.4 Manifest Constants

⇒ **14.1.4 Manifest Constants** *Add the following text after the current definitions of constants:*

If the Monotonic Clock option is supported, the following constant shall be defined in <time.h>:

CLOCK_MONOTONIC
: The identifier for the systemwide monotonic clock, which is defined as a clock whose value cannot be set via *clock_settime*() and which cannot have backward clock jumps. The maximum possible clock jump shall be implementation defined.

⇒ **14.1.4 Manifest Constants** *Replace the paragraph starting* "The maximum allowable resolution for ..." *and the following paragraph starting* "The minimum allowable maximum value ..." *by the following text:*

The maximum allowable resolution for the CLOCK_REALTIME and the CLOCK_MONOTONIC clocks and all time services based on these clocks is represented by {_POSIX_CLOCKRES_MIN} and is defined as 20 ms (1/50 of a second). Implementations may support smaller values of resolution for these clocks to provide finer granularity time bases. The actual resolution supported by an implementation for a specific clock is obtained using functions defined in this chapter. If the actual resolution supported for a time service based on one of these clocks differs from the resolution supported for that clock, the implementation shall document this difference.

The minimum allowable maximum value for the CLOCK_REALTIME and the CLOCK_MONOTONIC clocks and all absolute time services based on them is the same as that defined by the C Standard {2} for the *time_t* type. If the maximum value supported by a time service based on one of these clocks differs from the maximum value supported by that clock, the implementation shall document this difference.

14.2 Clock and Timer Functions

14.2.1 Clocks

⇒ **14.2.1.2 Clocks—Description** *Add the following text after the paragraph starting* "A clock may be systemwide ...":

If {_POSIX_MONOTONIC_CLOCK} is defined: 9

All implementations shall support a clock_id of CLOCK_MONOTONIC defined in 14.1.4. This clock represents the monotonic clock for the system. For this clock, the value returned by *clock_gettime*() represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK_MONOTONIC clock cannot be set via *clock_settime*(). This function shall fail if it is invoked with a *clock_id* argument of CLOCK_MONOTONIC.

NOTE: Notice that the absolute value of the monotonic clock is meaningless (because its origin is arbitrary) and thus there is no need to set it. Furthermore, realtime applications can rely on the fact that the value of this clock is never set and, therefore, that time intervals measured with this clock will not be affected by calls to *clock_settime*().

⇒ **14.2.1.2 Clocks—Description** *In the description of clock_settime(), add the following paragraphs after the text that describes the effects of setting a clock via clock_settime().*

If {_POSIX_CLOCK_SELECTION} is defined, and the value of the CLOCK_REALTIME clock is set via *clock_settime*(), the new value of the clock shall be used to determine the time at which the system shall awaken a thread blocked on an absolute *clock_nanosleep*() call based upon the CLOCK_REALTIME clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally. 9

If {_POSIX_CLOCK_SELECTION} is defined, setting the value of the CLOCK_REALTIME clock via *clock_settime*() shall have no effect on any thread that is blocked on a relative *clock_nanosleep*() call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock. 9

⇒ **14.2.1.4 Clocks—Errors** *Add the following condition to the error conditions that shall cause clock_settime() to fail:*

[EINVAL] The value of the *clock_id* argument is CLOCK_MONOTONIC.

⇒ **14.2.1.5 Clocks—Cross-References** *Add the following cross-references:*

timer_create(), 14.2.2; *timer_settime*(), 14.2.4; *nanosleep*(), 14.2.5; *clock_nanosleep*(), 14.2.6; *sem_timedwait*(), 11.2.6; *pthread_mutex_timedlock*(), 11.3.3; *mq_timedsend*(), 15.2.4; *mq_timedreceive*(), 15.2.5.

14.2.2 Create a Per-Process Timer

⇒ **14.2.2.2 Create a Per-Process Timer—Description** *Add the following text at the end of the paragraph starting "*Each implementation shall define a set of clocks that ...*":*

If {_POSIX_CLOCK_SELECTION} is defined, all implementations shall support a *clock_id* of CLOCK_MONOTONIC.

⇒ **14.2 Clock and Timer Functions** *Add the following subclause:*

14.2.6 High Resolution Sleep with Specifiable Clock

Function: *clock_nanosleep*()

14.2.6.1 Synopsis

```
#include <time.h>
int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *rqtp, struct timespec *rmtp);
```

14.2.6.2 Description

If {_POSIX_CLOCK_SELECTION} is defined:

If the flag TIMER_ABSTIME is not set in the argument *flags*, the *clock_nanosleep*() function shall cause the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The clock used to measure the time shall be the clock specified by *clock_id*.

NOTE: Calling *clock_nanosleep*() with the value TIMER_ABSTIME not set in the argument *flags* and with a *clock_id* of CLOCK_REALTIME is equivalent to calling *nanosleep*() with the same *rqtp* and *rmtp* arguments.

If the flag TIMER_ABSTIME is set in the argument *flags*, the *clock_nanosleep*() function shall cause the current thread to be suspended from execution until either the time value of the clock specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If at the time of the call the time value specified by *rqtp* is less than or equal to the time value of the specified clock, then *clock_nanosleep*() shall return immediately and the calling process shall not be suspended.

The suspension time caused by this function may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time for the relative *clock_nanosleep*() function (i.e., with the TIMER_ABSTIME flag not set) shall not be less than the time interval specified by *rqtp*, as measured by the corresponding clock. The suspension for the absolute *clock_nanosleep*() function (i.e., with the TIMER_ABSTIME flag set) shall be in effect at least until the value of the corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being interrupted by a signal.

The use of the *clock_nanosleep*() function shall have no effect on the action or blockage of any signal.

The *clock_nanosleep*() function shall fail if the *clock_id* argument refers to the CPU-time clock of the calling thread. It is unspecified if *clock_id* values of other CPU-time clocks are allowed.

9

14.2.6.3 Returns

If the *clock_nanosleep*() function returns because the requested time has elapsed, its return value shall be zero.

If the *clock_nanosleep*() function returns because it has been interrupted by a signal it shall return the corresponding error value. For the relative *clock_nanosleep*() function, if the *rmtp* argument is non-**NULL**, the *timespec* structure referenced by it shall be updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is **NULL**, the remaining time is not returned. The absolute *clock_nanosleep*() function has no effect on the structure referenced by *rmtp*.

If *clock_nanosleep*() fails, it shall return the corresponding error value.

14.2.6.4 Errors

If any of the following conditions occur, the *clock_nanosleep*() function shall return the corresponding error value:

[EINTR] The *clock_nanosleep*() function was interrupted by a signal.

[EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million; or the TIMER_ABSTIME flag was specified in *flags* and the *rqtp* argument is outside the range for the clock specified by *clock_id*; or the *clock_id* argument does not specify a known clock, or specifies the CPU-time clock of the calling thread.

[ENOTSUP] The *clock_id* argument specifies a clock for which *clock_nanosleep*() is not supported, such as a CPU-time clock.

14.2.6.5 Cross-References

sleep(), 3.4.3; *nanosleep*(), 14.2.5; *clock_settime*(), 14.2.1.

Section 15: Message Passing

NOTE: The amendments to Section 15 have been removed from this draft due to the shift from relative to absolute timeouts. The section is kept as a placeholder for the diff marks associated to the deletion of this text, and thus will not appear in the final standard.

Section 18: Thread Cancellation

18.1 Thread Cancellation Overview

⇒ **18.1.2 Cancellation Points** *Add the following functions to the list of functions for which a cancellation point shall occur:*

clock_nanosleep(). 8

⇒ **18.1.2 Cancellation Points** *Add the following functions to the list of functions for which a cancellation point may also occur:*

pthread_rwlock_rdlock(), *pthread_rwlock_timedrdlock*(), 8
pthread_rwlock_wrlock(), *pthread_rwlock_timedwrlock*(), 8
posix_typed_mem_open(). 8

Annex A
(informative)

Bibliography

A.4 Other Sources of Information

⇒ **A.4 Other Sources of Information** *Add the following bibliographic entries, in the correct sorted order.*

{B79} George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1989, ISBN 0-8053-0177-1.

{B80} Steven Brawer. *Introduction to Parallel Programming*. Academic Press, 1989, ISBN 0-12-128470-0.

Annex B
(informative)

Rationale and Notes

B.11 Synchronization

⇒ **B.11 Synchronization** *Add the following subclauses:*

B.11.5 Barriers

B.11.5.1 Background

Barriers are typically used in parallel DO/FOR loops to ensure that all threads have reached a particular stage in a parallel computation before allowing any to proceed to the next stage. Highly efficient implementation is possible on machines which support a "Fetch and Add" operation as described in {B79}.

The use of return value PTHREAD_BARRIER_SERIAL_THREAD is shown in the following example:

```
    if ( (status=pthread_barrier_wait(&barrier)) ==
          PTHREAD_BARRIER_SERIAL_THREAD) {
       ...serial section
    }
    else if (status != 0) {
       ...error processing
    }
    status=pthread_barrier_wait(&barrier);
    ...
```

This behavior allows a serial section of code to be executed by one thread as soon as all threads reach the first barrier. The second barrier prevents the other threads from proceeding until the serial section being executed by the one thread has completed.

Although barriers can be implemented with mutexes and condition variables, reference {B79} provides ample illustration that such implementations are significantly less efficient than is possible. While the relative efficiency of barriers may well vary by implementation it is important that they be recognized in the POSIX standard to facilitate application portability while providing the necessary freedom to P1003.1c implementors.

B.11.5.2 Lack of Timeout Feature

Alternate versions of most blocking routines have been provided to support watchdog timeouts. No alternate interface of this sort has been provided for barrier waits for the following reasons:

1. Multiple threads may use different timeout values, some of which may be indefinite. It is not clear which threads should break through the barrier with a timeout error if and when these timeouts expire.
2. The barrier may become unusable once a thread breaks out of a *pthread_barrier_wait*() with a timeout error. There is, in general, no way to guarantee the consistency of a barrier's internal data structures once a thread has timed out of a *pthread_barrier_wait*(). Even the inclusion of a special barrier re-initialization function would not help much since it's not clear how this function would affect the behavior of threads that reach the barrier between the original timeout and the call to the re-initialization function.

B.11.6 Reader/Writer Locks

B.11.6.1 Background

Reader/writer locks are often used to allow parallel access to data on multiprocessors, to avoid context switches on uniprocessors when multiple threads access the same data, and to protect data structures that are frequently accessed (that is, read) but rarely updated (that is, written). The in-core representation of a file system directory is a good example of such a data structure. One would like to achieve as much concurrency as possible when searching directories, but limit concurrent access when adding or deleting files.

Although reader/writer locks can be implemented with mutexes and condition variables, such implementations are significantly less efficient than is possible. Therefore, this synchronization primitive is included in this standard for the purpose of allowing more efficient implementations in multiprocessor systems.

B.11.6.2 Queuing of Waiting Threads

The *pthread_rwlock_unlock*() function description states that one writer or one or more readers shall acquire the lock if it is no longer held by any thread as a result of the call. However, the function does not specify which thread(s) acquire the lock, unless the Thread Execution Scheduling option is supported.

The Realtime System Services Working Group considered the issue of scheduling with respect to the queuing of threads blocked on a reader/writer lock. The question turned out to be whether this standard should require priority scheduling of reader/writer locks for threads whose execution scheduling policy is priority-based (for example, SCHED_FIFO or SCHED_RR). There are tradeoffs between priority scheduling, the amount of concurrency achievable among readers, and the prevention of writer and/or reader starvation.

For example, suppose one or more readers hold a reader/writer lock and the following threads request the lock in the listed order:

pthread_rwlock_wrlock() - Low priority thread *writer_a*
pthread_rwlock_rdlock() - High priority thread *reader_a*
pthread_rwlock_rdlock() - High priority thread *reader_b*
pthread_rwlock_rdlock() - High priority thread *reader_c*

When the lock becomes available, should *writer_a* block the high priority readers? Or, suppose a reader/writer lock becomes available and the following are queued:

pthread_rwlock_rdlock() - Low priority thread *reader_a*
pthread_rwlock_rdlock() - Low priority thread *reader_b*
pthread_rwlock_rdlock() - Low priority thread *reader_c*
pthread_rwlock_wrlock() - Medium priority thread *writer_a*
pthread_rwlock_rdlock() - High priority thread *reader_d*

If priority scheduling is applied then *reader_d* would acquire the lock and *writer_a* would block the remaining readers. But should the remaining readers also acquire the lock to increase concurrency? The solution adopted takes into account that when the Thread Execution Scheduling option is supported, high priority threads may in fact starve low priority threads (the application developer is responsible in this case to design the system in such a way that this starvation is avoided). Therefore, the standard specifies that high priority readers take precedence over lower priority writers. However, to prevent writer starvation from threads of the same or lower priority, writers take precedence over readers of the same or lower priority.

Priority inheritance mechanisms are non-trivial in the context of reader/writer locks. When a high priority writer is forced to wait for multiple readers, for example, it is not clear which subset of the readers should inherit the writer's priority. Furthermore, the internal data structures that record the inheritance must be accessible to all readers, and this implies some sort of serialization that could negate any gain in parallelism achieved through the use of multiple readers in the first place. Finally, existing practice does not support the use of priority inheritance for reader/writer locks. Therefore, no specification of priority inheritance or priority ceiling is attempted. If reliable priority-scheduled synchronization is absolutely required, it can always be obtained through the use of mutexes.

B.11.6.3 Comparison to ISO/IEC 9945-1 *fcntl*() locks

The reader/writer locks and the *fcntl*() locks share a common goal: increasing concurrency among readers, thus increasing throughput and decreasing delay.

However, the reader/writer locks have two features not present in the *fcntl*() locks. First, under priority scheduling, reader/writer locks are granted in priority order. Second, also under priority scheduling, writer starvation is prevented by giving writers preference over readers of equal or lower priority.

Also, reader/writer locks can be used in systems lacking a file system, such as those conforming to the minimal realtime system profile of the IEEE 1003.13 profile standard.

B.11.6.4 History of Resolution Issues

Based upon some balloting objections, the draft specified the behavior of threads waiting on a reader/writer lock during the execution of a signal handler, as if the thread had not called the lock operation. However, this specified behavior would require implementations to establish internal signal handlers even though this situation would be rare, or never happen for many programs. This would introduce an unacceptable performance hit in comparison to the little additional functionality gained. Therefore, the behavior of reader/writer locks and signals was reverted back to its previous mutex-like specification.

B.11.7 Spin Locks

B.11.7.1 Background

Spin locks represent an extremely low-level synchronization mechanism suitable primarily for use on shared memory multiprocessors. It is typically an atomically modified boolean value that is set to one when the lock is held and to zero when the lock is freed.

When a caller requests a spin lock that is already held, it typically spins in a loop testing whether the lock has become available. Such spinning wastes processor cycles so the lock should only be held for short durations and not across sleep/block operations. Callers should unlock spin locks before calling sleep operations.

Spin locks are available on a variety of systems. Section 11.7 is an attempt to standardize that existing practice.

B.11.7.2 Lack of Timeout Feature

Alternate versions of most blocking routines have been provided to support watchdog timeouts. No alternate interface of this sort has been provided for spin locks for the following reasons:

1. It is impossible to determine appropriate timeout intervals for spin locks in a portable manner. The amount of time one can expect to spend spin-waiting is inversely proportional to the degree of parallelism provided by the system. It can vary from a few cycles when each competing thread is running on its own processor, to an indefinite amount of time when all threads are multiplexed on a single processor (which is why spin locking is not advisable on uniprocessors).
2. When used properly, the amount of time the calling thread spends waiting on a spin lock should be considerably less than the time required to set up a corresponding watchdog timer. Since the primary purpose of spin locks it to provide a low-overhead synchronization mechanism for multiprocessors, the overhead of a timeout mechanism was deemed unacceptable.

It was also suggested that an additional *count* argument be provided (on the *pthread_spin_lock*() call) in lieu of a true timeout so that a spin lock call could fail

gracefully if it was unable to apply the lock after *count* attempts. This idea was rejected because it is not existing practice. Furthermore, the same effect can be obtained with *pthread_spin_trylock*() as illustrated below:

```
int n = MAX_SPIN;

while ( --n >= 0 )
{
     if ( !pthread_spin_try_lock(...) )
             break;
}
if ( n >= 0 )
{
     /* Successfully acquired the lock */
}
else
{
     /* Unable to acquire the lock */
}
```

B.11.7.3 `process-shared` Attribute

The initialization functions associated with most POSIX synchronization objects (e.g., mutexes, barriers, and reader/writer locks) take an attributes object with a `process-shared` attribute that specifies whether or not the object is to be shared across processes. In the draft corresponding to the first balloting round two separate initialization functions are provided for spin locks, however: One for spin locks that were to be shared across processes (*spin_init*()), and one for locks that were only used by multiple threads within a single process (*pthread_spin_init*()). This was done so as to keep the overhead associated with spin waiting to an absolute minimum. However, the balloting group requested that, since the overhead associated to a bit check was small, spin locks should be consistent with the rest of the synchronization primitives, and thus the `process-shared` attribute was introduced for spin locks.

B.11.7.4 Spin Locks vs. Mutexes

It has been suggested that mutexes are an adequate synchronization mechanism and spin locks are not necessary. Locking mechanisms typically must trade off the processor resources consumed while setting up to block the thread and the processor resources consumed by the thread while it is blocked. Spin locks require very little resources to set up the blocking of a thread. Existing practice is to simply loop, repeating the atomic locking operation until the lock is available. While the resources consumed to set up blocking of the thread are low, the thread continues to consume processor resources while it is waiting.

On the other hand, mutexes may be implemented such that the processor resources consumed to block the thread are large relative to a spin lock. After detecting that the mutex lock is not available, the thread must alter its scheduling state, add itself to a set of waiting threads, and, when the lock becomes available again, undo all of this before taking over ownership of the mutex. However, while

a thread is blocked by a mutex, no processor resources are consumed.

Therefore, spin locks and mutexes may be implemented to have different characteristics. Spin locks may have lower overall overhead for very short term blocking, and mutexes may have lower overall overhead when a thread will be blocked for longer periods of time. The presence of both interfaces allows implementations with these two different characteristics, both of which may be useful to a particular application.

It has also been suggested that applications can build their own spin locks from the *pthread_mutex_trylock*() function:

```
while (pthread_mutex_trylock(&mutex));
```

The apparent simplicity of this construct is somewhat deceiving, however. While the actual wait is quite efficient, various guarantees on the integrity of mutex objects (e.g., priority inheritance rules) may add overhead to the successful path of the trylock operation that is not required of spin locks. One could, of course, add an attribute to the mutex to bypass such overhead but the very act of finding and testing this attribute represents more overhead than is found in the typical spin lock.

The need to hold spin lock overhead to an absolute minimum also makes it impossible to provide guarantees against starvation similar to those provided for mutexes or reader/writer locks. The overhead required to implement such guarantees (e.g, disabling preemption before spinning) may well exceed the overhead of the spin wait itself by many orders of magnitude. If a "safe" spin wait seems desirable, it can always be provided (albeit at some performance cost) via appropriate mutex attributes.

B.12 Memory Management

⇒ **B.12 Memory Management** *Add the following subclause:*

B.12.4 Typed Memory Functions

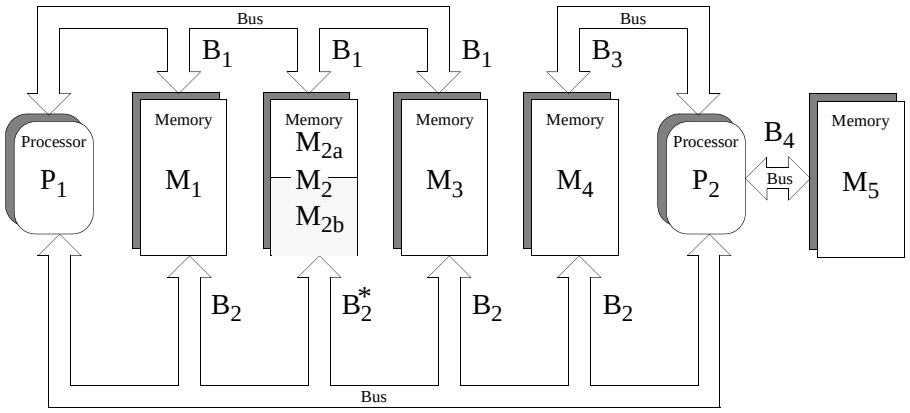
Implementations may support the Typed Memory Objects option without supporting either the Shared Memory option or the Memory Mapped Files option. Typed memory objects are pools of specialized storage, different from the main memory resource normally used by a processor to hold code and data, that can be mapped into the address space of one or more processes.

B.12.4.1 Model

Realtime systems conforming to one of the POSIX.13 realtime profiles are expected (and desired) to be supported on systems with more than one type or pool of memory (e.g., SRAM, DRAM, ROM, EPROM, EEPROM), where each type or pool of memory may be accessible by one or more processors via one or more busses

(ports). Memory Mapped Files, Shared Memory Objects, and the language-specific storage allocation operators (*malloc*() for ANSI C, **new** for ANSI Ada) fail to provide application program interfaces versatile enough to allow applications to control their utilization of such diverse memory resources. The Typed Memory interfaces *posix_typed_mem_open*(), *posix_mem_offset*(), *posix_typed_mem_get_info*(), *mmap*(), and *munmap*() defined herein support the model of typed memory described below.

For purposes of this model, a system comprises several processors (e.g., P_1 and P_2), several physical memory pools (e.g., M_1, M_2, M_{2a}, M_{2b}, M_3, M_4, and M_5), and several busses or "ports" (e.g., B_1, B_2, B_3, and B_4) interconnecting the various processors and memory pools in some system-specific way. Notice that some memory pools may be contained in others (e.g., M_{2a} and M_{2b} are contained in M_2). Figure 12-1 shows an example of such a model. In a system like this, an application should be able to perform the following operations:



*All addresses in pool M_2 (comprising pools M_{2a} and M_{2b}) accessible via port B_1.
Addresses in pool M_{2b} are also accessible via port B_2
Addresses in pool M_{2a} are NOT accessible via port B_2

Figure B-1 – Example of a system with typed memory

— *Typed memory allocation*. An application should be able to allocate memory dynamically from the desired pool using the desired bus, and map it into a process's address space. For example, processor P_1 can allocate some portion of memory pool M_1 through port B_1, treating all unmapped subareas of M_1 as a heap-storage resource from which memory may be allocated. This portion of memory is mapped into the process's address space, and subsequently deallocated when unmapped from all processes.

— *Using the same storage region from different busses*. An application process with a mapped region of storage that is accessed from one bus should be able to map that same storage area at another address (subject to page size restrictions detailed in 12.2.1.2), to allow it to be accessed from another bus.

For example, processor P_1 may wish to access the same region of memory pool M_{2b} both through ports B_1 and B_2.

— *Sharing typed memory regions*. Several application processes running on the same or different processors may wish to share a particular region of a typed memory pool. Each process or processor may wish to access this region through different busses. For example, processor P_1 may want to share a region of memory pool M_4 with processor P_2, and they may be required to use busses B_2 and B_3, respectively, to minimize bus contention. A problem arises here when a process allocates and maps a portion of fragmented memory and then wants to share this region of memory with another process, either in the same processor or different processors. The solution adopted is to allow the first process to find out the memory map (offsets and lengths) of all the different fragments of memory that were mapped into its address space, by repeatedly calling *posix_mem_offset*(). Then, this process can pass the offsets and lengths obtained to the second process, which can then map the same memory fragments into its address space.

— *Contiguous allocation*. The problem of finding the memory map of the different fragments of the memory pool that were mapped into logically contiguous addresses of a given process, can be solved by requesting contiguous allocation. For example, a process in P_1 can allocate 10 Kbytes of physically contiguous memory from M_3-B_1, and obtain the offset (within pool M_3) of this block of memory. Then, it can pass this offset (and the length) to a process in P_2 using some interprocess communication mechanism. The second process can map the same block of memory by using the offset transferred and specifying M_3-B_2.

— *Unallocated mapping*. Any subarea of a memory pool that is mapped to a process, either as the result of an allocation request or an explicit mapping, is normally unavailable for allocation. Special processes such as debuggers, however, may need to map large areas of a typed memory pool, yet leave those areas available for allocation.

Typed memory allocation and mapping has to coexist with storage allocation operators like *malloc*(), but systems are free to choose how to implement this coexistence. For example, it may be system configuration dependent if all available system memory is made part of one of the typed memory pools or if some part will be restricted to conventional allocation operators. Equally system configuration dependent may be the availability of operators like *malloc*() to allocate storage from certain typed memory pools. It is not excluded to configure a system such that a given named pool, P_1, is in turn split into non-overlapping named sub-pools. For example, M_1-B_1, M_2-B_1, and M_3-B_1 could also be accessed as one common pool M_{123}-B_1. A call to *malloc*() on P_1 could work on such a larger pool whilst full optimization of memory usage by P_1 would require typed memory allocation at the sub-pool level.

B.12.4.2 Existing Practice

OS-9 provides for the naming (numbering) and prioritization of memory types by a system administrator. It then provides APIs to request memory allocation of typed (colored) memory by number, and to generate a bus address from a mapped memory address (translate). When requesting colored memory, the user can specify type 0 to signify allocation from the first available type in priority order.

HP-RT presents interfaces to map different kinds of storage regions that are visible through a VME bus, although it does not provide allocation operations. It also provides functions to perform address translation between VME addresses and virtual addresses. It represents a VME-bus unique solution to the general problem.

The PSOS approach is similar (i.e. based on a pre-established mapping of bus address ranges to specific memories) with a concept of segments and regions (regions dynamically allocated from a heap which is a special segment). Therefore PSOS does not fully address the general allocation problem either. PSOS does not have a "process" based model, but more of a "thread" only based model of multitasking. So mapping to a process address space is not an issue.

QNX (a Canadian OS vendor specializing in realtime embedded systems on 80x86 based processors) uses the System V approach of opening specially named devices (shared memory segments) and using *mmap*() to then gain access from the process. They do not address allocation directly, but once typed shared memory can be mapped, an "allocation manager" process could be written to handle requests for allocation.

The System V approach also included allocation, implemented by opening yet other special "devices" which allocate, rather than appearing as a whole memory object.

The Orkid real-time kernel interface definition has operations to manage memory "regions" and "pools", which are areas of memory that may reflect the differing physical nature of the memory. Operations to allocate memory from these regions and pools are also provided.

B.12.4.3 Requirements

Existing practice in SVID derived UNIX[1)] systems relies on functionality similar to *mmap*() and its related interfaces to achieve mapping and allocation of typed memory. However, the issue of sharing typed memory (allocated or mapped) and the complication of multiple ports are not addressed in any consistent way by existing UNIX system practice. Part of this functionality is existing practice in specialized realtime operating systems. In order to solidify the capabilities implied by the model above, the following requirements are imposed on the interface:

---

1) UNIX is a registered trademark of The Open Group in the US and other countries.

— *Identification of typed memory pools and ports*. All processes (running in all processors) in the system shall be able to identify a particular (system configured) typed memory pool accessed through a particular (system configured) port by a name. That name shall be a member of a namespace common to all these processes, but need not be the same namespace as that containing ordinary file names. The association between memory pools/ports and corresponding names is typically established when the system is configured. The "open" operation for typed memory objects should be distinct from the *open*() function, for consistency with other similar services, but implementable on top of *open*(). This implies that the handle for a typed memory object will be a file descriptor.

— *Allocation and mapping of typed memory*. Once a typed memory object has been identified by a process, it shall be possible to both map user-selected subareas of that object into process address space and to map system-selected (i.e., dynamically allocated) subareas of that object, with user-specified length, into process address space. It shall also be possible to determine the maximum length of memory allocation that may be requested from a given typed memory object.

— *Sharing typed memory*. Two or more processes shall be able to share portions of typed memory, either user-selected or dynamically allocated. This requirement applies also to dynamically allocated regions of memory that are composed of several non-contiguous pieces.

— *Contiguous allocation*. For dynamic allocation, it shall be the user's option whether the system is required to allocate a contiguous subarea within the typed memory object, or whether it is permitted to allocate discontiguous fragments which appear contiguous in the process mapping. Contiguous allocation simplifies the process of sharing allocated typed memory, while discontiguous allocation allows for potentially better recovery of deallocated typed memory.

— *Accessing typed memory through different ports*. Once a subarea of a typed memory object has been mapped, it shall be possible to determine the location and length corresponding to a user-selected portion of that object within the memory pool. This location and length can then be used to remap that portion of memory for access from another port. If the referenced portion of typed memory was allocated discontiguously, the length thus determined may be shorter than anticipated, and the user code shall adapt to the value returned.

— *Deallocation*. When a previously mapped subarea of typed memory is no longer mapped by any process in the system—as a result of a call or calls to *munmap*() —, that subarea shall become potentially reusable for dynamic allocation; actual reuse of the subarea is a function of the dynamic typed memory allocation policy.

— *Unallocated mapping*. It shall be possible to map user-selected subareas of a typed memory object without marking that subarea as unavailable for allocation This option is not the default behavior, and shall require appropriate privilege.

B.12.4.4 Scenario

The following scenario will serve to clarify the use of the typed memory interfaces. Process *A* running on P_1 (see Figure 12-1) wants to allocate some memory from memory pool M_2, and it wants to share this portion of memory with process *B* running on P_2. Since P_2 only has access to the lower part of M_2, both processes will use the memory pool named M_{2b} which is the part of M_2 that is accessible both from P_1 and P_2. The operations that both processes need to perform are shown below:

— *Allocating typed memory*. Process *A* calls *posix_typed_mem_open*() with the name `/typed.m2b-b1` and a *tflag* of POSIX_TYPED_MEM_ALLOCATE to get a file descriptor usable for allocating from pool M_{2b} accessed through port B_1. It then calls *mmap*() with this file descriptor requesting a length of 4096 bytes. The system allocates two discontiguous blocks of sizes 1024 and 3072 bytes within M_{2b}. The *mmap*() function returns a pointer to a 4096 byte array in process *A*'s logical address space, mapping the allocated blocks contiguously. Process *A* can then utilize the array, and store data in it.

— *Determining the location of the allocated blocks.* Process *A* can determine the lengths and offsets (relative to M_{2b}) of the two blocks allocated, by using the following procedure: First, process *A* calls *posix_mem_offset*() with the address of the first element of the array and length 4096. Upon return, the offset and length (1024 bytes) of the first block are returned. A second call to *posix_mem_offset*() is then made using the address of the first element of the array plus 1024 (the length of the first block), and a new length of 4096-1024. If there were more fragments allocated, this procedure could have been continued within a loop until the offsets and lengths of all the blocks were obtained. Notice that this relatively complex procedure can be avoided if contiguous allocation is requested (by opening the typed memory object with the *tflag* POSIX_TYPED_MEM_ALLOCATE_CONTIG).

— *Sharing data across processes*. Process *A* passes the two offset values and lengths obtained from the *posix_mem_offset*() calls to process *B* running on P_2, via some form of interprocess communication. Process *B* can gain access to process *A*'s data by calling *posix_typed_mem_open*() with the name `/typed.m2b-b2` and a *tflag* of zero, then using two *mmap*() calls on the resulting file descriptor to map the two subareas of that typed memory object to its own address space.

B.12.4.5 Rationale for *posix_typed_mem_get_info*()

An application that needs to allocate a block of typed memory with length dependent upon the amount of memory currently available must either query the typed memory object to obtain the amount available, or repeatedly invoke *mmap*() attempting to guess an appropriate length. While the latter method is existing practice with *malloc*(), it is awkward and imprecise. The *posix_typed_mem_get_info*() function allows an application to immediately determine available memory. This is particularly important for typed memory objects that may in some cases be scarce resources. Note that when a typed memory pool

is a shared resource, some form of mutual exclusion or synchronization may be required while typed memory is being queried and allocated to prevent race conditions.

The existing *fstat*() function is not suitable for this purpose. We realize that implementations may wish to provide other attributes of typed memory objects (e.g., alignment requirements, page size, etc.). The *fstat*() function returns a structure which is not extensible and, furthermore, contains substantial information that is inappropriate for typed memory objects.

B.12.4.6 Rationale for no *mem_alloc*() and *mem_free*()

The working group had originally proposed a pair of new flags to *mmap*() which, when applied to a Typed Memory object descriptor, would cause *mmap*() to allocate dynamically from an unallocated and unmapped area of the Typed Memory object. Deallocation was similarly accomplished through the use of *munmap*(). This was rejected by the ballot group because it excessively complicated the (already rather complex) *mmap*() interface and introduced semantics useful only for typed memory, to a function which must also map shared memory and files. They felt that a memory allocator should be built on top of *mmap*() instead of being incorporated within the same interface, much as the ISO C libraries build *malloc*() on top of the virtual memory mapping functions *brk*() and *sbrk*(). This would eliminate the complicated semantics involved with unmapping only part of an allocated block of typed memory. 8

To attempt to achieve ballot group consensus, typed memory allocation and deallocation was first migrated from *mmap*() and *munmap*() to a pair of complementary functions modeled on ISO C *malloc*() and *free*(). The function *mem_alloc*() specified explicitly the typed memory object (typed memory pool/access port) from which allocation takes place, unlike *malloc*() where the memory pool and port are unspecified. The *mem_free*() function handled deallocation. These new semantics still met all of the requirements detailed above without modifying the behavior of *mmap*() except to allow it to map specified areas of typed memory objects. An implementation would have been free to implement *mem_alloc*() and *mem_free*() over *mmap*(), through *mmap*(), or independently but cooperating with *mmap*().

The ballot group was queried to see if this was an acceptable alternative, and while there was some agreement that it achieved the goal of removing the complicated semantics of allocation from the *mmap*() interface, several balloters realized that it just created two additional functions that behaved, in great part, like *mmap*(). These balloters proposed an alternative which we have implemented here in place of a separate *mem_alloc*() and *mem_free*(). This alternative is based on four specific suggestions:

— The function *posix_typed_mem_open*() should provide a flag which specifies "allocate on *mmap*()" (otherwise, *mmap*() just maps the underlying object). This allows things roughly similar to `/dev/zero` vs. `/dev/swap`. We have implemented two such flags, one of which forces contiguous allocation.

— The function *posix_mem_offset*() is acceptable because it can be applied usefully to mapped objects in general. It should return the file descriptor of

the underlying object.

— The function named *mem_get_info*() in an earlier draft should be renamed *posix_typed_mem_get_info*() because it is not generally applicable to memory objects. It should probably return the file descriptor's allocation attribute. We have implemented the renaming of the function, but reject having it return a piece of information which is readily known by an application without this function. Its whole purpose is to query the typed memory object for attributes that are not user specified, but determined by the implementation.

— There should be no separate *mem_alloc*() or *mem_free*() functions. Instead, using *mmap*() on a typed memory object opened with an "allocate on *mmap*()" flag should be used to force allocation. These are precisely the semantics defined in the current draft.

B.12.4.7 Rationale for no Typed Memory Access Management

The working group had originally defined an additional interface (and an additional kind of object: Typed Memory Master) to establish and dissolve mappings to typed memory on behalf of devices or processors which were independent of the operating system and had no inherent capability to directly establish mappings on their own. This was to have provided functionality similar to device driver interfaces such as *physio*() and their underlying bus-specific interfaces (e.g., *mballoc*()) which serve to set up and break down DMA pathways, and derive mapped addresses for use by hardware devices and processor cards.

The ballot group felt that this was beyond the scope of IEEE 1003.1 and its amendments. Furthermore, the removal of interrupt handling interfaces from a preceding amendment (IEEE 1003.1d) during its balloting process renders these Typed Memory Access Management interfaces an incomplete solution to portable device management from a user process; it would be possible to initiate a device transfer to/from typed memory, but impossible to handle the transfer-complete interrupt in a portable way.

To achieve ballot group consensus, all references to Typed Memory Access Management capabilities were removed. The concept of portable interfaces from a device driver to both operating system and hardware is being addressed by the Uniform Driver Interface (UDI) industry forum, with formal standardization deferred until proof of concept and industry-wide acceptance and implementation.

B.14 Clocks and Timers

⇒ **B.14 Clocks and Timers** *Add the following subclause after the unnumbered subclause* "clocks"*:*

Rationale for the Monotonic Clock

For those applications that use time services to achieve realtime behavior, changing the value of the clock on which these services rely may cause erroneous timing behavior. For these applications, it is necessary to have a monotonic clock which cannot run backwards, and which has a maximum clock jump that is required to be documented by the implementation. Additionally, it is desirable (but not required by this standard) that the monotonic clock increases its value uniformly. This clock should not be affected by changes to the system time, for example to synchronize the clock with an external source or to account for leap seconds. Such changes would cause errors in the measurement of time intervals for those time services that use the absolute value of the clock.

One could argue that by defining the behavior of time services when the value of a clock is changed, deterministic realtime behavior can be achieved. For example, one could specify that relative time services should be unaffected by changes in the value of a clock. However, there are time services that are based upon an absolute time, but that are essentially intended as relative time services. For example, *pthread_cond_timedwait*() uses an absolute time to allow it to wake up after the required interval despite spurious wakeups. Although sometimes the *pthread_cond_timedwait*() timeouts are absolute in nature, there are many occasions in which they are relative, and their absolute value is determined from the current time plus a relative time interval. In this latter case, if the clock changes while the thread is waiting, the wait interval will not be the expected length. If a *pthread_cond_timedwait*() function were created that would take a relative time, it would not solve the problem because to retain the intended "deadline" a thread would need to compensate for latency due to the spurious wakeup, and preemption between wakeup and the next wait.

The solution is to create a new monotonic clock, whose value does not change except for the regular ticking of the clock, and use this clock for implementing the various relative timeouts that appear in the different POSIX interfaces, as well as allow *pthread_cond_timedwait*() to choose this new clock for its timeout. A new *clock_nanosleep*() function is created to allow an application to take advantage of this newly defined clock. Notice that the monotonic clock may be implemented using the same hardware clock as the system clock.

Relative timeouts for *sigtimedwait*() and *aio_suspend*() have been redefined to use the monotonic clock, if present. The *alarm*() function has not been redefined, because the same effect but with better resolution can be achieved by creating a timer (for which the appropriate clock may be chosen).

The *pthread_cond_timedwait*() function has been treated in a different way, compared to other functions with absolute timeouts, because it is used to wait for an event, and thus it may have a deadline, while the other timeouts are generally used as an error recovery mechanism, and for them the use of the monotonic clock is not so important. Since the desired timeout for the *pthread_cond_timedwait*() function may either be a relative interval, or an absolute time of day deadline, a new initialization attribute has been created for condition variables, to specify the clock that shall be used for measuring the timeout in a call to *pthread_cond_timedwait*(). In this way, if a relative timeout is desired, the monotonic clock will be used; if an absolute deadline is required instead, the CLOCK_REALTIME or another appropriate clock may be used. This capability has not been added to other functions with absolute timeouts because for those functions the expected use of the timeout is mostly to prevent errors, and not so often to meet precise deadlines. As a consequence, the complexity of adding this capability is not justified by its perceived application usage.

The *nanosleep*() function has not been modified with the introduction of the monotonic clock. Instead, a new *clock_nanosleep*() function has been created, in which the desired clock may be specified in the function call.

History of Resolution Issues

Due to the shift from relative to absolute timeouts in IEEE 1003.1d, the amendments to the *sem_timedwait*(), *pthread_mutex_timedlock*(), *mq_timedreceive*(), and *mq_timedsend*() functions of that standard have been removed. Those amendments specified that CLOCK_MONOTONIC would be used for the (relative) timeouts if the Monotonic Clock option was supported.

Having these functions continue to be tied solely to CLOCK_MONOTONIC would not work. Since the absolute value of a time value obtained from CLOCK_MONOTONIC is unspecified, under the absolute timeouts interface, applications would behave differently depending on whether the Monotonic Clock option was supported or not (because the absolute value of the clock would have different meanings in either case).

Two options were considered: 1) leave the current behavior unchanged, which specifies the CLOCK_REALTIME clock for these (absolute) timeouts, to allow portability of applications between implementations supporting or not the Monotonic Clock option, or 2) modify these functions in the way that *pthread_cond_timedwait*() was modified to allow a choice of clock, so that an application could use CLOCK_REALTIME when it is trying to achieve an absolute timeout and CLOCK_MONOTONIC when it is trying to achieve a relative timeout.

It was decided that the features of CLOCK_MONOTONIC are not as critical to these functions as they are to *pthread_cond_timedwait*(). When *pthread_cond_timedwait*() is given a relative timeout, the timeout may represent a deadline for an event. When these functions are given relative timeouts, the timeouts are typically for error recovery purposes and need not be so precise.

Therefore, it was decided that these functions should be tied to CLOCK_REALTIME and not complicated by being given a choice of clock.

B.14.2 Clock and Timer Functions

⇒ **B.14.2 Clock and Timer Functions** *Add the following subclause:*

B.14.2.6 High Resolution Sleep with Specifiable Clock

Rationale for clock_nanosleep()

The *nanosleep*() function specifies that the systemwide clock CLOCK_REALTIME is used to measure the elapsed time for this time service. However, with the introduction of the monotonic clock CLOCK_MONOTONIC a new relative sleep function is needed to allow an application to take advantage of the special characteristics of this clock.

Rationale for absolute clock_nanosleep()

There are many applications in which a process needs to be suspended and then activated multiple times in a periodic way, for example to poll the status of a non-interrupting device or to refresh a display device. For these cases, it is known that precise periodic activation cannot be achieved with a relative *sleep*() or *nanosleep*() function call. Suppose for example, a periodic process that is activated at time T_0, executes for a while, and then wants to suspend itself until time T_0+T, the period being T. If this process wants to use the *nanosleep*() function, it must first call *clock_gettime*() to get the current time, then calculate the difference between the current time and T_0+T and, finally, call *nanosleep*() using the computed interval. However, the process could be preempted by a different process between the two function calls, and in this case the interval computed would be wrong; the process would wake up later than desired. This problem would not occur with the absolute *clock_nanosleep*() function, since only one function call would be necessary to suspend the process until the desired time. In other cases, however, a relative sleep is needed, and that is why both functionalities are required.

Although it is possible to implement periodic processes using the timers interface, this implementation would require the use of signals, and the reservation of some signal numbers. In this regard, the reasons for including an absolute version of the *clock_nanosleep*() function in the standard are the same as for the inclusion of the relative *nanosleep*().

It is also possible to implement precise periodic processes using *pthread_cond_timedwait*(), in which an absolute timeout is specified that takes effect if the condition variable involved is never signaled. However, the use of this interface is unnatural, and involves performing other operations on mutexes and condition variables that imply an unnecessary overhead. Furthermore, *pthread_cond_timedwait*() is not available in implementations that do not support threads.

Although the interface of the relative and absolute versions of the new high resolution sleep service is the same *clock_nanosleep*() function, the *rmtp* argument is only used in the relative sleep. This argument is needed in the relative *clock_nanosleep*() function to re-issue the function call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep*() function call; if the call is interrupted by a signal, the absolute *clock_nanosleep*() function can be invoked again with the same *rqtp* argument used in the interrupted call.

B.18 Thread Cancellation

B.18.1 Thread Cancellation Overview

B.18.1.2 Cancellation Points

⇒ **B.18.1.2 Cancellation Points** *Replace the third and fourth paragraphs, starting with* "There is one important blocking routine..." *and ending with* "... be protected with condition variables." *with the following:*

Several important blocking routines are not cancellation points.

(1) *pthread_mutex_lock*()

If *pthread_mutex_lock*() were a cancellation point, every routine that called it would also become a cancellation point (that is, any routine that touched shared state would automatically become a cancellation point). For example, *malloc*(), *free*(), and *rand*(), would become cancellation points under this scheme. Having too many cancellation points makes programming very difficult, leading to either much disabling and restoring of cancelability or much difficulty in trying to arrange for reliable cleanup at every possible place.

Since *pthread_mutex_lock*() is not a cancellation point, threads could result in being blocked uninterruptibly for long periods of time if mutexes were used as a general synchronization mechanism. As this is normally not acceptable, mutexes should only be used to protect resources that are held for small fixed lengths of time where not being cancelable will not be a problem. Resources that need to be held exclusively for long periods of time should be protected with condition variables.

(2) *barrier_wait*()

Canceling a barrier wait will render a barrier unusable. Similar to a barrier timeout (which the Working Group rejected), there is no way to guarantee the consistency of a barrier's internal data structures if a barrier wait is canceled.

(3) *pthread_spin_lock*()

As with mutexes, spin locks should only be used to protect resources that are held for small fixed lengths of time where not being cancelable will not be a problem.

Annex F
(informative)
Portability Considerations

F.3 Profiling Considerations

⇒ **F.3.1 Configuration Options** *Add the following options in order:* A

{_POSIX_BARRIERS}
The system supports barrier synchronization.

This option was created to allow efficient synchronization of multiple parallel threads in multiprocessor systems in which the operation is supported in part by the hardware architecture.

{_POSIX_CLOCK_SELECTION} 9
The system supports the Clock Selection option. 9

This option allows applications to request a high resolution sleep in order to suspend a thread during a relative time interval, or until an absolute time value, using the desired clock. It also allows the application to select the clock used in 9
a *pthread_cond_timedwait*() function call. 9

{_POSIX_MONOTONIC_CLOCK}
The system supports the Monotonic Clock option.

This option allows realtime applications to rely on a monotonically increasing clock that does not jump backwards, and whose value does not change except for the regular ticking of the clock.

{_POSIX_READER_WRITER_LOCKS}
The system supports reader/writer locks.

This option was created to support efficient synchronization in shared memory multiprocessors in which multiple

simultaneous reads are allowed to a shared resource.

{_POSIX_SPIN_LOCKS}
The system supports spin locks.

This option was created to support a simple and efficient synchronization mechanism for threads executing in multiprocessor systems.

{_POSIX_TYPED_MEMORY_OBJECTS}
The system supports typed memory objects.

This option was created to allow realtime applications to access different kinds of physical memory, and allow processes in these applications to share portions of this memory.

Identifier Index

Alphabetic Topical Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W