

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

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

P1003.1j / D10 September 1999

Copyright © 1999 by the Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change. Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities. Permission is also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position.

Other entities seeking permission to reproduce this document for standardization or other activities, or to reproduce portions of this document for these or other uses, must contact the IEEE Standards Department for the appropriate license. Use of information contained in this unapproved draft is at your own risk.

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

1 *Editor's Notes*

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

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

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

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

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

29 *Please report typographical errors to:*

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

39 *(Electronic mail is preferred.)*

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

43
44
45
46
47
48
49
50

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>

51 *POSIX.1j Change History*

52 This section is provided to track major changes between drafts.

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

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

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

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

61 — Because of the Backwards Compatibility requirement, the
62 “Otherwise” clauses in those functions whose names do not
63 start with the “posix_” reserved prefix, were deleted. See
64 “Stubs and Support for Optional Features” in this Editor’s
65 Notes, for further information.

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

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

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

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

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

79 — Annex I (Thread Management Considerations), Annex J (Syn-
80 chronized Clock), and Annex K (Balloting Instructions) were
81 removed from the draft.

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

83 — Moved the “Conventions” and “Normative References” sub-
84 clauses into these editor’s notes, because no amendment to the
85 equivalent subclauses in POSIX.1 was intended.

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

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

- 91 Draft 7 [October 1998] Reballot with new ballot group.
92 — Added the new backwards compatibility requirement and
93 changed draft accordingly.
- 94 Draft 6 [November 1997] First recirculation.
95 — Merged the process and thread spin locks, and changed all the
96 names of the barrier and reader/writer lock functions to follow
97 the pthreads model.
98 — Changed the requirements on stubs to resolve balloting objec-
99 tions requesting consistency with POSIX.1c.
100 — Deleted the Typed Memory Access Management option.
101 — Moved the typed memory allocation flags out of *mmap()*, into
102 *posix_typed_mem_open()*.
103 — Moved the Thread Abortion chapter to an informative annex.
104 — Moved the Synchronized Clock to an informative annex.
- 105 Draft 5 [May 1995] First balloting round.
106 — Minor editorial changes, and deletion of all diff marks.
107 — Changed the requirements on stubs to follow the new SICC pol-
108 icy.
- 109 Draft 4 [Apr 1995]
110 — Added the Monotonic Clock, the Synchronized Clock, and the
111 *nanosleep_rel()* function. Changed relative timeouts to depend
112 on the Monotonic Clock, if present. Added an initialization
113 attribute to condition variables, to specify the clock that shall
114 be used for the timeout service in *pthread_cond_timedwait()*.
115 — Added the Synchronized Clock.
116 — Added initialization attributes objects to barriers and
117 reader/writer locks, and made some changes to the synchroni-
118 zation functions.
119 — Some minor changes to typed memory.
- 120 Draft 3 [Nov 1994]
121 — Some changes to the Synchronization Chapter.
122 — Added the barrier wait, reader/writer lock and spin lock calls
123 to the list of blocking routines that are not cancellation points.
- 124 Draft 2 [Sep 1994]
125 — Added the Thread Abortion Chapter.
126 — Specified the effects of changing the time of
127 CLOCK_REALTIME.

- 128 — Minor technical changes to the synchronization chapter.
129 Draft 1 [Jul 1994]
130 — Added new options and definitions to Sections 1 and 2, related
131 to the Synchronization Section.
132 — Added the Synchronization Section.
133 — Deleted the placeholder for the Message Passing Section.
134 Draft 0 [Apr 1994]
135 — Preliminary draft, prior to PAR approval. Not reviewed by the
136 Working Group.

137 **Stubs and Support for Optional Features**

138 Drafts of POSIX.1j previous to Draft 9 had required that implementations not sup-
139 porting a specific option must either not provide a function named under the
140 option, or provide that function exactly as specified in the standard. This was
141 stated in the “Otherwise” clause that appeared in every optional function; among
142 other things, this requirement prevented the implementation from providing
143 stubs.

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

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

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

165 **Normative References**

166 NOTE: This standard does not amend subclause 1.2, Normative References, of ISO/IEC 9945-
167 1:1996. However, the Normative References of ISO/IEC 9945-1:1996 are repeated here for informa-
168 tion. In addition, since IEEE P1003.1j modifies ISO/IEC 9945-1:1996 as amended by IEEE
169 1003.1d:1999 (and by IEEE P1003.1a, if approved before this standard), we have included the latter

170 two among this informal list of references.

171 The following standards contain provisions which, through references in this text,
172 constitute provisions of this standard. At the time of publication, the editions
173 indicated were valid. All standards are subject to revision, and parties to agree-
174 ments based on this part of this International Standard are encouraged to investi-
175 gate the possibility of applying the most recent editions of the standards listed
176 below. Members of IEC and ISO maintain registers of currently valid Interna-
177 tional Standards.

- 178 {1} ISO/IEC 9899: 1995¹⁾, *Information processing systems—Programming*
179 *languages—C.*
- 180 {2} ISO/IEC 9945-1: 1996 (IEEE Std 1003.1-1996), *Information technology—*
181 *Portable operating system interface (POSIX)—Part 1: System application*
182 *program interface (API) [C Language].*
- 183 {3} IEEE Draft Std. P1003.1a, Draft 14, January 1998, *Information Technology*
184 *— Portable Operating System Interface (POSIX) — Part 1: System Applica-*
185 *tion Program Interface (API) [C Language] — Amendment*
- 186 {4} IEEE Std 1003.1d:1999, *Information Technology — Portable Operating Sys-*
187 *tem Interface (POSIX) — Part 1: System Application Program Interface*
188 *(API) [C Language] — Amendment x: Additional Realtime Extensions*
- 189 {5} IEEE Std 610-1990, *IEEE Standard Computer Dictionary — A Compilation*
190 *of IEEE Standard Computer Glossaries*

191 Conventions

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

194 This document uses the following typographic conventions:

- 195 (1) The *italic* font is used for:
- 196 — Cross references to defined terms within 1.3, 2.2.1, and 2.2.2; symbolic
197 parameters that are generally substituted with real values by the
198 application
 - 199 — C language data types and function names (except in function
200 Synopsis subclauses)
 - 201 — Global external variable names
 - 202 — Function families; references to groups of closely related functions
203 (such as *directory*, *exec*, *sigsetops*, *sigwait*, *stdio*, and *wait*)

204

205 1) ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembe, Case Postale 56, CH-
206 1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States
207 from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th
208 Floor, New York, NY 10036, USA.

- 209 (2) The **bold** font is used with a word in all capital letters, such as
210 **PATH**
211 to represent an environment variable, as described in 2.6. It is also used
212 for the term “**NULL** pointer.”
- 213 (3) The `constant-width` (Courier) font is used:
214 — For C language data types and function names within function
215 Synopsis subclauses
216 — To illustrate examples of system input or output where exact usage is
217 depicted
218 — For references to utility names and C language headers
219 — For names of attributes in attributes objects
- 220 (4) Symbolic constants returned by many functions as error numbers are
221 represented as:
222 [ERRNO]
223 See 2.4.
- 224 (5) Symbolic constants or limits defined in certain headers are represented
225 as
226 {LIMIT}
227 See 2.8 and 2.9.
- 228 In some cases tabular information is presented “inline”; in others it is presented in
229 a separately labeled table. This arrangement was employed purely for ease of
230 typesetting and there is no normative difference between these two cases.
- 231 The conventions listed previously are for ease of reading only. Editorial incon-
232 sistencies in the use of typography are unintentional and have no normative
233 meaning in this standard.
- 234 NOTES provided as parts of labeled tables and figures are integral parts of this
235 standard (normative). Footnotes and notes within the body of the text are for
236 information only (informative).
- 237 Numerical quantities are presented in international style: comma is used as a
238 decimal sign and units are from the International System (SI).

239 *POSIX.1j Technical Reviewers*

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

243 **Table i — POSIX.1j Technical Reviewers**

244

245	Section	Description	Reviewer
246		<i>Ballot Coordinators</i>	Joe Gwinn and Jim Oblinger
247	11.5-7	<i>Synchronization</i>	Karen Gordon and Michael González
248	5,6,8,12	<i>Typed Memory</i>	Frank Prindle
249	3,6.7,11.2-4,14,15	<i>Monotonic Clock and Nanosleep</i>	Michael González
250			

Contents

	PAGE
253	Introduction V
254	Section 1: General 1
255	1.3 Conformance 1
256	Section 2: Terminology and General Requirements 3
257	2.2 Definitions 3
258	2.2.2 General Terms 3
259	2.5 Primitive System Data Types 4
260	2.7 C Language Definitions 5
261	2.7.3 Headers and Function Prototypes 5
262	2.8 Numerical Limits 6
263	2.8.7 Maximum Values 6
264	2.9 Symbolic Constants 7
265	2.9.3 Compile-Time Symbolic Constants for Portability
266	Specifications 7
267	Section 3: Process Primitives 9
268	3.1 Process Creation and Execution 9
269	3.1.2 Execute a File 9
270	3.2 Process Termination 9
271	3.2.2 Terminate a Process 9
272	3.3 Signals 9
273	3.3.8 Synchronously Accept a Signal 9
274	Section 4: Process Environment 11
275	4.8 Configurable System Variables 11
276	4.8.1 Get Configurable System Variables 11
277	Section 5: Files and Directories 13
278	5.6 File Characteristics 13
279	5.6.1 File Characteristics: Header and Data Structure 13
280	5.6.2 Get File Status 13
281	5.6.4 Change File Modes 13
282	Section 6: Input and Output Primitives 15
283	6.3 File Descriptor Deassignment 15
284	6.3.1 Close a File 15
285	6.4 Input and Output 15
286	6.4.1 Read from a File 15
287	6.4.2 Write to a File 15
288	6.5 Control Operations on Files 16

289	6.5.2	File Control	16
290	6.5.3	Reposition Read/Write File Offset	16
291	6.7	Asynchronous Input and Output	16
292	6.7.8	Wait for an Asynchronous I/O Request	16
293	Section 8:	Language-Specific Services for the C Language	17
294	8.2	C Language Input/Output Functions	17
295	8.2.2	Open a Stream on a File Descriptor	17
296	Section 11:	Synchronization	19
297	11.4	Condition Variables	19
298	11.4.1	Condition Variable Initialization Attributes	19
299	11.4.4	Waiting on a Condition	20
300	11.5	Barriers	21
301	11.5.1	Barrier Initialization Attributes	21
302	11.5.2	Initialize/Destroy a Barrier	23
303	11.5.3	Synchronize at a Barrier	24
304	11.6	Reader/Writer Locks	26
305	11.6.1	Reader/Writer Lock Initialization Attributes	26
306	11.6.2	Initialize/Destroy a Reader/Writer Lock	28
307	11.6.3	Apply a Read Lock	30
308	11.6.4	Apply a Write Lock	33
309	11.6.5	Unlock a Reader/Writer Lock	35
310	11.7	Spin Locks	36
311	11.7.1	Initialize/Destroy a Spin Lock	36
312	11.7.2	Lock a Spin Lock	38
313	11.7.3	Unlock a Spin Lock	39
314	Section 12:	Memory Management	41
315	12.2	Memory Mapping Functions	41
316	12.2.1	Map Process Addresses to a Memory Object	42
317	12.2.2	Unmap Previously Mapped Addresses	43
318	12.2.4	Memory Object synchronization	44
319	12.4	Typed Memory Functions	44
320	12.4.1	Data Definitions	44
321	12.4.2	Open a Typed Memory Object	44
322	12.4.3	Find Offset and Length of a Mapped Typed Memory Block	47
323	12.4.4	Query Typed Memory Information	48
324	Section 14:	Clocks and Timers	51
325	14.1	Data Definitions for Clocks and Timers	51
326	14.1.4	Manifest Constants	51
327	14.2	Clock and Timer Functions	52
328	14.2.1	Clocks	52
329	14.2.2	Create a Per-Process Timer	53
330	14.2.6	High Resolution Sleep with Specifiable Clock	53
331	Section 15:	Message Passing	57

332	Section 18: Thread Cancellation	59
333	18.1 Thread Cancellation Overview	59
334	Annex A (informative) Bibliography	61
335	A.4 Other Sources of Information	61
336	Annex B (informative) Rationale and Notes	63
337	B.11 Synchronization	63
338	B.12 Memory Management	68
339	B.14 Clocks and Timers	76
340	B.18 Thread Cancellation	79
341	Annex F (informative) Portability Considerations	81
342	F.3 Profiling Considerations	81
343	Identifier Index	83
344	Alphabetic Topical Index	85
345	FIGURES	
346	Figure B-1 – Example of a system with typed memory	69
347	TABLES	
348	Table 2-2 – Optional Primitive System Data Types	5
349	Table 2-11 – Versioned Compile-Time Symbolic Constants	8
350	Table 4-3 – Optional Configurable System Variables	11

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.)

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

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

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

13 This standard defines systems interfaces to support the source portability of appli-
14 cations with realtime requirements. The system interfaces are all extensions of or
15 additions to ISO/IEC 9945-1:1996, as amended by IEEE-1003.1d:1999 (and by IEEE 9
16 1003.1a, if approved before this standard). Although rooted in the culture defined 9
17 by ISO/IEC 9945-1: 1990, they are focused upon the realtime application require-
18 ments, and the support of multiple threads of control within a process, which were
19 beyond the scope of ISO/IEC 9945-1: 1990. The interfaces included in this stan-
20 dard were the set required to make ISO/IEC 9945-1: 1990 efficiently usable to real-
21 time applications or applications running in multiprocessor systems with require-
22 ments that were not covered by the realtime or threads extensions specified in
23 IEEE-1003.1b, IEEE-1003.1c, and IEEE-1003.1d. The scope is to take existing real-
24 time or multiprocessor operating system practice and add it to the base standard.

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

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

28 The key elements of defining the scope are a) defining a sufficient set of functional-
29 ity to cover the realtime application program domain in the areas not covered by
30 IEEE-1003.1b, IEEE-1003.1c, and IEEE-1003.1d; b) defining a sufficient set of func-
31 tionality to cover efficient synchronization in multiprocessors that allows applica-
32 tions to achieve the performance benefits of such architectures; c) defining
33 sufficient performance constraints and performance-related functions to allow a
34 realtime application to achieve deterministic response from the system; and d)
35 specifying changes or additions to improve or complete the definition of the facili-
36 ties specified in the previous real-time or threads extensions IEEE-1003.1b, IEEE-
37 1003.1c, and IEEE-1003.1d.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

38 Wherever possible, the requirements of other application environments were
39 included in the interface definition. The specific areas are noted in the scope over-
40 views of each of the interface areas given below.

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

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

52 This standard has been defined exclusively at the source code level, for the C pro-
53 gramming language. Although the interfaces will be portable, some of the parame-
54 ters used by an implementation may have hardware or configuration dependen-
55 cies.

56 **Related Standards Activities**

57 Activities to extend this standard to address additional requirements are in pro-
58 gress, and similar efforts can be anticipated in the future.

59 The following areas are under active consideration at this time, or are expected to
60 become active in the near future:²⁾

- 61 (1) Additional System Application Program Interfaces in C Language 8
- 62 (2) Ada language bindings to this standard
- 63 (3) Shell and utility facilities
- 64 (4) Verification testing methods
- 65 (5) Tracing facilities 8
- 66 (6) Fault tolerance 8
- 67 (7) Checkpoint/restart facilities 8
- 68 (8) Resource limiting facilities 8
- 69 (9) Network interface facilities
- 70 (10) System administration
- 71 (11) Profiles describing application- or user-specific combinations of Open Sys- 8
72 tems standards 8
- 73 (12) An overall guide to POSIX-based or related Open Systems standards and
74 profiles

75 Extensions are approved as “amendments” or “revisions” to this document, fol-
76 lowing the IEEE and ISO/IEC Procedures.

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

79 If you have interest in participating in the PASC working groups addressing these
80 issues, please send your name, address, and phone number to the Secretary, IEEE
81 Standards Board, Institute of Electrical and Electronics Engineers, Inc., P.O. Box
82 1331, 445 Hoes Lane, Piscataway, NJ 08855-1331, and ask to have this forwarded
83 to the chairperson of the appropriate PASC working group. If you have interest in
84 participating in this work at the international level, contact your ISO/IEC national
85 body.

86 _____
87 2) A *Standards Status Report* that lists all current IEEE Computer Society standards projects is
88 available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC
89 20036-1903; Telephone: +1 202 371-0101; FAX: +1 202 728-9614. Working drafts of POSIX
90 standards under development are also available from this office.

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

95 **Portable Application Standards Committee**
96 **(PASC)**

97 Chair: Lowell Johnson
98 Vice Chair: Joe Gwinn
99 Functional Chairs: Jay Ashford
100 Andrew Josey
101 Curtis Royster
102 Secretary: Nick Stoughton

103 **System Services Working Group—Realtime: Officials**

104 Chair: Joe Gwinn
105 Susan Corwin (until 1995)
106 Editor: Michael González
107 Secretary: Karen Gordon
108 Lee Schemerhorn (until 1995)

109 **Ballot Coordinators**

110 Joe Gwinn Jim Oblinger

111 **Technical Reviewers**

112 Michael González Karen Gordon Frank Prindle

113 **Working Group**

114 to be supplied to be supplied to be supplied

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

117 Institutional Representatives <To be filled in>

118 Individual Balloters <To be filled in>

119 When the IEEE Standards Board approved this standard on *<date to be pro-*
120 *vided>*, it had the following membership:

121 (to be pasted in by IEEE)

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

8

2 **1.3 Conformance**

3 **1.3.1 Implementation Conformance**

4 ⇒ **1.3.1.3 Conforming Implementation Options** *Add to the table of imple-*
5 *mentation options that warrant requirement by applications or in*
6 *specifications:*

7	{_POSIX_BARRIERS}	Barriers option in (2.9.3)	
8	{_POSIX_CLOCK_SELECTION}	Clock Selection option (in 2.9.3)	9
9	{_POSIX_MONOTONIC_CLOCK}	Monotonic Clock option (in 2.9.3)	
10	{_POSIX_READER_WRITER_LOCKS}	Reader/Writer Locks option in (2.9.3)	
11	{_POSIX_SPIN_LOCKS}	Spin Locks option (in 2.9.3)	
12	{_POSIX_TYPED_MEMORY_OBJECTS}	Typed Memory Objects option (in 2.9.3)	

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 2: Terminology and General Requirements

1

8

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.

23 **2.2.2.137 spin lock:** A synchronization object used to allow multiple threads to
 24 serialize their access to shared data.

25 **2.2.2.138 typed memory namespace:** A system-wide namespace that contains
 26 the names of the typed memory objects present in the system. It is configurable
 27 for a given implementation.

28 **2.2.2.139 typed memory object:** A combination of a typed memory pool and a
 29 typed memory port. The entire contents of the pool shall be accessible from the
 30 port. The typed memory object is identified through a name that belongs to the
 31 typed memory namespace.

32 **2.2.2.140 typed memory pool:** An extent of memory with the same operational
 33 characteristics. Typed memory pools may be contained within each other.

34 **2.2.2.141 typed memory port:** A hardware access path to one or more typed
 35 memory pools.

36 **2.5 Primitive System Data Types**

37 ⇒ **2.5 Primitive System Data Types** *Add the following text at the end of the*
 38 *first paragraph, starting "Some data types used by..."*

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

43 ⇒ **2.5 Primitive System Data Types** *In the second paragraph, replace "All of*
 44 *the types listed in Table 2-1 ..." by the following:*

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

46 ⇒ **2.5 Primitive System Data Types** *Add the following datatypes to the list of*
 47 *types for which there are no defined comparison or assignment operations:*

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

50 ⇒ **2.5 Primitive System Data Types** *Add the following paragraphs after the*
 51 *paragraph starting “There are no defined comparison...”:*

52 An implementation need not provide the types *pthread_barrier_t* and
 53 *pthread_barrierattr_t* unless the Barriers option is supported (see 2.9.3). 8

54 An implementation need not provide the types *pthread_rwlock_t* and
 55 *pthread_rwlockattr_t* unless the Reader Writer Locks option is supported (see 8
 56 2.9.3). 8

57 An implementation need not provide the type *pthread_spinlock_t* unless the
 58 Spin Locks option is supported (see 2.9.3). 8

59 ⇒ **2.5 Primitive System Data Types** *Add the following table, and renumber*
 60 *subsequent tables in this Section accordingly:*

61 **Table 2-2 – Optional Primitive System Data Types**

63 Defined Type	Description	Implementation Option
65 <i>pthread_barrier_t</i>	Used to identify a barrier	Barriers option
66 <i>pthread_barrierattr_t</i>	Used to define a barrier attributes object	Barriers option
68 <i>pthread_rwlock_t</i>	Used to identify a reader/writer lock	Reader Writer Locks option
69 <i>pthread_rwlockattr_t</i>	Used to define a reader/writer lock attributes object	Reader Writer Locks option
70 <i>pthread_spinlock_t</i>	Used to identify a spin lock	Spin Locks option

73 2.7 C Language Definitions

74 2.7.3 Headers and Function Prototypes

75 ⇒ **2.7.3 Headers and Function Prototypes** *Add the following text after the*
 76 *sentence “For other functions in this part of ISO/IEC 9945, the prototypes or*
 77 *declarations shall appear in the headers listed below.”:*

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

81	⇒ 2.7.3 Headers and Function Prototypes	<i>Modify the contents of subclause</i>	
82		<i>2.7.3 to add the following optional functions, at the end of the current list of</i>	8
83		<i>headers and functions.</i>	8
84	If the Typed Memory Objects option is supported:		8
85	<code><sys/mman.h></code>	<i>posix_typed_mem_open(), posix_mem_offset(),</i>	
86		<i>posix_typed_mem_get_info()</i>	
87	If the Spin Locks option is supported:		8
88	<code><pthread.h></code>	<i>pthread_spin_init(), pthread_spin_destroy(),</i>	
89		<i>pthread_spin_lock(), pthread_spin_trylock(),</i>	
90		<i>pthread_spin_unlock()</i>	
91			9
92	If the Barriers option is supported:		8
93	<code><pthread.h></code>	<i>pthread_barrierattr_init(),</i>	
94		<i>pthread_barrierattr_destroy(),</i>	
95		<i>pthread_barrierattr_getpshared(),</i>	
96		<i>pthread_barrierattr_setpshared(),</i>	
97		<i>pthread_barrier_init(), pthread_barrier_destroy(),</i>	
98		<i>pthread_barrier_wait()</i>	
99	If the Reader/Writer Locks option is supported:		8
100	<code><pthread.h></code>	<i>pthread_rwlockattr_init(), pthread_rwlockattr_destroy(),</i>	
101		<i>pthread_rwlockattr_getpshared(),</i>	
102		<i>pthread_rwlockattr_setpshared(), pthread_rwlock_init(),</i>	
103		<i>pthread_rwlock_destroy(), pthread_rwlock_rdlock(),</i>	
104		<i>pthread_rwlock_tryrdlock(),</i>	
105		<i>pthread_rwlock_timedrdlock(), pthread_rwlock_wrlock(),</i>	
106		<i>pthread_rwlock_trywrlock(),</i>	
107		<i>pthread_rwlock_timedwrlock(), pthread_rwlock_unlock()</i>	9
108	If the Clock Selection option is supported:		9
109	<code><time.h></code>	<i>clock_nanosleep()</i>	
110	<code><pthread.h></code>	<i>pthread_condattr_setclock(), pthread_condattr_getclock()</i>	9

111 **2.8 Numerical Limits** 8

112 **2.8.7 Maximum Values** 8

113 ⇒ **2.8.7 Maximum Values** *In Table 2-7a, replace the description of {_POSIX_-* 8
 114 *CLOCKRES_MIN}, currently reading “The CLOCK_REALTIME clock resolution,* 8
 115 *in nanoseconds”, with the following:* 8

116 The resolution of the clocks CLOCK_REALTIME and CLOCK_MONOTONIC (if 8
 117 supported), in nanoseconds 8

118 **2.9 Symbolic Constants**

119 **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**

120 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** 8
 121 *Change the first words in the first paragraph, currently saying “The constants* 8
 122 *in Table 2-10 may be used...” to the following:* 8

123 The constants in Table 2-10 and Table 2-11 may be used... 8

124 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** 8
 125 *Add the following sentence at the end of the first paragraph:* 8

126 If any of the constants in Table 2-11 is defined, it shall be defined with the 8
 127 value shown in that Table. This value represents the version of the associated 8
 128 option that is supported by the implementation. 8

129 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** 8
 130 *Add Table 2-11, shown below, after Table 2-10 renumbering all subsequent* 8
 131 *tables accordingly.* 8

132 NOTE: (Editor’s note) The value 199ymmL corresponds to the date of approval of IEEE P1003.1j. 8

133 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
 134 *Add the following paragraphs:*

135 If the symbol {_POSIX_BARRIERS} is defined, then the symbols {_POSIX_-
 136 THREADS} and {_POSIX_THREAD_SAFE_FUNCTIONS} shall also be defined. If
 137 the symbol {_POSIX_READER_WRITER_LOCKS} is defined, then the symbols
 138 {_POSIX_THREADS} and {_POSIX_THREAD_SAFE_FUNCTIONS} shall also be
 139 defined. If the symbol {_POSIX_SPIN_LOCKS} is defined, then the symbols
 140 {_POSIX_THREADS} and {_POSIX_THREAD_SAFE_FUNCTIONS} shall also be
 141 defined.

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

144 If the symbol {_POSIX_CLOCK_SELECTION} is defined, then the symbol 9
 145 {_POSIX_TIMERS} shall also be defined.

Section 3: Process Primitives

1 **3.1 Process Creation and Execution**

2 **3.1.2 Execute a File**

3 ⇒ **3.1.2.2 Execute a File—Description** *Add the following paragraph after the*
 4 *paragraph starting “If the Memory Mapped Files or Shared Memory Objects*
 5 *option ...”*

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

9 **3.2 Process Termination**

10 **3.2.2 Terminate a Process**

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

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

16 **3.3 Signals**

17 **3.3.8 Synchronously Accept a Signal**

18 ⇒ **3.3.8.2 Synchronously Accept a Signal—description** *Add the following*
19 *text at the end of the paragraph starting “The function `sigtimedwait()` behaves*
20 *the same as ...”*

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

Section 4: Process Environment

1 4.8 Configurable System Variables

2 4.8.1 Get Configurable System Variables

3 ⇒ **4.8.1.2 Get Configurable System Variables— Description** *Add the follow-*
 4 *ing text after the sentence “The implementation shall support all of the vari-*
 5 *ables listed in Table 4-2 and may support others”, in the second paragraph:*

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

9 ⇒ **4.8.1.2 Get Configurable System Variables— Description** *In the second*
 10 *paragraph, replace the text “The variables in Table 4-2 come from ...” by the*
 11 *following:*

12 “The variables in Table 4-2 and Table 4-3 come from ...”

13 ⇒ **4.8.1.2 Get Configurable System Variables— Description** *Add the follow-*
 14 *ing table:*

15 **Table 4-3 – Optional Configurable System Variables**

16 Variable	17 <i>name</i> Value
18 {_POSIX_BARRIERS}	{_SC_BARRIERS}
19 {_POSIX_READER_WRITER_LOCKS}	{_SC_READER_WRITER_LOCKS}
20 {_POSIX_SPIN_LOCKS}	{_SC_SPIN_LOCKS}
21 {_POSIX_TYPED_MEMORY_OBJECTS}	{_SC_TYPED_MEMORY_OBJECTS}
22 {_POSIX_MONOTONIC_CLOCK}	{_SC_MONOTONIC_CLOCK}
23 {_POSIX_CLOCK_SELECTION}	{_SC_CLOCK_SELECTION}

9

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 5: Files and Directories

1 5.6 File Characteristics

2 5.6.1 File Characteristics: Header and Data Structure

3 8

4 ⇒ **5.6.1.1 <sys/stat.h> File Types** *Add the following text and macro after* 8
 5 *S_TYPEISSTM:* 8

6 If the Typed Memory Objects option is supported, the implementation may 8
 7 implement typed memory objects as distinct file types, and the following macro 8
 8 shall test whether a file is of the specified type: 8

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

10 5.6.2 Get File Status

11 ⇒ **5.6.2.2 Get File Status—Description** *Replace the text “If the Shared* A
 12 *Memory Objects option is supported and *fildev* references a shared memory* A
 13 *object,” by the following:* A

14 If the Shared Memory Objects option is supported and *fildev* references a A
 15 shared memory object or the Typed Memory Objects option is supported and A
 16 *fildev* references a typed memory object, A

17 5.6.4 Change File Modes

18 ⇒ **5.6.4.2 Change File Modes—Description** *Add the following paragraph*
 19 *before the paragraph starting “If the calling process does not have appropriate*
 20 *privileges...”:*

21 If `{_POSIX_TYPED_MEMORY_OBJECTS}` is defined and *fildev* references a typed 9
 22 memory object, the behavior of *fchmod()* is unspecified.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 6: Input and Output Primitives

1 **6.3 File Descriptor Deassignment**

2 **6.3.1 Close a File**

3 ⇒ **6.3.1.2 Close a File—Description** *In the `close()` function, replace the para-*
 4 *graph starting “If a memory object remains referenced...” by the following:*

5 If a shared memory object or a memory mapped file remains referenced
 6 at the last close (i.e., a process has it mapped), then the entire contents
 7 of the memory object shall persist until the memory object becomes
 8 unreferenced. If this is the last close of a shared memory object or a
 9 memory mapped file and the close results in the memory object becom-
 10 ing unreferenced, and the memory object has been unlinked, then the
 11 memory object shall be removed.

12 **6.4 Input and Output**

13 **6.4.1 Read from a File**

14 ⇒ **6.4.1.2 Read from a File—description** *Add the following text at the end of*
 15 *the description of the `read()` function:*

16 If the Typed Memory Objects option is supported:

17 If *fildev* refers to a typed memory object, the result of the *read()* function
 18 is unspecified.

19 **6.4.2 Write to a File**

20 ⇒ **6.4.2.2 Write to a File—Description** *Add the following text at the end of the*
 21 *description of the write() function:*

22 If the Typed Memory Objects option is supported:

23 If *fildev* refers to a typed memory object, the result of the *write()* func-
 24 tion is unspecified.

25 **6.5 Control Operations on Files**

26 **6.5.2 File Control**

27 ⇒ **6.5.2.2 File Control—Description** *Add the following text at the end of the*
 28 *description of the fcntl() function:*

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

31 **6.5.3 Reposition Read/Write File Offset**

32 ⇒ **6.5.3.2 Reposition Read/Write File Offset—Description** *Add the follow-*
 33 *ing text at the end of the description of the lseek() function:*

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

36 **6.7 Asynchronous Input and Output**

37 **6.7.8 Wait for an Asynchronous I/O Request**

38 ⇒ **6.7.8.2 Wait for an Asynchronous I/O Request—Description** *In the*
 39 *description of the aio_suspend() function, add the following text at the end of*
 40 *the paragraph starting “If the time interval indicated in ...”:*

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

Section 8: Language-Specific Services for the C Language

1 **8.2 C Language Input/Output Functions**

2 **8.2.2 Open a Stream on a File Descriptor**

3 ⇒ **8.2.2.2 Open a Stream on a File Descriptor—Description** *Add the follow-*
4 *ing text at the end of the description of the `fdopen()` function:*

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 11: Synchronization

1

8

2 11.4 Condition Variables

3 11.4.1 Condition Variable Initialization Attributes

4 ⇒ **11.4.1.1 Condition Variable Initialization Attributes—Synopsis** *Add*
5 *the following function synopses:*

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

```
8 int pthread_condattr_setclock(pthread_condattr_t *attr,  
9 clockid_t clock_id);
```

10 ⇒ **11.4.1.2 Condition Variable Initialization Attributes—Description** *Add*
11 *the following text before the “Otherwise” clause:*

12 If `{_POSIX_CLOCK_SELECTION}` is defined, the implementation shall provide 9
13 the `clock` attribute and the associated functions `pthread_condattr_setclock()`
14 and `pthread_condattr_getclock()`. The `clock` attribute is the clock id of the 8
15 clock that shall be used to measure the timeout service of
16 `pthread_cond_timedwait()`. The default value of the `clock` attribute shall
17 refer to the system clock.

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

24 ⇒ **11.4.1.2 Condition Variable Initialization Attributes—Description** *Add*
 25 *the `pthread_condattr_getclock()` and `pthread_condattr_setclock()` functions to*
 26 *the “Otherwise” list.*

27 ⇒ **11.4.1.3 Condition Variable Initialization Attributes—Returns** *Add the*
 28 *`pthread_condattr_setclock()` function to the list of functions appearing in the*
 29 *first paragraph. In addition, add the following paragraph:*

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

34 ⇒ **11.4.1.4 Condition Variable Initialization Attributes—Errors** *Add the*
 35 *`pthread_condattr_setclock()` and `pthread_condattr_getclock()` functions to the*
 36 *list of functions for which the error value [EINVAL] is returned if the implemen-*
 37 *tation detects that the value specified by `attr` is invalid. In addition, add the*
 38 *following text at the end of this subclause:*

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

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

44 ⇒ **11.4.1.5 Condition Variable Initialization Attributes—**
 45 **Cross-References** *Add the following cross-reference:*

46 `pthread_cond_timedwait()`, 11.4.4.

47 **11.4.4 Waiting on a Condition**

48 ⇒ **11.4.4.2 Waiting on a Condition—Description** *add the following text after*
 49 *the sentence starting “The `pthread_cond_timedwait` function is the same as*
 50 *...”:*

51 If `{_POSIX_CLOCK_SELECTION}` is defined, the condition variable shall have a 9
 52 `clock` attribute which specifies the clock that shall be used to measure the
 53 time specified by the `abstime` argument.

54 ⇒ **11 Synchronization** *Add these subclauses:*

55 **11.5 Barriers**

56 **11.5.1 Barrier Initialization Attributes**

57 **Functions:** `pthread_barrierattr_init()`, `pthread_barrierattr_destroy()`,
58 `pthread_barrierattr_getpshared()`, `pthread_barrierattr_setpshared()`.

59 **11.5.1.1 Synopsis**

```
60 #include <sys/types.h>
61 #include <pthread.h>
62 int pthread_barrierattr_init(pthread_barrierattr_t *attr);
63 int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
64 int pthread_barrierattr_getpshared(const pthread_barrierattr_t *attr,
65                                 int *pshared);
66 int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
67                                 int pshared);
```

68 **11.5.1.2 Description**

69 If `{_POSIX_BARRIERS}` is defined:

9

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

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

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

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

83 If `{_POSIX_THREAD_PROCESS_SHARED}` is defined, the implementation 9
84 shall provide the attribute `process-shared` and the associated functions
85 `pthread_barrierattr_getpshared()` and `pthread_barrierattr_setpshared()`. 8
86 The `process-shared` attribute is set to `PTHREAD_PROCESS_SHARED` to
87 permit a barrier to be operated upon by any thread that has access to the
88 memory where the barrier is allocated. If the `process-shared` attribute is

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

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

100

101 Additional attributes, their default values, and the names of the associated func-
 102 tions to get and set those attribute values are implementation defined.

9

103 11.5.1.3 Returns

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

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

111 11.5.1.4 Errors

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

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

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

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

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

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

127 **11.5.1.5 Cross-References**128 *pthread_barrier_init()*, 11.5.2.129 **11.5.2 Initialize/Destroy a Barrier**130 Functions: *pthread_barrier_init()*, *pthread_barrier_destroy()*.131 **11.5.2.1 Synopsis**

```

132 #include <sys/types.h>
133 #include <pthread.h>
134 int pthread_barrier_init(pthread_barrier_t *barrier,
135                          const pthread_barrierattr_t *attr,
136                          unsigned int count);
137 int pthread_barrier_destroy(pthread_barrier_t *barrier);

```

138

8

139 **11.5.2.2 Description**140 If `{_POSIX_BARRIERS}` is defined:

9

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

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

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

155 Only the object referenced by *barrier* may be used for performing synchron- 9
 156 ization. The result of referring to copies of that object in calls to 9
 157 *pthread_barrier_destroy()* or *pthread_barrier_wait()* is undefined. 9

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

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

165

8

166

9

167 **11.5.2.3 Returns**

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

171 **11.5.2.4 Errors**

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

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

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

177 [ENOMEM] Insufficient memory exists to initialize the barrier.

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

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

183 [EINVAL] The value specified by `attr` is invalid.

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

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

189 [EINVAL] The value specified by `barrier` is invalid.

190 **11.5.2.5 Cross-References**

191 `pthread_barrier_wait()`, 11.5.3.

192 **11.5.3 Synchronize at a Barrier**

193 Functions: `pthread_barrier_wait()`.

194 **11.5.3.1 Synopsis**

```

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

```

198 **11.5.3.2 Description**

199 If `{_POSIX_BARRIERS}` is defined:

9

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

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

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

212 The results are undefined if this function is called with an uninitialized bar-
 213 rier.

214 If a signal is delivered to a thread blocked on a barrier, upon return from 8
 215 the signal handler the thread shall resume waiting at the barrier if the bar- 8
 216 rier wait has not completed (that is, if the required number of threads have 8
 217 not arrived at the barrier during the execution of the signal handler); other- 8
 218 wise, the thread shall continue as normally from the completed barrier 8
 219 wait. Until the thread in the signal handler returns from it, it is 8
 220 unspecified whether other threads may proceed past the barrier once they 8
 221 have all reached it. 8

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

227 9

228 **11.5.3.3 Returns**

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

233 11.5.3.4 Errors

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

236 [EINVAL] The value specified by *barrier* does not refer to an initialized bar-
237 rier object.

238 11.5.3.5 Cross-References

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

240 11.6 Reader/Writer Locks

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

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

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

256 11.6.1 Reader/Writer Lock Initialization Attributes

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

259 11.6.1.1 Synopsis

```
260 #include <sys/types.h>
261 #include <pthread.h>
262 int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
263 int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
264 int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
265                                 int *pshared);
266 int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
267                                 int pshared);
```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

268 **11.6.1.2 Description**

269 If `{_POSIX_READER_WRITER_LOCKS}` is defined: 9

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

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

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

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

284 If `{_POSIX_THREAD_PROCESS_SHARED}` is defined, the the implementation 9
285 shall provide the attribute `process-shared` and the associated functions
286 *pthread_rwlockattr_getpshared()* and *pthread_rwlockattr_setpshared()*. If
287 this option is not supported, then the `process-shared` attribute and these
288 functions are not supported. The `process-shared` attribute is set to
289 `PTHREAD_PROCESS_SHARED` to permit a reader/writer lock to be operated
290 upon by any thread that has access to the memory where the reader/writer
291 lock is allocated. If the `process-shared` attribute is
292 `PTHREAD_PROCESS_PRIVATE`, the reader/writer lock shall only be operated
293 upon by threads created within the same process as the thread that initial-
294 ized the reader/writer lock; if threads of different processes attempt to
295 operate on such a reader/writer lock, the behavior is undefined. The default
296 value of the attribute shall be `PTHREAD_PROCESS_PRIVATE`.

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

301 9
302 Additional attributes, their default values, and the names of the associated func-
303 tions to get and set those attribute values are implementation defined.

304 **11.6.1.3 Returns**

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

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

312 **11.6.1.4 Errors**

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

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

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

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

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

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

328 **11.6.1.5 Cross-References**

329 *pthread_rwlock_init()*, 11.6.2.

330 **11.6.2 Initialize/Destroy a Reader/Writer Lock**

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

332 **11.6.2.1 Synopsis**

```
333 #include <sys/types.h>
334 #include <pthread.h>
335 int pthread_rwlock_init(pthread_rwlock_t *lock,
336                        const pthread_rwlockattr_t *attr);
337 int pthread_rwlock_destroy(pthread_rwlock_t *lock);
```

338 8

339 **11.6.2.2 Description**

340 If `{_POSIX_READER_WRITER_LOCKS}` is defined: 9

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

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

351 Only the object referenced by *lock* may be used for performing synchroniza- 9
 352 tion. The result of referring to copies of that object in calls to 9
 353 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, 9
 354 *pthread_rwlock_timedrdlock()*, *pthread_rwlock_tryrdlock()*, 9
 355 *pthread_rwlock_wrlock()*, *pthread_rwlock_timedwrlock()*, 9
 356 *pthread_rwlock_trywrlock()*, or *pthread_rwlock_unlock()* is undefined. 9

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

364 8
 365 9

366 11.6.2.3 Returns

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

370 11.6.2.4 Errors

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

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

375 [ENOMEM] Insufficient memory exists to initialize the lock.

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

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

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

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

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

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

388 11.6.2.5 Cross-References

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

393 11.6.3 Apply a Read Lock

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

396 11.6.3.1 Synopsis

```
397 #include <sys/types.h>
398 #include <time.h>
399 #include <pthread.h>
400 int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
401 int pthread_rwlock_timedrdlock(pthread_rwlock_t *lock,
402                               const struct timespec *abs_timeout);
403 int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
```

404 11.6.3.2 Description

405
406 If `{_POSIX_READER_WRITER_LOCKS}` is defined:

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

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

423 The maximum number of simultaneous read locks that an implementation 8
 424 guarantees can be applied to a reader/writer lock shall be implementation- 8
 425 defined. The *pthread_rwlock_rdlock()* function may fail if this maximum 8
 426 would be exceeded. 8

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

432 The results are undefined if any of these functions is called with an unini-
 433 tialized reader/writer lock.

434 If a signal that causes a signal handler to be executed is delivered to a
 435 thread blocked on a reader/writer lock via a call to
 436 *pthread_rwlock_rdlock()*, upon return from the signal handler the thread 8
 437 shall resume waiting for the lock as if it was not interrupted. 8

438 9
 439 If `{_POSIX_READER_WRITER_LOCKS}` and `{_POSIX_TIMEOUTS}` are both defined: 9

440 The *pthread_rwlock_timedrdlock()* function applies a read lock to the
 441 reader/writer lock referenced by *lock* as in the *pthread_rwlock_rdlock()*
 442 function. However, if the lock cannot be acquired without waiting for other A
 443 threads to unlock the lock, this wait shall be terminated when the specified A
 444 timeout expires. The timeout expires when the absolute time specified by A
 445 *abs_timeout* passes, as measured by the clock on which timeouts are based 8
 446 (that is, when the value of that clock equals or exceeds *abs_timeout*), or if 8
 447 the absolute time specified by *abs_timeout* has already been passed at the 8
 448 time of the call. If `{_POSIX_TIMERS}` is defined, the timeout is based on the 9
 449 `CLOCK_REALTIME` clock; if `{_POSIX_TIMERS}` is not defined, the timeout is 8
 450 based on the system clock as returned by the *time()* function. The resolu- 8
 451 tion of the timeout is the resolution of the clock on which it is based. The
 452 *timespec* datatype is defined as a structure in the header `<time.h>`. Under
 453 no circumstances shall the function fail with a timeout if the lock can be
 454 acquired immediately. The validity of the *abs_timeout* parameter need not
 455 be checked if the lock can be immediately acquired.

456 If a signal that causes a signal handler to be executed is delivered to a
 457 thread blocked on a reader/writer lock via a call to
 458 *pthread_rwlock_timedrdlock()*, upon return from the signal handler the 8
 459 thread shall resume waiting for the lock as if it was not interrupted. 8

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

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

464 11.6.3.3 Returns

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

468 11.6.3.4 Errors

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

471 [EBUSY] A writer holds the lock, or a writer with appropriate priority is 9
 472 blocked on the lock. 9

473 If any of the following conditions occur, the *pthread_rwlock_timedrdlock()* func-
 474 tion shall return the corresponding value:

475 [ETIMEDOUT]
 476 The lock could not be acquired before the specified timeout 8
 477 expired. 8

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

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

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

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

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

491 [EAGAIN] The read lock could not be acquired because the maximum
 492 number of read locks for *lock* would be exceeded. 8

493 11.6.3.5 Cross-References

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

497 **11.6.4 Apply a Write Lock**

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

500 **11.6.4.1 Synopsis**

```
501 #include <sys/types.h>
502 #include <time.h>
503 #include <pthread.h>
504 int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
505 int pthread_rwlock_timedwrlock(pthread_rwlock_t *lock,
506                               const struct timespec *abs_timeout);
507 int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
```

508 **11.6.4.2 Description**

509 If `{_POSIX_READER_WRITER_LOCKS}` is defined: 9

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

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

522 The results are undefined if any of these functions is called with an unini-
 523 tialized reader/writer lock.

524 If a signal that causes a signal handler to be executed is delivered to a
 525 thread blocked on a reader/writer lock via a call to
 526 *pthread_rwlock_wrlock()*, upon return from the signal handler the thread
 527 shall resume waiting for the lock as if it was not interrupted. 8 8

528 9
 529 If `{_POSIX_READER_WRITER_LOCKS}` and `{_POSIX_TIMEOUTS}` are both defined: 9

530 The *pthread_rwlock_timedwrlock()* function applies a write lock to the
 531 reader/writer lock referenced by *lock* as in the *pthread_rwlock_wrlock()*
 532 function. However, if the lock cannot be acquired without waiting for other
 533 threads to unlock the lock, this wait shall be terminated when the specified
 534 timeout expires. The timeout expires when the absolute time specified by
 535 *abs_timeout* passes, as measured by the clock on which timeouts are based
 536 (that is, when the value of that clock equals or exceeds *abs_timeout*), or if 8 8

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

537 the absolute time specified by *abs_timeout* has already been passed at the 8
 538 time of the call. If `{_POSIX_TIMERS}` is defined, the timeout is based on the 9
 539 `CLOCK_REALTIME` clock; if `{_POSIX_TIMERS}` is not defined, the timeout is 9
 540 based on the system clock as returned by the *time()* function. The resolu- 8
 541 tion of the timeout is the resolution of the clock on which it is based. The 8
 542 *timespec* datatype is defined as a structure in the header `<time.h>`. Under
 543 no circumstances shall the function fail with a timeout if the lock can be
 544 acquired immediately. The validity of the *abs_timeout* parameter need not
 545 be checked if the lock can be immediately acquired.

546 If a signal that causes a signal handler to be executed is delivered to a
 547 thread blocked on a reader/writer lock via a call to
 548 *pthread_rwlock_timedwrlock()*, upon return from the signal handler the 8
 549 thread shall resume waiting for the lock as if it was not interrupted. 8

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

553 9

554 11.6.4.3 Returns

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

558 11.6.4.4 Errors

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

561 [EBUSY] A reader or writer holds the lock.

562 If any of the following conditions occur, the *pthread_rwlock_timedwrlock()* func-
 563 tion shall return the corresponding value:

564 [ETIMEDOUT]
 565 The lock could not be acquired before the specified timeout 8
 566 expired. 8

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

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

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

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

577 **11.6.4.5 Cross-References**

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

581 **11.6.5 Unlock a Reader/Writer Lock**

582 Function: *pthread_rwlock_unlock()*. 9

583 **11.6.5.1 Synopsis**

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

587 **11.6.5.2 Description**

588 A

589 If `{_POSIX_READER_WRITER_LOCKS}` is defined: 9

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

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

609 The results are undefined if any of these functions are called with an unini-
 610 tialized reader/writer lock.

611 9

612 **11.6.5.3 Returns**

613 Upon successful completion, the *pthread_rwlock_unlock()* function shall return 9
 614 zero. Otherwise, an error number shall be returned to indicate the error.

615 **11.6.5.4 Errors**

616 For each of the following conditions, if the condition is detected, the
 617 *pthread_rwlock_unlock()* function shall return the corresponding value: 9

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

620 [EPERM] The calling thread does not hold a lock on the reader/writer lock. 9

621 **11.6.5.5 Cross-References**

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

626 **11.7 Spin Locks**

627 **11.7.1 Initialize/Destroy a Spin Lock**

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

629 **11.7.1.1 Synopsis**

```
630 #include <sys/types.h>
631 #include <pthread.h>
632 int pthread_spin_init(pthread_spinlock_t *lock, int pshared); 8
633 int pthread_spin_destroy(pthread_spinlock_t *lock);
634 8
```

635 **11.7.1.2 Description**

636 If `{_POSIX_SPIN_LOCKS}` is defined: 9

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

640 If `{_POSIX_THREAD_PROCESS_SHARED}` is defined: 9

681 [ENOMEM] Insufficient memory exists to initialize the lock.

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

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

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

689 11.7.1.5 Cross-References

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

692 11.7.2 Lock a Spin Lock

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

694 11.7.2.1 Synopsis

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

699 11.7.2.2 Description

700 If `{_POSIX_SPIN_LOCKS}` is defined:

9

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

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

708 The results are undefined if any of these functions is called with an unini-
709 tialized spin lock.

710

9

711 11.7.2.3 Returns

712 Upon successful completion, the *pthread_spin_lock()* and *pthread_spin_trylock()*
713 functions shall return zero. Otherwise, an error number shall be returned to indi-
714 cate the error.

715 **11.7.2.4 Errors**

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

718 [EBUSY] A thread currently holds the lock.

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

721 [EDEADLK] The calling thread already holds the lock.

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

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

727 **11.7.2.5 Cross-References**

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

730 **11.7.3 Unlock a Spin Lock**

731 Function: *pthread_spin_unlock()*.

732 **11.7.3.1 Synopsis**

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

736 **11.7.3.2 Description**

737 If `{_POSIX_SPIN_LOCKS}` is defined:

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

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

746

9

9

747 **11.7.3.3 Returns**

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

750 **11.7.3.4 Errors**

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

753 [EINVAL] An invalid argument was specified.

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

755 **11.7.3.5 Cross-References**

756 *pthread_spin_init()*, 11.7.1; *pthread_spin_destroy()*, 11.7.1; *pthread_spin_lock()*,
757 11.7.2; *pthread_spin_trylock()*, 11.7.2.

Section 12: Memory Management

1 ⇒ **12 Memory Management** *Replace the first paragraph with:*

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

5 ⇒ **12 Memory Management** *Add the following new paragraphs after the para-*
6 *graph that begins with “An unlink() of a file...” and ends with “...of the*
7 *memory object mapped.”:*

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

25 **12.2 Memory Mapping Functions**

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

26 12.2.1 Map Process Addresses to a Memory Object

27 ⇒ 12.2.1.2 Map Process Addresses to a Memory Object—Description

28 *Replace the first paragraph with:*

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

31 ⇒ **12.2.1.2 Map Process Addresses to a Memory Object—Description** *In*
32 *the paragraph beginning with “The `mmap()` function establishes...” and ending*
33 *“...object represented by `fildev`.”, replace the last sentence (beginning “The*
34 *range of bytes starting...”) with:*

35 The range of bytes starting at *off* and continuing for *len* bytes shall be legiti-
36 mate for the possible (not necessarily current) offsets in the file, shared
37 memory object, or typed memory object represented by *fildev*. If *fildev*
38 represents a typed memory object opened with either the
39 `POSIX_TYPED_MEM_ALLOCATE` flag or the
40 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, the memory object to be mapped
41 shall be that portion of the typed memory object allocated by the implementa-
42 tion as specified below. In this case, if *off* is non-zero, the behavior of *mmap()* is
43 undefined. If *fildev* refers to a valid typed memory object that is not accessible
44 from the calling process, *mmap()* shall fail.

45 ⇒ **12.2.1.2 Map Process Addresses to a Memory Object—Description** *Add*
46 *the following new paragraph after the paragraph that begins with*
47 *“MAP_SHARED and MAP_PRIVATE describe...” and ends with “...is retained*
48 *across `fork()`.”:*

49 When *fildev* represents a typed memory object opened with either the
50 `POSIX_TYPED_MEM_ALLOCATE` flag or the
51 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, *mmap()* shall, if there are
52 enough resources available, map *len* bytes allocated from the corresponding
53 typed memory object which were not previously allocated to any process in any
54 processor that may access that typed memory object. If there are not enough
55 resources available, the function shall fail. If *fildev* represents a typed memory
56 object opened with the `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, these
57 allocated bytes shall be contiguous within the typed memory object. If *fildev*
58 represents a typed memory object opened with the
59 `POSIX_TYPED_MEM_ALLOCATE` flag, these allocated bytes may be composed of
60 non-contiguous fragments within the typed memory object. If *fildev* represents
61 a typed memory object opened with neither the
62 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag nor the
63 `POSIX_TYPED_MEM_ALLOCATE` flag, *len* bytes starting at offset *off* within the
64 typed memory object are mapped, exactly as when mapping a file or shared
65 memory object. In this case, if two processes map an area of typed memory
66 using the same *off* and *len* values and using file descriptors that refer to the
67 same memory pool (either from the same port or from a different port), both

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

68 processes shall map the same region of storage.

69 ⇒ **12.2.1.4 Map Process Addresses to a Memory Object—Errors** *Add to the*
70 *description of [ENOMEM] the following additional paragraph:*

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

73 ⇒ **12.2.1.4 Map Process Addresses to a Memory Object—Errors** *Add to the*
74 *description of [ENXIO] the following additional paragraph:*

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

77 ⇒ **12.2.1.5 Map Process Addresses to a Memory Object—**
78 **Cross-References** *Add the following cross-reference:*

79 *posix_typed_mem_open()*, 12.4.2.

80 **12.2.2 Unmap Previously Mapped Addresses**

81 ⇒ **12.2.2.2 Unmap Previously Mapped Addresses—Description** *Replace*
82 *the first paragraph with:*

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

85 ⇒ **12.2.2.2 Unmap Previously Mapped Addresses—Description** *Add the*
86 *following new paragraphs after the paragraph which begins with “Any memory*
87 *locks...” and ending with “...an appropriate call to munlock().”:*

88 If a mapping removed from a typed memory object causes the corresponding
89 address range of the memory pool to be inaccessible by any process in the sys-
90 tem except through allocatable mappings (i.e., mappings of typed memory
91 objects opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then
92 that range of the memory pool shall become deallocated and may become avail-
93 able to satisfy future typed memory allocation requests.

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

97 ⇒ **12.2.2.5 Unmap Previously Mapped Addresses—Cross-References** *Add*
 98 *the following cross-reference:*

99 *posix_typed_mem_open()*, 12.4.2.

100 **12.2.4 Memory Object synchronization**

101 ⇒ **12.2.4.2 Memory Object synchronization—Description** *Change the sen-*
 102 *tence “The effect of *msync()* on shared memory objects is unspecified.” to:*

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

105 ⇒ **12 Memory Management** *Add the following clause:*

106 **12.4 Typed Memory Functions**

107 **12.4.1 Data Definitions**

108 If `{_POSIX_TYPED_MEMORY_OBJECTS}` is defined, the header `<sys/mman.h>` 9
 109 shall define the memory information structure *posix_typed_mem_info*, which shall
 110 include at least the following member:

111	Member	Member	Description	
112	Type	Name		
113	<code>size_t</code>	<code>posix_tmi_length</code>	Maximum length which may be allocated from a typed memory	8
114			object.	

115 **12.4.2 Open a Typed Memory Object**

116 Function: *posix_typed_mem_open()*

117 **12.4.2.1 Synopsis**

```
118 #include <sys/mman.h>
119 int posix_typed_mem_open(const char *name, int oflag, int tflag); 8
```

120 **12.4.2.2 Description**121 If `{_POSIX_TYPED_MEMORY_OBJECTS}` is defined:

9

122 The *posix_typed_mem_open()* function establishes a connection between the
 123 typed memory object specified by the string pointed to by *name* and a file
 124 descriptor. It creates an open file description that refers to the typed
 125 memory object and a file descriptor that refers to that open file description.
 126 The file descriptor is used by other functions to refer to that typed memory
 127 object. It is unspecified whether the name appears in the file system and is
 128 visible to other functions that take pathnames as arguments. The *name*
 129 argument shall conform to the construction rules for a pathname. If *name*
 130 begins with the slash character, then processes calling
 131 *posix_typed_mem_open()* with the same value of *name* shall refer to the
 132 same typed memory object. If *name* does not begin with the slash charac-
 133 ter, the effect is implementation defined. The interpretation of slash char-
 134 acters other than the leading slash character in *name* is implementation
 135 defined.

136 Each typed memory object supported in a system is identified by a *name*
 137 which specifies not only its associated typed memory pool, but also the path
 138 or port by which it is accessed. That is, the same typed memory pool
 139 accessed via several different ports has several different corresponding
 140 names. The binding between *names* and typed memory objects is estab-
 141 lished in an implementation-defined manner. Unlike shared memory 8
 142 objects, there is ordinarily no way for a program to create a typed memory
 143 object.

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

147	Symbolic Constant	Description
148		
149	<code>POSIX_TYPED_MEM_ALLOCATE</code>	Allocate on <i>mmap()</i> .
150	<code>POSIX_TYPED_MEM_ALLOCATE_CONTIG</code>	Allocate contiguously on <i>mmap()</i> .
151	<code>POSIX_TYPED_MEM_MAP_ALLOCATABLE</code>	Map on <i>mmap()</i> , without affecting allo- catability.
152		

153 If *tflag* has the flag `POSIX_TYPED_MEM_ALLOCATE` specified, any subse-
 154 quent call to *mmap()* using the returned file descriptor shall result in allo-
 155 cation and mapping of typed memory from the specified typed memory pool.
 156 The allocated memory may be a contiguous previously unallocated area of
 157 the typed memory pool or several non-contiguous previously unallocated
 158 areas (mapped to a contiguous portion of the process address space). If *tflag*
 159 has the flag `POSIX_TYPED_MEM_ALLOCATE_CONTIG` specified, any subse-
 160 quent call to *mmap()* using the returned file descriptor shall result in allo-
 161 cation and mapping of a single contiguous previously unallocated area of
 162 the typed memory pool (also mapped to a contiguous portion of the process
 163 address space). If *tflag* has none of the flags
 164 `POSIX_TYPED_MEM_ALLOCATE` or
 165 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` specified, any subsequent call to

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

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

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

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

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

191 `O_RDONLY` Open for read access only.
 192 `O_WRONLY` Open for write access only.
 193 `O_RDWR` Open for read or write access.

194 Otherwise:

195 Either the implementation shall support the *posix_typed_mem_open()* func-
 196 tion as described above or this function shall not be provided.

197 **12.4.2.3 Returns**

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

201 **12.4.2.4 Errors**

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

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

206	[EINTR]	The <i>posix_typed_mem_open()</i> operation was interrupted by a signal.	
207			
208			8
209	[EINVAL]	The flags specified in <i>tflag</i> are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE,	
210		POSIX_TYPED_MEM_ALLOCATE_CONTIG,	or
211		or	
212		POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).	
213	[EMFILE]	Too many file descriptors are currently in use by this process.	
214	[ENAMETOOLONG]		
215		The length of the <i>name</i> string exceeds {PATH_MAX}, or a path-	
216		name component is longer than {NAME_MAX} while {_POSIX-	
217		NO_TRUNC} is in effect.	
218	[ENFILE]	Too many file descriptors are currently open in the system.	8
219	[ENOENT]	The named typed memory object does not exist.	
220			8
221	[EPERM]	The caller lacks the appropriate privilege to specify the flag	
222		POSIX_TYPED_MEM_MAP_ALLOCATABLE in argument <i>tflag</i> .	

223 12.4.2.5 Cross-References

224 *close()*, 6.3.1; *dup()*, 6.2.1; *exec*, 3.1.2; *fcntl()*, 6.5.2; <fcntl.h>, 6.5.1; *umask()*,
 225 5.3.3; *mmap()*, 12.2.1; <sys/mman.h>, 12.1.1.2; *posix_mem_offset()*, 12.4.3.

226 12.4.3 Find Offset and Length of a Mapped Typed Memory Block

227 Function: *posix_mem_offset()*

228 12.4.3.1 Synopsis

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

232 12.4.3.2 Description

233 If {_POSIX_TYPED_MEMORY_OBJECTS} is defined: 9

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

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

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

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

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

255 Otherwise:

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

258 12.4.3.3 Returns

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

261 12.4.3.4 Errors

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

264 [EACCES] The process has not mapped a memory object supported by this
 265 function at the given address *addr*.
 266

8

267 12.4.3.5 Cross-References

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

269 12.4.4 Query Typed Memory Information

270 Function: *posix_typed_mem_get_info()*

271 12.4.4.1 Synopsis

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

275 **12.4.4.2 Description**

276 If `{_POSIX_TYPED_MEMORY_OBJECTS}` is defined: 9

277 The *posix_typed_mem_get_info()* function returns, in the *posix_tmi_length* 8
 278 field of the *posix_typed_mem_info* structure pointed to by *info*, the max-
 279 imum length which may be successfully allocated by the typed memory
 280 object designated by *fildes*. This maximum length shall take into account
 281 the flag `POSIX_TYPED_MEM_ALLOCATE` or
 282 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` specified when the typed memory
 283 object represented by *fildes* was opened. The maximum length is dynamic;
 284 therefore, the value returned is valid only while the current mapping of the
 285 corresponding typed memory pool remains unchanged.

286 If *fildes* represents a typed memory object opened with neither the
 287 `POSIX_TYPED_MEM_ALLOCATE` flag nor the
 288 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag specified, the returned value
 289 of *info.posix_tmi_length* is unspecified. 8

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

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

295 Otherwise:

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

298 **12.4.4.3 Returns**

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

301 **12.4.4.4 Errors**

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

304 [EBADF] The *fildes* argument is not a valid open file descriptor.
 305 [ENODEV] The *fildes* argument is not connected to a memory object sup-
 306 ported by this function.
 307

308 **12.4.4.5 Cross-References**

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 14: Clocks and Timers

1 14.1 Data Definitions for Clocks and Timers

2 14.1.4 Manifest Constants

3 ⇒ **14.1.4 Manifest Constants** *Add the following text after the current*
4 *definitions of constants:*

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

7 `CLOCK_MONOTONIC`

8 The identifier for the systemwide monotonic clock, which
9 is defined as a clock whose value cannot be set via
10 `clock_settime()` and which cannot have backward clock
11 jumps. The maximum possible clock jump shall be imple-
12 mentation defined.

13 ⇒ **14.1.4 Manifest Constants** *Replace the paragraph starting “The maximum*
14 *allowable resolution for ...” and the following paragraph starting “The*
15 *minimum allowable maximum value ...” by the following text:*

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

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

31 14.2 Clock and Timer Functions

32 14.2.1 Clocks

33 ⇒ **14.2.1.2 Clocks—Description** *Add the following text after the paragraph*
 34 *starting “A clock may be systemwide ...”:*

35 If `{_POSIX_MONOTONIC_CLOCK}` is defined: 9

36 All implementations shall support a `clock_id` of `CLOCK_MONOTONIC`
 37 defined in 14.1.4. This clock represents the monotonic clock for the sys-
 38 tem. For this clock, the value returned by `clock_gettime()` represents
 39 the amount of time (in seconds and nanoseconds) since an unspecified
 40 point in the past (for example, system start-up time, or the Epoch). This
 41 point does not change after system start-up time. The value of the
 42 `CLOCK_MONOTONIC` clock cannot be set via `clock_settime()`. This func-
 43 tion shall fail if it is invoked with a `clock_id` argument of
 44 `CLOCK_MONOTONIC`.

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

49 ⇒ **14.2.1.2 Clocks—Description** *In the description of `clock_settime()`, add the*
 50 *following paragraphs after the text that describes the effects of setting a clock*
 51 *via `clock_settime()`.*

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

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

64 ⇒ **14.2.1.4 Clocks—Errors** *Add the following condition to the error conditions*
 65 *that shall cause `clock_settime()` to fail:*

66 [EINVAL] The value of the `clock_id` argument is CLOCK_MONOTONIC. 8

67 ⇒ **14.2.1.5 Clocks—Cross-References** *Add the following cross-references:*

68 `timer_create()`, 14.2.2; `timer_settime()`, 14.2.4; `nanosleep()`, 14.2.5;
 69 `clock_nanosleep()`, 14.2.6; `sem_timedwait()`, 11.2.6;
 70 `pthread_mutex_timedlock()`, 11.3.3; `mq_timedsend()`, 15.2.4;
 71 `mq_timedreceive()`, 15.2.5.

72 14.2.2 Create a Per-Process Timer

73 ⇒ **14.2.2.2 Create a Per-Process Timer—Description** *Add the following text*
 74 *at the end of the paragraph starting “Each implementation shall define a set of*
 75 *clocks that ...”:*

76 If `{_POSIX_CLOCK_SELECTION}` is defined, all implementations shall support a 9
 77 `clock_id` of CLOCK_MONOTONIC.

78 ⇒ **14.2 Clock and Timer Functions** *Add the following subclause:*

79 14.2.6 High Resolution Sleep with Specifiable Clock

80 Function: `clock_nanosleep()`

81 14.2.6.1 Synopsis

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

85 14.2.6.2 Description

86 If `{_POSIX_CLOCK_SELECTION}` is defined: 9

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

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

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

105 The suspension time caused by this function may be longer than requested
 106 because the argument value is rounded up to an integer multiple of the
 107 sleep resolution, or because of the scheduling of other activity by the sys-
 108 tem. But, except for the case of being interrupted by a signal, the suspen-
 109 sion time for the relative *clock_nanosleep()* function (i.e., with the
 110 `TIMER_ABSTIME` flag not set) shall not be less than the time interval
 111 specified by *rqtp*, as measured by the corresponding clock. The suspension
 112 for the absolute *clock_nanosleep()* function (i.e., with the `TIMER_ABSTIME`
 113 flag set) shall be in effect at least until the value of the corresponding clock
 114 reaches the absolute time specified by *rqtp*, except for the case of being
 115 interrupted by a signal.

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

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

121

9

122 14.2.6.3 Returns

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

125 If the *clock_nanosleep()* function returns because it has been interrupted by a sig-
 126 nal it shall return the corresponding error value. For the relative
 127 *clock_nanosleep()* function, if the *rmtp* argument is non-`NULL`, the *timespec* struc-
 128 ture referenced by it shall be updated to contain the amount of time remaining in
 129 the interval (the requested time minus the time actually slept). If the *rmtp* argu-
 130 ment is `NULL`, the remaining time is not returned. The absolute
 131 *clock_nanosleep()* function has no effect on the structure referenced by *rmtp*.

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

133 **14.2.6.4 Errors**

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

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

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

143

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

8

146 **14.2.6.5 Cross-References**

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 15: Message Passing

1 NOTE: The amendments to Section 15 have been removed from this draft due to the shift from 8
2 relative to absolute timeouts. The section is kept as a placeholder for the diff marks associated to 8
3 the deletion of this text, and thus will not appear in the final standard. 8

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 18: Thread Cancellation

1 18.1 Thread Cancellation Overview

2 ⇒ **18.1.2 Cancellation Points** *Add the following functions to the list of func-*
 3 *tions for which a cancellation point shall occur:*

4 *clock_nanosleep().* 8

5 ⇒ **18.1.2 Cancellation Points** *Add the following functions to the list of func-*
 6 *tions for which a cancellation point may also occur:*

7	<i>pthread_rwlock_rdlock(),</i>	<i>pthread_rwlock_timedrdlock(),</i>	8
8	<i>pthread_rwlock_wrlock(),</i>	<i>pthread_rwlock_timedwrlock(),</i>	8
9	<i>posix_typed_mem_open().</i>		8

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Annex A
(informative)
Bibliography

1 **A.4 Other Sources of Information**

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

4 {B79} George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The
5 Benjamin/Cummings Publishing Company, Inc., 1989, ISBN 0-8053-
6 0177-1.

7 {B80} Steven Brawer. *Introduction to Parallel Programming*. Academic Press,
8 1989, ISBN 0-12-128470-0.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Annex B (informative)

Rationale and Notes

1

2 **B.11 Synchronization**

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

4 **B.11.5 Barriers**

5 **B.11.5.1 Background**

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

10 The use of return value PTHREAD_BARRIER_SERIAL_THREAD is shown in the fol-
11 lowing example:

```
12     if ( (status=pthread_barrier_wait(&barrier)) ==
13         PTHREAD_BARRIER_SERIAL_THREAD) {
14         ...serial section
15     }
16     else if (status != 0) {
17         ...error processing
18     }
19     status=pthread_barrier_wait(&barrier);
20     ...
```

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

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

31 **B.11.5.2 Lack of Timeout Feature**

32 Alternate versions of most blocking routines have been provided to support watch-
 33 dog timeouts. No alternate interface of this sort has been provided for barrier
 34 waits for the following reasons:

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

46 **B.11.6 Reader/Writer Locks**

47 **B.11.6.1 Background**

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

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

59 **B.11.6.2 Queuing of Waiting Threads**

60 The *pthread_rwlock_unlock()* function description states that one writer or one or
 61 more readers shall acquire the lock if it is no longer held by any thread as a result
 62 of the call. However, the function does not specify which thread(s) acquire the
 63 lock, unless the Thread Execution Scheduling option is supported. 9

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

71 For example, suppose one or more readers hold a reader/writer lock and the fol-
72 lowing threads request the lock in the listed order:

73 *pthread_rwlock_wrlock()* - Low priority thread *writer_a*
74 *pthread_rwlock_rdlock()* - High priority thread *reader_a*
75 *pthread_rwlock_rdlock()* - High priority thread *reader_b*
76 *pthread_rwlock_rdlock()* - High priority thread *reader_c*

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

79 *pthread_rwlock_rdlock()* - Low priority thread *reader_a*
80 *pthread_rwlock_rdlock()* - Low priority thread *reader_b*
81 *pthread_rwlock_rdlock()* - Low priority thread *reader_c*
82 *pthread_rwlock_wrlock()* - Medium priority thread *writer_a*
83 *pthread_rwlock_rdlock()* - High priority thread *reader_d*

84 If priority scheduling is applied then *reader_d* would acquire the lock and *writer_a*
85 would block the remaining readers. But should the remaining readers also
86 acquire the lock to increase concurrency? The solution adopted takes into account
87 that when the Thread Execution Scheduling option is supported, high priority 9
88 threads may in fact starve low priority threads (the application developer is
89 responsible in this case to design the system in such a way that this starvation is
90 avoided). Therefore, the standard specifies that high priority readers take pre-
91 cedence over lower priority writers. However, to prevent writer starvation from
92 threads of the same or lower priority, writers take precedence over readers of the
93 same or lower priority.

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

104 **B.11.6.3 Comparison to ISO/IEC 9945-1 *fcntl()* locks** 8

105 The reader/writer locks and the *fcntl()* locks share a common goal: increasing 8
106 concurrency among readers, thus increasing throughput and decreasing delay. 8

107 However, the reader/writer locks have two features not present in the *fcntl()* 8
108 locks. First, under priority scheduling, reader/writer locks are granted in priority 8
109 order. Second, also under priority scheduling, writer starvation is prevented by 8
110 giving writers preference over readers of equal or lower priority. 8

111 Also, reader/writer locks can be used in systems lacking a file system, such as 8
112 those conforming to the minimal realtime system profile of the IEEE 1003.13 8
113 profile standard. 8

114 **B.11.6.4 History of Resolution Issues** 8

115 Based upon some balloting objections, the draft specified the behavior of threads 8
116 waiting on a reader/writer lock during the execution of a signal handler, as if the 8
117 thread had not called the lock operation. However, this specified behavior would 8
118 require implementations to establish internal signal handlers even though this 8
119 situation would be rare, or never happen for many programs. This would intro- 8
120 duce an unacceptable performance hit in comparison to the little additional func- 8
121 tionality gained. Therefore, the behavior of reader/writer locks and signals was 8
122 reverted back to its previous mutex-like specification. 8

123 **B.11.7 Spin Locks**

124 **B.11.7.1 Background**

125 Spin locks represent an extremely low-level synchronization mechanism suitable 8
126 primarily for use on shared memory multiprocessors. It is typically an atomically 8
127 modified boolean value that is set to one when the lock is held and to zero when 8
128 the lock is freed.

129 When a caller requests a spin lock that is already held, it typically spins in a loop 8
130 testing whether the lock has become available. Such spinning wastes processor 8
131 cycles so the lock should only be held for short durations and not across 8
132 sleep/block operations. Callers should unlock spin locks before calling sleep opera- 8
133 tions.

134 Spin locks are available on a variety of systems. Section 11.7 is an attempt to 8
135 standardize that existing practice. 8

136 **B.11.7.2 Lack of Timeout Feature**

137 Alternate versions of most blocking routines have been provided to support watch- 8
138 dog timeouts. No alternate interface of this sort has been provided for spin locks 8
139 for the following reasons:

- 140 1. It is impossible to determine appropriate timeout intervals for spin locks in a 8
141 portable manner. The amount of time one can expect to spend spin-waiting 8
142 is inversely proportional to the degree of parallelism provided by the system. 8
143 It can vary from a few cycles when each competing thread is running on its 8
144 own processor, to an indefinite amount of time when all threads are multi- 8
145 plexed on a single processor (which is why spin locking is not advisable on 8
146 uniprocessors).
- 147 2. When used properly, the amount of time the calling thread spends waiting 8
148 on a spin lock should be considerably less than the time required to set up a 8
149 corresponding watchdog timer. Since the primary purpose of spin locks it to 8
150 provide a low-overhead synchronization mechanism for multiprocessors, the 8
151 overhead of a timeout mechanism was deemed unacceptable.

152 It was also suggested that an additional *count* argument be provided (on the 8
153 *pthread_spin_lock()* call) in lieu of a true timeout so that a spin lock call could fail 8

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

```

157         int n = MAX_SPIN;                                     8
158         while ( --n >= 0 )                                   8
159         {                                                    8
160             if ( !pthread_spin_try_lock(...) )              8
161                 break;                                       8
162         }                                                    8
163         if ( n >= 0 )                                        8
164         {                                                    8
165             /* Successfully acquired the lock */            8
166         }                                                    8
167         else                                                8
168         {                                                    8
169             /* Unable to acquire the lock */                8
170         }                                                    8

```

171 **B.11.7.3 process-shared Attribute**

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

184 **B.11.7.4 Spin Locks vs. Mutexes**

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

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

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

199 Therefore, spin locks and mutexes may be implemented to have different charac-
 200 teristics. Spin locks may have lower overall overhead for very short term blocking,
 201 and mutexes may have lower overall overhead when a thread will be blocked for
 202 longer periods of time. The presence of both interfaces allows implementations
 203 with these two different characteristics, both of which may be useful to a particu-
 204 lar application.

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

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

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

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

222 **B.12 Memory Management**

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

224 **B.12.4 Typed Memory Functions**

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

230 **B.12.4.1 Model**

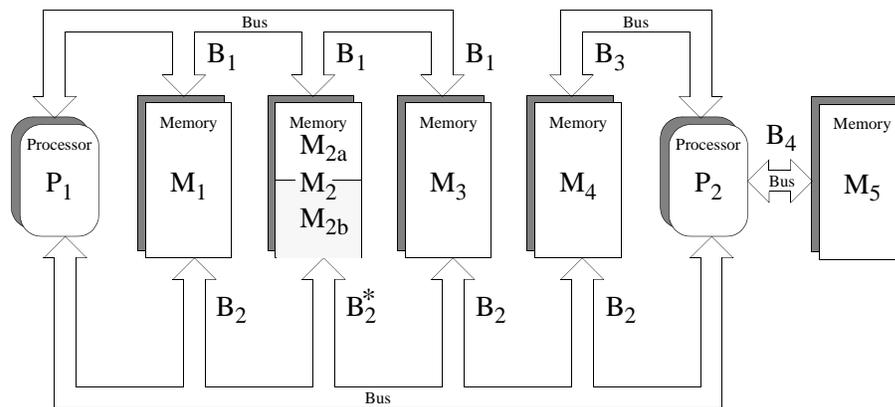
231 Realtime systems conforming to one of the POSIX.13 realtime profiles are expected
 232 (and desired) to be supported on systems with more than one type or pool of
 233 memory (e.g., SRAM, DRAM, ROM, EPROM, EEPROM), where each type or pool of
 234 memory may be accessible by one or more processors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

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

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

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

249



* 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

250

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

- 252 — *Typed memory allocation.* An application should be able to allocate memory
 253 dynamically from the desired pool using the desired bus, and map it into a
 254 process's address space. For example, processor P_1 can allocate some por-
 255 tion of memory pool M_1 through port B_1 , treating all unmapped subareas of
 256 M_1 as a heap-storage resource from which memory may be allocated. This
 257 portion of memory is mapped into the process's address space, and subse-
 258 quently deallocated when unmapped from all processes.
- 259 — *Using the same storage region from different busses.* An application process
 260 with a mapped region of storage that is accessed from one bus should be
 261 able to map that same storage area at another address (subject to page size
 262 restrictions detailed in 12.2.1.2), to allow it to be accessed from another bus.

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

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

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

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

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

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

306 **B.12.4.2 Existing Practice**

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

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

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

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

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

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

335 **B.12.4.3 Requirements**

336 Existing practice in SVID derived UNIX¹⁾ systems relies on functionality similar to
337 *mmap()* and its related interfaces to achieve mapping and allocation of typed
338 memory. However, the issue of sharing typed memory (allocated or mapped) and
339 the complication of multiple ports are not addressed in any consistent way by
340 existing UNIX system practice. Part of this functionality is existing practice in
341 specialized realtime operating systems. In order to solidify the capabilities implied by
342 the model above, the following requirements are imposed on the interface:

343 _____

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

8

8

- 345 — *Identification of typed memory pools and ports.* All processes (running in all
346 processors) in the system shall be able to identify a particular (system
347 configured) typed memory pool accessed through a particular (system
348 configured) port by a name. That name shall be a member of a namespace
349 common to all these processes, but need not be the same namespace as that
350 containing ordinary file names. The association between memory 8
351 pools/ports and corresponding names is typically established when the sys- 8
352 tem is configured. The “open” operation for typed memory objects should 8
353 be distinct from the *open()* function, for consistency with other similar ser-
354 vices, but implementable on top of *open()*. This implies that the handle for
355 a typed memory object will be a file descriptor.
- 356 — *Allocation and mapping of typed memory.* Once a typed memory object has
357 been identified by a process, it shall be possible to both map user-selected
358 subareas of that object into process address space and to map system-
359 selected (i.e., dynamically allocated) subareas of that object, with user-
360 specified length, into process address space. It shall also be possible to
361 determine the maximum length of memory allocation that may be
362 requested from a given typed memory object.
- 363 — *Sharing typed memory.* Two or more processes shall be able to share por-
364 tions of typed memory, either user-selected or dynamically allocated. This
365 requirement applies also to dynamically allocated regions of memory that
366 are composed of several non-contiguous pieces.
- 367 — *Contiguous allocation.* For dynamic allocation, it shall be the user’s option
368 whether the system is required to allocate a contiguous subarea within the
369 typed memory object, or whether it is permitted to allocate discontinuous
370 fragments which appear contiguous in the process mapping. Contiguous
371 allocation simplifies the process of sharing allocated typed memory, while
372 discontinuous allocation allows for potentially better recovery of deallocated
373 typed memory.
- 374 — *Accessing typed memory through different ports.* Once a subarea of a typed
375 memory object has been mapped, it shall be possible to determine the loca-
376 tion and length corresponding to a user-selected portion of that object
377 within the memory pool. This location and length can then be used to
378 remap that portion of memory for access from another port. If the refer-
379 enced portion of typed memory was allocated discontinuously, the length
380 thus determined may be shorter than anticipated, and the user code shall
381 adapt to the value returned.
- 382 — *Deallocation.* When a previously mapped subarea of typed memory is no
383 longer mapped by any process in the system—as a result of a call or calls to
384 *munmap()* —, that subarea shall become potentially reusable for dynamic
385 allocation; actual reuse of the subarea is a function of the dynamic typed
386 memory allocation policy.
- 387 — *Unallocated mapping.* It shall be possible to map user-selected subareas of
388 a typed memory object without marking that subarea as unavailable for
389 allocation. This option is not the default behavior, and shall require
390 appropriate privilege.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

391 **B.12.4.4 Scenario**

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

- 399 — *Allocating typed memory.* Process A calls `posix_typed_mem_open()` with the
 400 name `/typed.m2b-b1` and a `tflag` of `POSIX_TYPED_MEM_ALLOCATE` to get
 401 a file descriptor usable for allocating from pool M_{2b} accessed through port
 402 B_1 . It then calls `mmap()` with this file descriptor requesting a length of
 403 4096 bytes. The system allocates two discontinuous blocks of sizes 1024 and
 404 3072 bytes within M_{2b} . The `mmap()` function returns a pointer to a 4096
 405 byte array in process A 's logical address space, mapping the allocated blocks
 406 contiguously. Process A can then utilize the array, and store data in it.
- 407 — *Determining the location of the allocated blocks.* Process A can determine
 408 the lengths and offsets (relative to M_{2b}) of the two blocks allocated, by using
 409 the following procedure: First, process A calls `posix_mem_offset()` with the
 410 address of the first element of the array and length 4096. Upon return, the
 411 offset and length (1024 bytes) of the first block are returned. A second call
 412 to `posix_mem_offset()` is then made using the address of the first element of
 413 the array plus 1024 (the length of the first block), and a new length of
 414 4096-1024. If there were more fragments allocated, this procedure could
 415 have been continued within a loop until the offsets and lengths of all the
 416 blocks were obtained. Notice that this relatively complex procedure can be
 417 avoided if contiguous allocation is requested (by opening the typed memory
 418 object with the `tflag` `POSIX_TYPED_MEM_ALLOCATE_CONTIG`).
- 419 — *Sharing data across processes.* Process A passes the two offset values and
 420 lengths obtained from the `posix_mem_offset()` calls to process B running on
 421 P_2 , via some form of interprocess communication. Process B can gain
 422 access to process A 's data by calling `posix_typed_mem_open()` with the name
 423 `/typed.m2b-b2` and a `tflag` of zero, then using two `mmap()` calls on the
 424 resulting file descriptor to map the two subareas of that typed memory
 425 object to its own address space.

426 **B.12.4.5 Rationale for `posix_typed_mem_get_info()`**

427 An application that needs to allocate a block of typed memory with length depen-
 428 dent upon the amount of memory currently available must either query the typed
 429 memory object to obtain the amount available, or repeatedly invoke `mmap()`
 430 attempting to guess an appropriate length. While the latter method is existing
 431 practice with `malloc()`, it is awkward and imprecise. The
 432 `posix_typed_mem_get_info()` function allows an application to immediately deter-
 433 mine available memory. This is particularly important for typed memory objects
 434 that may in some cases be scarce resources. Note that when a typed memory pool

435 is a shared resource, some form of mutual exclusion or synchronization may be
 436 required while typed memory is being queried and allocated to prevent race condi-
 437 tions.

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

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

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

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

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

- 473 — The function *posix_typed_mem_open()* should provide a flag which specifies
 474 “allocate on *mmap()*” (otherwise, *mmap()* just maps the underlying object).
 475 This allows things roughly similar to */dev/zero* vs. */dev/swap*. We have
 476 implemented two such flags, one of which forces contiguous allocation.
- 477 — The function *posix_mem_offset()* is acceptable because it can be applied use-
 478 fully to mapped objects in general. It should return the file descriptor of

- 479 the underlying object.
- 480 — The function named *mem_get_info()* in an earlier draft should be renamed
481 *posix_typed_mem_get_info()* because it is not generally applicable to
482 memory objects. It should probably return the file descriptor's allocation
483 attribute. We have implemented the renaming of the function, but reject
484 having it return a piece of information which is readily known by an appli-
485 cation without this function. Its whole purpose is to query the typed
486 memory object for attributes that are not user specified, but determined by
487 the implementation.
- 488 — There should be no separate *mem_alloc()* or *mem_free()* functions. Instead,
489 using *mmap()* on a typed memory object opened with an “allocate on
490 *mmap()*” flag should be used to force allocation. These are precisely the
491 semantics defined in the current draft.

492 **B.12.4.7 Rationale for no Typed Memory Access Management**

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

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

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

513 **B.14 Clocks and Timers**

514

8

515 ⇒ **B.14 Clocks and Timers** *Add the following subclause after the unnumbered*
 516 *subclause “clocks”:*

517 ***Rationale for the Monotonic Clock***

518 For those applications that use time services to achieve realtime behavior,
 519 changing the value of the clock on which these services rely may cause errone-
 520 ous timing behavior. For these applications, it is necessary to have a monotonic 8
 521 clock which cannot run backwards, and which has a maximum clock jump that 8
 522 is required to be documented by the implementation. Additionally, it is desir- 8
 523 able (but not required by this standard) that the monotonic clock increases its 8
 524 value uniformly. This clock should not be affected by changes to the system 8
 525 time, for example to synchronize the clock with an external source or to
 526 account for leap seconds. Such changes would cause errors in the measure-
 527 ment of time intervals for those time services that use the absolute value of the
 528 clock.

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

545 The solution is to create a new monotonic clock, whose value does not change
 546 except for the regular ticking of the clock, and use this clock for implementing 8
 547 the various relative timeouts that appear in the different POSIX interfaces, as 8
 548 well as allow *pthread_cond_timedwait()* to choose this new clock for its 8
 549 timeout. A new *clock_nanosleep()* function is created to allow an application to
 550 take advantage of this newly defined clock. Notice that the monotonic clock
 551 may be implemented using the same hardware clock as the system clock.

552 Relative timeouts for *sigtimedwait()* and *aio_suspend()* have been redefined 8
 553 to use the monotonic clock, if present. The *alarm()* function has not been
 554 redefined, because the same effect but with better resolution can be achieved
 555 by creating a timer (for which the appropriate clock may be chosen).

556 The *pthread_cond_timedwait()* function has been treated in a different way, 8
 557 compared to other functions with absolute timeouts, because it is used to wait 8
 558 for an event, and thus it may have a deadline, while the other timeouts are 8
 559 generally used as an error recovery mechanism, and for them the use of the 8
 560 monotonic clock is not so important. Since the desired timeout for the 8
 561 *pthread_cond_timedwait()* function may either be a relative interval, or an 8
 562 absolute time of day deadline, a new initialization attribute has been created 8
 563 for condition variables, to specify the clock that shall be used for measuring the 8
 564 timeout in a call to *pthread_cond_timedwait()*. In this way, if a relative 8
 565 timeout is desired, the monotonic clock will be used; if an absolute deadline is 8
 566 required instead, the `CLOCK_REALTIME` or another appropriate clock may be 8
 567 used. This capability has not been added to other functions with absolute 8
 568 timeouts because for those functions the expected use of the timeout is mostly 8
 569 to prevent errors, and not so often to meet precise deadlines. As a consequence, 8
 570 the complexity of adding this capability is not justified by its perceived applica- 8
 571 tion usage. 8

572 The *nanosleep()* function has not been modified with the introduction of the 8
 573 monotonic clock. Instead, a new *clock_nanosleep()* function has been created, 8
 574 in which the desired clock may be specified in the function call.

575 ***History of Resolution Issues*** 8

576 Due to the shift from relative to absolute timeouts in IEEE 1003.1d, the amend- 8
 577 ments to the *sem_timedwait()*, *pthread_mutex_timedlock()*, *mq_timedreceive()*, 8
 578 and *mq_timedsend()* functions of that standard have been removed. Those 8
 579 amendments specified that `CLOCK_MONOTONIC` would be used for the (rela- 8
 580 tive) timeouts if the Monotonic Clock option was supported. 8

581 Having these functions continue to be tied solely to `CLOCK_MONOTONIC` 8
 582 would not work. Since the absolute value of a time value obtained from 8
 583 `CLOCK_MONOTONIC` is unspecified, under the absolute timeouts interface, 8
 584 applications would behave differently depending on whether the Monotonic 8
 585 Clock option was supported or not (because the absolute value of the clock 8
 586 would have different meanings in either case). 8

587 Two options were considered: 1) leave the current behavior unchanged, which 8
 588 specifies the `CLOCK_REALTIME` clock for these (absolute) timeouts, to allow 8
 589 portability of applications between implementations supporting or not the 8
 590 Monotonic Clock option, or 2) modify these functions in the way that 8
 591 *pthread_cond_timedwait()* was modified to allow a choice of clock, so that an 8
 592 application could use `CLOCK_REALTIME` when it is trying to achieve an abso- 8
 593 lute timeout and `CLOCK_MONOTONIC` when it is trying to achieve a relative 8
 594 timeout. 8

595 It was decided that the features of `CLOCK_MONOTONIC` are not as critical to 8
 596 these functions as they are to *pthread_cond_timedwait()*. When 8
 597 *pthread_cond_timedwait()* is given a relative timeout, the timeout may 8
 598 represent a deadline for an event. When these functions are given relative 8
 599 timeouts, the timeouts are typically for error recovery purposes and need not 8
 600 be so precise. 8

601 Therefore, it was decided that these functions should be tied to 8
602 CLOCK_REALTIME and not complicated by being given a choice of clock. 8

603 **B.14.2 Clock and Timer Functions**

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

605 **B.14.2.6 High Resolution Sleep with Specifiable Clock**

606 ***Rationale for clock_nanosleep()***

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

612 ***Rationale for absolute clock_nanosleep()***

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

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

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

641 Although the interface of the relative and absolute versions of the new high reso-
 642 lution sleep service is the same `clock_nanosleep()` function, the `rmtp` argument is
 643 only used in the relative sleep. This argument is needed in the relative
 644 `clock_nanosleep()` function to re-issue the function call if it is interrupted by a sig-
 645 nal, but it is not needed in the absolute `clock_nanosleep()` function call; if the call
 646 is interrupted by a signal, the absolute `clock_nanosleep()` function can be invoked
 647 again with the same `rtp` argument used in the interrupted call.

648 **B.18 Thread Cancellation**

649 **B.18.1 Thread Cancellation Overview**

650 **B.18.1.2 Cancellation Points**

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

654 Several important blocking routines are not cancellation points.

655 (1) `pthread_mutex_lock()`

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

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

671 (2) `barrier_wait()`

672 Canceling a barrier wait will render a barrier unusable. Similar to a bar-
 673 rier timeout (which the Working Group rejected), there is no way to
 674 guarantee the consistency of a barrier’s internal data structures if a bar-
 675 rier wait is canceled.

676 (3) `pthread_spin_lock()`

677
678
679

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

1 **F.3 Profiling Considerations**

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

3 `{_POSIX_BARRIERS}`

4 The system supports barrier synchronization.

5 This option was created to allow efficient synchronization of
6 multiple parallel threads in multiprocessor systems in which
7 the operation is supported in part by the hardware architec-
8 ture.

9 `{_POSIX_CLOCK_SELECTION}` 9

10 The system supports the Clock Selection option. 9

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

16 `{_POSIX_MONOTONIC_CLOCK}`

17 The system supports the Monotonic Clock option.

18 This option allows realtime applications to rely on a monoton-
19 ically increasing clock that does not jump backwards, and
20 whose value does not change except for the regular ticking of
21 the clock.

22 `{_POSIX_READER_WRITER_LOCKS}`

23 The system supports reader/writer locks.

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

26 simultaneous reads are allowed to a shared resource.

27 `{_POSIX_SPIN_LOCKS}`

28 The system supports spin locks.

29 This option was created to support a simple and efficient syn-
30 chronization mechanism for threads executing in multipro-
31 cessor systems.

32 `{_POSIX_TYPED_MEMORY_OBJECTS}`

33 The system supports typed memory objects.

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

Identifier Index

<i>clock_nanosleep()</i>	High Resolution Sleep with Specifiable Clock {14.2.6}	53
<i>posix_mem_offset()</i>	Find Offset and Length of a Mapped Typed Memory Block {12.4.3}	47
<i>posix_typed_mem_get_info()</i>	Query Typed Memory Information {12.4.4}	48
<i>posix_typed_mem_open()</i>	Open a Typed Memory Object {12.4.2}	44
<i>pthread_barrierattr_destroy()</i>	Barrier Initialization Attributes {11.5.1}	21
<i>pthread_barrierattr_getpshared()</i>	Barrier Initialization Attributes {11.5.1}	21
<i>pthread_barrierattr_init()</i>	Barrier Initialization Attributes {11.5.1}	21
<i>pthread_barrierattr_setpshared()</i>	Barrier Initialization Attributes {11.5.1}	21
<i>pthread_barrier_destroy()</i>	Initialize/Destroy a Barrier {11.5.2}	23
<i>pthread_barrier_init()</i>	Initialize/Destroy a Barrier {11.5.2}	23
<i>pthread_barrier_wait()</i>	Synchronize at a Barrier {11.5.3}	24
<i>pthread_condattr_getclock()</i>	Condition Variable Initialization Attributes {11.4.1}	19
<i>pthread_condattr_setclock()</i>	Condition Variable Initialization Attributes {11.4.1}	19
<i>pthread_rwlockattr_destroy()</i>	Reader/Writer Lock Initialization Attributes {11.6.1}	26
<i>pthread_rwlockattr_getpshared()</i>	Reader/Writer Lock Initialization Attributes {11.6.1}	26
<i>pthread_rwlockattr_init()</i>	Reader/Writer Lock Initialization Attributes {11.6.1}	26
<i>pthread_rwlockattr_setpshared()</i>	Reader/Writer Lock Initialization Attributes {11.6.1}	26
<i>pthread_rwlock_destroy()</i>	Initialize/Destroy a Reader/Writer Lock {11.6.2}	28
<i>pthread_rwlock_init()</i>	Initialize/Destroy a Reader/Writer Lock {11.6.2}	28
<i>pthread_rwlock_rdlock()</i>	Apply a Read Lock {11.6.3}	30
<i>pthread_rwlock_timedrdlock()</i>	Apply a Read Lock {11.6.3}	30

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

<i>pthread_rwlock_timedwrlock()</i>	
Apply a Write Lock {11.6.4}	33
<i>pthread_rwlock_tryrdlock()</i>	
Apply a Read Lock {11.6.3}	30
<i>pthread_rwlock_trywrlock()</i>	
Apply a Write Lock {11.6.4}	33
<i>pthread_rwlock_unlock()</i>	
Unlock a Reader/Writer Lock {11.6.5}	35
<i>pthread_rwlock_wrlock()</i>	
Apply a Write Lock {11.6.4}	33
<i>pthread_spin_destroy()</i>	
Initialize/Destroy a Spin Lock {11.7.1}	36
<i>pthread_spin_init()</i>	
Initialize/Destroy a Spin Lock {11.7.1}	36
<i>pthread_spin_lock()</i>	
Lock a Spin Lock {11.7.2}	38
<i>pthread_spin_trylock()</i>	
Lock a Spin Lock {11.7.2}	38
<i>pthread_spin_unlock()</i>	
Unlock a Spin Lock {11.7.3}	39
<i>S_TYPEISTMO</i>	
File Characteristics: Header and Data Structure {5.6.1}	13

Alphabetic Topical Index

A

ai_suspend() ... 16, 76
alarm() ... 76
 ANSI ... 69
 Apply a Read Lock ... 30
 Apply a Write Lock ... 33
 appropriate privileges ... 13, 46-47, 72
 Asynchronous Input and Output ... 16
 attributes
 clock ... 19-20
 attributes
 process-shared ... 21-22, 27-28, 67

B

background ... 63-64, 66
 Background ... 63-64, 66
 barrier
 definition of ... 3
 Barrier Initialization Attributes ... 21
 Barriers ... 21, 63
 Barriers option ... 5, 21, 23, 25
barrier_wait() ... 79
 Bibliography ... 61
brk() ... 74

C

Cancellation Points ... 59, 79
 Change File Modes—Description ... 13
 Change File Modes ... 13
 C Language Input/Output Functions ... 17
clock
 attribute ... 19-20
 Clock and Timer Functions ... 52-53, 78
clock_gettime() ... 52, 78
clock_jump
 definition of ... 3
 CLOCK_MONOTONIC ... 10, 16, 51-53, 77-78
clock_nanosleep() ... 52-55, 59, 76-79
 function definition ... 53

CLOCK_REALTIME ... 31, 34, 51-52, 54, 77-78
 Clocks—Cross-References ... 53
 Clocks—Description ... 52
 Clocks—Errors ... 53
 Clocks ... 52
 Clocks and Timers ... 51, 76
 Clock Selection option ... 19-20, 52-53
clock_settime() ... 3, 51-53, 55
close() ... 15, 46-47
 Close a File—Description ... 15
 Close a File ... 15
 Comparison to ISO/IEC 9945-1 *fcntl()* locks ... 65
 Condition Variable Initialization Attributes—Cross-References ... 20
 Condition Variable Initialization Attributes—Description ... 19-20
 Condition Variable Initialization Attributes—Errors ... 20
 Condition Variable Initialization Attributes—Returns ... 20
 Condition Variable Initialization Attributes—Synopsis ... 19
 Condition Variable Initialization Attributes ... 19
 Condition Variables ... 19
 Configurable System Variables ... 11
 Configuration Options ... 81
 conformance ... 1
 Conformance ... 1
 Conforming Implementation Options ... 1
 Control Operations on Files ... 16
 Create a Per-Process Timer—Description ... 53
 Create a Per-Process Timer ... 53
 Cross-References ... 23-24, 26, 28, 30, 32, 35-36, 38-40, 47-49, 55
 C Standard ... 51, 69

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

D

Data Definitions ... 44
 Data Definitions for Clocks and Timers ... 51
 Definitions ... 3
 /dev/swap ... 74
 /dev/zero ... 74
 DMA ... 75
 document ... 51, 76
 DO/FOR ... 63
 DRAM ... 68
 dup() ... 46-47
 dup2() ... 46

E

[EACCESS] ... 46, 48
 [EAGAIN] ... 24, 29, 32, 37
 [EBADF] ... 49
 [EBUSY] ... 24, 29-30, 32, 34, 38-39
 [EDEADLK] ... 32, 35, 39
 EEPROM ... 68
 [EINTR] ... 47, 55
 [EINVAL] ... 20, 22, 24, 26, 28, 30, 32, 34, 36, 38-40, 47, 53, 55
 [EMFILE] ... 47
 [ENAMETOOLONG] ... 47
 [ENFILE] ... 47
 [ENODEV] ... 49
 [ENOENT] ... 47
 [ENOMEM] ... 22, 24, 28-29, 38, 43
 [ENOTSUP] ... 55
 [ENXIO] ... 43
 [EPERM] ... 36, 40, 47
 EPROM ... 68
 [ETIMEDOUT] ... 32, 34
 Example of a system with typed memory ... 69
 Execute a File—Description ... 9
 Execute a File ... 9
 Existing Practice ... 71

F

F.3 ... 81

fchmod() ... 13
 fcntl() ... 16, 47, 65
 <fcntl.h> ... 46-47
 FD_CLOEXEC ... 46
 fdopen() ... 17
 File Characteristics ... 13
 File Characteristics: Header and Data Structure ... 13
 File Control—Description ... 16
 File Control ... 16
 file descriptor ... 15, 17, 42, 45-49, 73, 75
 File Descriptor Deassignment ... 15
 Files and Directories ... 13
 file system ... 41, 45, 64
 Find Offset and Length of a Mapped Typed Memory Block ... 47
 free() ... 74, 79
 fstat() ... 46, 74
 ftruncate() ... 46
 functions
 clock_nanosleep() ... 53
 posix_mem_offset() ... 47
 posix_typed_mem_get_info() ... 48
 posix_typed_mem_open() ... 44
 pthread_barrierattr_destroy() ... 21
 pthread_barrierattr_getpshared() ... 21
 pthread_barrierattr_init() ... 21
 pthread_barrierattr_setpshared() ... 21
 pthread_barrier_destroy() ... 23
 pthread_barrier_init() ... 23
 pthread_barrier_wait() ... 24
 pthread_condattr_getclock() ... 19
 pthread_condattr_setclock() ... 19
 pthread_rwlockattr_destroy() ... 26
 pthread_rwlockattr_getpshared() ... 26
 pthread_rwlockattr_init() ... 26
 pthread_rwlockattr_setpshared() ... 26
 pthread_rwlock_destroy() ... 28
 pthread_rwlock_init() ... 28
 pthread_rwlock_rdlock() ... 30
 pthread_rwlock_timedrdlock() ... 30
 pthread_rwlock_timedwrlock() ... 33
 pthread_rwlock_tryrdlock() ... 30
 pthread_rwlock_trywrlock() ... 33
 pthread_rwlock_unlock() ... 35
 pthread_rwlock_wrlock() ... 33
 pthread_spin_destroy() ... 36
 pthread_spin_init() ... 36
 pthread_spin_lock() ... 38
 pthread_spin_trylock() ... 38
 pthread_spin_unlock() ... 39

G

General ... 1
 General Terms ... 3
 Get Configurable System Variables—Description ... 11
 Get Configurable System Variables ... 11
 Get File Status—Description ... 13
 Get File Status ... 13

H

High Resolution Sleep with Specifiable Clock ... 53, 78
 History of Resolution Issues ... 66
 HP-RT ... 71

I

IEEE 1003.13 ... 65
 IEEE 1003.1 ... 75
 IEEE 1003.1d ... 75, 77
 IEEE P1003.1c ... 63
 Implementation Conformance ... 1
 implementation defined ... 22, 27, 30-31, 35, 41, 45-46, 48-49, 51
 Initialize/Destroy a Barrier ... 23
 Initialize/Destroy a Reader/Writer Lock ... 28
 Initialize/Destroy a Spin Lock ... 36
 Input and Output ... 15
 Input and Output Primitives ... 15
 ISBN ... 61
 ISO/IEC 9899 ... 51, 69
 ISO/IEC 9945-1 ... 41, 65

L

Lack of Timeout Feature ... 64, 66
 Language-Specific Services for the C Language ... 17
 Lock a Spin Lock ... 38
lseek() ... 16

M

malloc() ... 69-70, 73-74, 79
 Manifest Constants ... 51
 MAP_PRIVATE ... 42
 Map Process Addresses to a Memory Object—Cross-References ... 43
 Map Process Addresses to a Memory Object—Description ... 42
 Map Process Addresses to a Memory Object—Errors ... 43
 Map Process Addresses to a Memory Object ... 42
 MAP_SHARED ... 42
 MAX_SPIN ... 67
mbralloc() ... 75
mem_alloc() ... 74-75
mem_free() ... 74-75
mem_get_info() ... 75
 Memory Management ... 41, 44, 68
 Memory Mapped Files option ... 41
 Memory Mapping Functions ... 41
 memory object
 definition of ... 3
 Memory Object synchronization—Description ... 44
 Memory Object synchronization ... 44
 Message Passing ... 57
mmap() ... 3, 42, 45-49, 69, 71, 73-75
 Model ... 68
 monotonic clock
 definition of ... 3
 Monotonic Clock option ... 10, 16, 51-52, 77
mq_timedreceive() ... 53, 77
mq_timedsend() ... 53, 77
msync() ... 44, 46
munmap() ... 9, 69, 72, 74

N

NAME_MAX ... 47
nanosleep() ... 53-55, 77-78
 NULL ... 23, 29

O

open() ... 72

- Open a Stream on a File Descriptor—
Description ... 17
- Open a Stream on a File Descriptor ... 17
- Open a Typed Memory Object ... 44
- Optional Configurable System Variables ... 11
- options
 - Barriers ... 5, 21, 23, 25
 - Clock Selection ... 19-20, 52-53
 - Memory Mapped Files ... 41
 - Monotonic Clock ... 10, 16, 51-52, 77
 - Reader Writer Locks ... 5, 27-28, 30-31, 33, 35
 - Shared Memory Objects ... 41
 - Spin Locks ... 5, 36, 38-39
 - Thread Execution Scheduling ... 30, 35, 64-65
 - Timeouts ... 31, 33
 - Timers ... 31, 34
 - Typed Memory Objects ... 9, 13, 15-17, 41, 44-45, 47, 49
- options
 - Process-Shared Synchronization ... 21, 27, 36
- O_RDONLY ... 46
- O_RDWR ... 46
- OS-9 ... 71
- Other Sources of Information ... 61
- O_WRONLY ... 46

- P**
- PATH_MAX ... 47
- physio()* ... 75
- Portability Considerations ... 81
- POSIX.13 ... 68
- _POSIX_BARRIERS ... 1, 11, 21, 23, 25, 81
- _POSIX_CLOCKRES_MIN ... 51
- _POSIX_CLOCK_SELECTION ... 1, 11, 19-20, 52-53, 81
- _POSIX_MAPPED_FILES ... 42-43
- posix_mem_offset()* ... 46-48, 69-70, 73-74
function definition ... 47
- _POSIX_MONOTONIC_CLOCK ... 1, 11, 16, 52, 81
- _POSIX_NO_TRUNC ... 47
- _POSIX_READER_WRITER_LOCKS ... 1, 11, 27-28, 30-31, 33, 35, 81
- _POSIX_SHARED_MEMORY_OBJECTS ... 42-43
- _POSIX_SPIN_LOCKS ... 1, 11, 36, 38-39, 82
- _POSIX_THREAD_PRIORITY_SCHEDULING ... 30, 35
- _POSIX_THREAD_PROCESS_SHARED ... 21, 27, 36
- _POSIX_TIMEOUTS ... 31, 33
- _POSIX_TIMERS ... 31, 34
- POSIX_TYPED_MEM_ALLOCATE ... 42, 45, 47-49, 73
- POSIX_TYPED_MEM_ALLOCATE_CONTIG ... 42, 45, 47-49, 73
- posix_typed_mem_get_info()* ... 46, 48-49, 69, 73, 75
function definition ... 48
- POSIX_TYPED_MEM_MAP_ALLOCATABLE ... 43, 45-47
- posix_typed_mem_open()* ... 43-49, 59, 69, 73-74
function definition ... 44
- _POSIX_TYPED_MEMORY_OBJECTS ... 1, 11, 13, 42-45, 47, 49, 82
- Primitive System Data Types ... 4-5
- Process Creation and Execution ... 9
- Process Environment ... 11
- Process Primitives ... 9
- process-shared
 - attribute ... 21-22, 27-28
 - Attribute ... 67
 - attribute ... 67
- Process-Shared Synchronization option ... 21, 27, 36
- Process Termination ... 9
- Profiling Considerations ... 81
- PSOS ... 71
- pthread_barrierattr_destroy()* ... 21-22
function definition ... 21
- pthread_barrierattr_getpshared()* ... 21-22
function definition ... 21
- pthread_barrierattr_init()* ... 21-22
function definition ... 21
- pthread_barrierattr_setpshared()* ... 21-22
function definition ... 21
- pthread_barrier_destroy()* ... 23-24, 26
function definition ... 23
- pthread_barrier_init()* ... 23-26
function definition ... 23
- PTHREAD_BARRIER_SERIAL_THREAD ... 25, 63
- pthread_barrier_wait()* ... 23-26, 64
function definition ... 24

pthread_condattr_getclock() ... 19-20
 function definition ... 19
pthread_condattr_setclock() ... 19-20
 function definition ... 19
pthread_cond_timedwait() ... 19-20, 76-78,
 81
 <pthread.h> ... 22, 25
pthread_mutex_lock() ... 79
pthread_mutex_timedlock() ... 53, 77
pthread_mutex_trylock() ... 68
 PTHREAD_PROCESS_PRIVATE ... 22, 27-28,
 37
 PTHREAD_PROCESS_SHARED ... 21-22, 27-
 28, 37
pthread_rwlockattr_destroy() ... 26-28
 function definition ... 26
pthread_rwlockattr_getpshared() ... 26-28
 function definition ... 26
pthread_rwlockattr_init() ... 26-28
 function definition ... 26
pthread_rwlockattr_setpshared() ... 26-28
 function definition ... 26
pthread_rwlock_destroy() ... 28-30, 32,
 35-36
 function definition ... 28
pthread_rwlock_init() ... 28-29, 32, 35-36
 function definition ... 28
pthread_rwlock_rdlock() ... 26, 29-32, 35-36,
 59
 function definition ... 30
pthread_rwlock_timedrdlock() ... 29-32, 35-
 36, 59
 function definition ... 30
pthread_rwlock_timedwrlock() ... 29-30,
 32-36, 59
 function definition ... 33
pthread_rwlock_tryrdlock() ... 29-32, 35-36
 function definition ... 30
pthread_rwlock_trywrlock() ... 29-30, 32-36
 function definition ... 33
pthread_rwlock_unlock() ... 29-30, 32, 35-
 36, 64
 function definition ... 35
pthread_rwlock_wrlock() ... 26, 29-30, 32-
 36, 59
 function definition ... 33
pthread_spin_destroy() ... 36-40
 function definition ... 36
pthread_spin_init() ... 36-40, 67
 function definition ... 36
pthread_spin_lock() ... 37-40, 66, 79
 function definition ... 38

pthread_spin_trylock() ... 37-40, 67
 function definition ... 38
pthread_spin_unlock() ... 37-40
 function definition ... 39

Q

QNX ... 71
 Query Typed Memory Information ... 48
 Queuing of Waiting Threads ... 64

R

rand() ... 79
read() ... 15
 reader/writer lock
 definition of ... 3
 Reader/Writer Lock Initialization Attributes
 ... 26
 Reader/Writer Locks ... 26, 64
 Reader Writer Locks option ... 5, 27-28, 30-
 31, 33, 35
 Read from a File—description ... 15
 Read from a File ... 15
 Reposition Read/Write File Offset—Description
 ... 16
 Reposition Read/Write File Offset ... 16
 Requirements ... 71
 ROM ... 68

S

sbrk() ... 74
 _SC_BARRIERS ... 11
 limit definition ... 11
 _SC_CLOCK_SELECTION ... 11
 limit definition ... 11
 Scenario ... 73
 SCHED_FIFO ... 30, 35, 64
 SCHED_RR ... 30, 35, 64
 SCHED_SPORADIC ... 30, 35
 _SC_MONOTONIC_CLOCK ... 11
 limit definition ... 11
 _SC_READER_WRITER_LOCKS ... 11
 limit definition ... 11
 _SC_SPIN_LOCKS ... 11
 limit definition ... 11

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

_SC_TYPED_MEMORY_OBJECTS ... 11
 limit definition ... 11
sem_timedwait() ... 53, 77
 Shared Memory Objects option ... 41
 Signals ... 9
sigtimedwait() ... 76
sleep() ... 55, 78
spin_init() ... 67
 spin lock
 definition of ... 4
 Spin Locks ... 36, 66
 Spin Locks option ... 5, 36, 38-39
 Spin Locks vs. Mutexes ... 67
 SRAM ... 68
 S_TYPEISSHM ... 13
S_TYPEISTMO
 definition of ... 13
 S_TYPEISTMO ... 13
 SVID ... 71
 Synchronization ... 19, 21, 63
 Synchronize at a Barrier ... 24
 Synchronously Accept a Signal—description
 ... 10
 Synchronously Accept a Signal ... 9
 <sys/mman.h> ... 44-45, 47-49
 System V ... 11, 71
 <sys/types.h> ... 4

T

Terminate a Process—Description ... 9
 Terminate a Process ... 9
 Terminology and General Requirements ... 3
 terms ... 3
 Thread Cancellation ... 59, 79
 Thread Cancellation Overview ... 59, 79
 Thread Execution Scheduling option ... 30,
 35, 64-65
time() ... 31, 34
 <time.h> ... 31, 34, 51
 Timeouts option ... 31, 33
 TIMER_ABSTIME ... 53-55
timer_create() ... 53
timer_settime() ... 53
 Timers option ... 31, 34
 TOC ... 1
 /typed.m2b-b1 ... 73

/typed.m2b-b2 ... 73
 Typed Memory Functions ... 44, 68
 typed memory namespace
 definition of ... 4
 typed memory object
 definition of ... 4
 Typed Memory Objects option ... 9, 13, 15-
 17, 41, 44-45, 47, 49
 typed memory pool
 definition of ... 4
 typed memory port
 definition of ... 4

U

UDI ... 75
umask() ... 47
 undefined ... 21-23, 25, 27, 29, 31, 33-35,
 37-39, 42, 49
 UNIX ... 71
 Unlock a Reader/Writer Lock ... 35
 Unlock a Spin Lock ... 39
 Unmap Previously Mapped Addresses—
 Cross-References ... 44
 Unmap Previously Mapped Addresses—
 Description ... 43
 Unmap Previously Mapped Addresses ... 43
 unspecified ... 13, 15-17, 25, 39, 44-46, 49,
 52, 54, 74, 77

V

VME ... 71

W

Wait for an Asynchronous I/O Request—
 Description ... 16
 Wait for an Asynchronous I/O Request ... 16
 Waiting on a Condition—Description ... 20
 Waiting on a Condition ... 20
write() ... 16
 Write to a File—Description ... 16
 Write to a File ... 15

