

1 **TODO**

2 Check for overlaps with Mantis bugs: 374 and 1218 (once resolved; NB 374 may also affect
3 aligned_alloc()), and any that get tagged tc3 or issue8 after 2019-11-18

4 **Introduction**

5 This document details the changes needed to align POSIX.1/SUS with ISO C 9899:2018 (C17) in
6 Issue 8. It covers technical changes only; it does not cover simple editorial changes that the editor
7 can be expected to handle as a matter of course (such as updating normative references). It is
8 entirely possible that C2x will be approved before Issue 8, in which case a further set of changes to
9 align with C2x will need to be identified during work on the Issue 8 drafts.

10 Note that the removal of *gets()* is not included here, as it is already shaded OB and so will
11 automatically be removed by default in Issue 8.

12 All page and line numbers refer to the SUSv4 2018 edition (C181.pdf).

13 **Global Change**

14 Change all occurrences of “c99” to “c17”, except in CHANGE HISTORY sections and on XRAT
15 page 3556 line 120684 section A.12.2 Utility Syntax Guidelines.

16 *Note to the editors: use a troff string for c17, e.g. *(cy or *(cY, so that it can be easily changed
17 again if necessary.*

18 **Changes to XBD**

19 Ref G.1 para 1

20 On page 9 line 249 section 1.7.1 Codes, add a new code:

21 [MXC]IEC 60559 Complex Floating-Point[/MXC]

22 The functionality described is optional. The functionality described is mandated by the ISO
23 C standard only for implementations that define `__STDC_IEC_559_COMPLEX__`.

24 Ref (none)

25 On page 29 line 1063, 1067 section 2.2.1 Strictly Conforming POSIX Application, change:

26 the ISO/IEC 9899: 1999 standard

27 to:

28 the ISO C standard

29 Ref 6.2.8

30 On page 34 line 1184 section 3.11 Alignment, change:

31 See also the ISO C standard, Section B3.

32 to:

33 See also the ISO C standard, Section 6.2.8.

34 Ref 5.1.2.4

35 On page 38 line 1261 section 3 Definitions, add a new subsection:

36 **3.31 Atomic Operation**

37 An operation that cannot be broken up into smaller parts that could be performed separately.
38 An atomic operation is guaranteed to complete either fully or not at all. In the context of the
39 functionality provided by the `<stdatomic.h>` header, there are different types of atomic
40 operation that are defined in detail in [xref to XSH 4.12.1].

41 Ref 7.26.3

42 On page 50 line 1581 section 3.107 Condition Variable, add a new paragraph:

43 There are two types of condition variable: those of type **pthread_cond_t** which are
44 initialized using `pthread_cond_init()` and those of type **cond_t** which are initialized using
45 `cond_init()`. If an application attempts to use the two types interchangeably (that is, pass a
46 condition variable of type **pthread_cond_t** to a function that takes a **cond_t**, or vice versa),
47 the behavior is undefined.

48 **Note:** The `pthread_cond_init()` and `cond_init()` functions are defined in detail in the System
49 Interfaces volume of POSIX.1-20xx.

50 Ref 5.1.2.4

51 On page 53 line 1635 section 3 Definitions, add a new subsection:

52 **3.125 Data Race**

53 A situation in which there are two conflicting actions in different threads, at least one of
54 which is not atomic, and neither “happens before” the other, where the “happens before”
55 relation is defined formally in [xref to XSH 4.12.1.1].

56 Ref 5.1.2.4

57 On page 67 line 1973 section 3 Definitions, add a new subsection:

58 **3.215 Lock-Free Operation**

59 An operation that does not require the use of a lock such as a mutex in order to avoid data
60 races.

61 Ref 7.26.5.1

62 On page 70 line 2048 section 3.233 Multi-Threaded Program, change:

63 the process can create additional threads using `pthread_create()` or `SIGEV_THREAD`
64 notifications.

65 to:

66 the process can create additional threads using `pthread_create()`, `thrd_create()`, or
67 `SIGEV_THREAD` notifications.

68 Ref 7.26.4

69 On page 70 line 2054 section 3.234 Mutex, add a new paragraph:

70 There are two types of mutex: those of type **pthread_mutex_t** which are initialized using
71 *pthread_mutex_init()* and those of type **mtx_t** which are initialized using *mtx_init()*. If an
72 application attempts to use the two types interchangeably (that is, pass a mutex of type
73 **pthread_mutex_t** to a function that takes a **mtx_t**, or vice versa), the behavior is undefined.

74 **Note:** The *pthread_mutex_init()* and *mtx_init()* functions are defined in detail in the System
75 Interfaces volume of POSIX.1-20xx.

76 Ref 7.26.5.5

77 On page 82 line 2345 section 3.303 Process Termination, change:

78 or when the last thread in the process terminates by returning from its start function, by
79 calling the *pthread_exit()* function, or through cancellation.

80 to:

81 or when the last thread in the process terminates by returning from its start function, by
82 calling the *pthread_exit()* or *thrd_exit()* function, or through cancellation.

83 Ref 7.26.5.1

84 On page 90 line 2530 section 3.354 Single-Threaded Program, change:

85 if the process attempts to create additional threads using *pthread_create()* or
86 SIGEV_THREAD notifications

87 to:

88 if the process attempts to create additional threads using *pthread_create()*, *thrd_create()*, or
89 SIGEV_THREAD notifications

90 Ref 5.1.2.4

91 On page 95 line 2639 section 3 Definition, add a new subsection:

92 **3.382 Synchronization Operation**

93 An operation that synchronizes memory. See [xref to XSH 4.12].

94 Ref 7.26.5.1

95 On page 99 line 2745 section 3.405 Thread ID, change:

96 Each thread in a process is uniquely identified during its lifetime by a value of type
97 **pthread_t** called a thread ID.

98 to:

99 A value that uniquely identifies each thread in a process during the thread's lifetime. The
100 value shall be unique across all threads in a process, regardless of whether the thread is:

- 101 • The initial thread.
102 • A thread created using *pthread_create()*.
103 • A thread created using *thrd_create()*.
104 • A thread created via a SIGEV_THREAD notification.

105 **Note:** Since *pthread_create()* returns an ID of type **pthread_t** and *thrd_create()* returns an ID of
106 type **thrd_t**, this uniqueness requirement necessitates that these two types are defined as the
107 same underlying type because calls to *pthread_self()* and *thrd_current()* from the initial
108 thread need to return the same thread ID. The *pthread_create()*, *pthread_self()*, *thrd_create()*
109 and *thrd_current()* functions and SIGEV_THREAD notifications are defined in detail in the
110 System Interfaces volume of POSIX.1-20xx.

111 Ref 5.1.2.4

112 On page 99 line 2752 section 3.407 Thread-Safe, change:

113 A thread-safe function can be safely invoked concurrently with other calls to the same
114 function, or with calls to any other thread-safe functions, by multiple threads.

115 to:

116 A thread-safe function shall avoid data races with other calls to the same function, and with
117 calls to any other thread-safe functions, by multiple threads.

118 Ref 5.1.2.4

119 On page 99 line 2756 section 3.407 Thread-Safe, add a new paragraph:

120 A function that is not required to be thread-safe need not avoid data races with other calls to
121 the same function, nor with calls to any other function (including thread-safe functions), by
122 multiple threads, unless explicitly stated otherwise.

123 Ref 7.26.6

124 On page 99 line 2758 section 3.408 Thread-Specific Data Key, change:

125 A process global handle of type **pthread_key_t** which is used for naming thread-specific
126 data.

127 Although the same key value may be used by different threads, the values bound to the key
128 by *pthread_setspecific()* and accessed by *pthread_getspecific()* are maintained on a per-
129 thread basis and persist for the life of the calling thread.

130 **Note:** The *pthread_getspecific()* and *pthread_setspecific()* functions are defined in detail in the
131 System Interfaces volume of POSIX.1-2017.

132 to:

133 A process global handle which is used for naming thread-specific data. There are two types
134 of key: those of type **pthread_key_t** which are created using *pthread_key_create()* and
135 those of type **tss_t** which are created using *tss_create()*. If an application attempts to use the
136 two types of key interchangeably (that is, pass a key of type **pthread_key_t** to a function
137 that takes a **tss_t**, or vice versa), the behavior is undefined.

138 Although the same key value can be used by different threads, the values bound to the key
139 by *pthread_setspecific()* for keys of type **pthread_key_t**, and by *tss_set()* for keys of type

140 **tss_t**, are maintained on a per-thread basis and persist for the life of the calling thread.

141 **Note:** The *pthread_key_create()*, *pthread_setspecific()*, *tss_create()* and *tss_set()* functions are
142 defined in detail in the System Interfaces volume of POSIX.1-20xx.

143 Ref 5.1.2.4, 7.17.3

144 On page 111 line 3060 section 4.12 Memory Synchronization, change:

145 **4.12 Memory Synchronization**

146 Applications shall ensure that access to any memory location by more than one thread of
147 control (threads or processes) is restricted such that no thread of control can read or modify
148 a memory location while another thread of control may be modifying it. Such access is
149 restricted using functions that synchronize thread execution and also synchronize memory
150 with respect to other threads. The following functions synchronize memory with respect to
151 other threads:

152 to:

153 **4.12 Memory Ordering and Synchronization**

154 **4.12.1 Memory Ordering**

155 *4.12.1.1 Data Races*

156 The value of an object visible to a thread *T* at a particular point is the initial value of the
157 object, a value stored in the object by *T*, or a value stored in the object by another thread,
158 according to the rules below.

159 Two expression evaluations *conflict* if one of them modifies a memory location and the other
160 one reads or modifies the same memory location.

161 This standard defines a number of atomic operations (see <**stdatomic.h**>) and operations on
162 mutexes (see <**threads.h**>) that are specially identified as synchronization operations. These
163 operations play a special role in making assignments in one thread visible to another. A
164 synchronization operation on one or more memory locations is either an *acquire operation*, a
165 *release operation*, both an acquire and release operation, or a *consume operation*. A
166 synchronization operation without an associated memory location is a *fence* and
167 can be either an acquire fence, a release fence, or both an acquire and release fence. In
168 addition, there are *relaxed atomic operations*, which are not synchronization operations, and
169 atomic *read-modify-write operations*, which have special characteristics.

170 **Note:** For example, a call that acquires a mutex will perform an acquire operation on the locations
171 composing the mutex. Correspondingly, a call that releases the same mutex will perform a
172 release operation on those same locations. Informally, performing a release operation on *A*
173 forces prior side effects on other memory locations to become visible to other threads that
174 later perform an acquire or consume operation on *A*. Relaxed atomic operations are not
175 included as synchronization operations although, like synchronization operations, they
176 cannot contribute to data races.

177 All modifications to a particular atomic object *M* occur in some particular total order, called
178 the *modification order* of *M*. If *A* and *B* are modifications of an atomic object *M*, and *A*
179 happens before *B*, then *A* shall precede *B* in the modification order of *M*, which is defined
180 below.

181 **Note:** This states that the modification orders must respect the “happens before” relation.

182 **Note:** There is a separate order for each atomic object. There is no requirement that these can be
183 combined into a single total order for all objects. In general this will be impossible since
184 different threads may observe modifications to different variables in inconsistent orders.

185 *A release sequence* headed by a release operation *A* on an atomic object *M* is a maximal
186 contiguous sub-sequence of side effects in the modification order of *M*, where the first
187 operation is *A* and every subsequent operation either is performed by the same thread that
188 performed the release or is an atomic read-modify-write operation.

189 Certain system interfaces *synchronize with* other system interfaces performed by another
190 thread. In particular, an atomic operation *A* that performs a release operation on an object *M*
191 shall synchronize with an atomic operation *B* that performs an acquire operation on *M* and
192 reads a value written by any side effect in the release sequence headed by *A*.

193 **Note:** Except in the specified cases, reading a later value does not necessarily ensure visibility as
194 described below. Such a requirement would sometimes interfere with efficient
195 implementation.

196 **Note:** The specifications of the synchronization operations define when one reads the value written
197 by another. For atomic variables, the definition is clear. All operations on a given mutex
198 occur in a single total order. Each mutex acquisition “reads the value written” by the last
199 mutex release.

200 An evaluation *A* carries a dependency to an evaluation *B* if:

- 201 • the value of *A* is used as an operand of *B*, unless:
 - 202 — *B* is an invocation of the *kill_dependency()* macro,
 - 203 — *A* is the left operand of a *&&* or *||* operator,
 - 204 — *A* is the left operand of a *?:* operator, or
 - 205 — *A* is the left operand of a *,* (comma) operator; or
- 206 • *A* writes a scalar object or bit-field *M*, *B* reads from *M* the value written by *A*, and *A*
207 is sequenced before *B*, or
- 208 • for some evaluation *X*, *A* carries a dependency to *X* and *X* carries a dependency to *B*.

209 An evaluation *A* is *dependency-ordered before* an evaluation *B* if:

- 210 • *A* performs a release operation on an atomic object *M*, and, in another thread, *B*
211 performs a consume operation on *M* and reads a value written by any side effect in
212 the release sequence headed by *A*, or
- 213 • for some evaluation *X*, *A* is dependency-ordered before *X* and *X* carries a dependency
214 to *B*.

215 An evaluation *A* *inter-thread happens before* an evaluation *B* if *A* synchronizes with *B*, *A* is
216 dependency-ordered before *B*, or, for some evaluation *X*:

- 217 • *A* synchronizes with *X* and *X* is sequenced before *B*,
- 218 • *A* is sequenced before *X* and *X* inter-thread happens before *B*, or
- 219 • *A* inter-thread happens before *X* and *X* inter-thread happens before *B*.

220 **Note:** The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced

221 before”, “synchronizes with”, and “dependency-ordered before” relationships, with two
222 exceptions. The first exception is that a concatenation is not permitted to end with
223 “dependency-ordered before” followed by “sequenced before”. The reason for this limitation
224 is that a consume operation participating in a “dependency-ordered before” relationship
225 provides ordering only with respect to operations to which this consume operation actually
226 carries a dependency. The reason that this limitation applies only to the end of such a
227 concatenation is that any subsequent release operation will provide the required ordering for
228 a prior consume operation. The second exception is that a concatenation is not permitted to
229 consist entirely of “sequenced before”. The reasons for this limitation are (1) to permit
230 “inter-thread happens before” to be transitively closed and (2) the “happens before” relation,
231 defined below, provides for relationships consisting entirely of “sequenced before”.

232 An evaluation *A* *happens before* an evaluation *B* if *A* is sequenced before *B* or *A* inter-thread
233 happens before *B*. The implementation shall ensure that a cycle in the “happens before”
234 relation never occurs.

235 **Note:** This cycle would otherwise be possible only through the use of consume operations.

236 A *visible side effect* *A* on an object *M* with respect to a value computation *B* of *M* satisfies
237 the conditions:

- 238 • *A* happens before *B*, and
- 239 • there is no other side effect *X* to *M* such that *A* happens before *X* and *X* happens
240 before *B*.

241 The value of a non-atomic scalar object *M*, as determined by evaluation *B*, shall be the value
242 stored by the visible side effect *A*.

243 **Note:** If there is ambiguity about which side effect to a non-atomic object is visible, then there is a
244 data race and the behavior is undefined.

245
246 **Note:** This states that operations on ordinary variables are not visibly reordered. This is not actually
247 detectable without data races, but it is necessary to ensure that data races, as defined here,
248 and with suitable restrictions on the use of atomics, correspond to data races in a simple
249 interleaved (sequentially consistent) execution.

251 The value of an atomic object *M*, as determined by evaluation *B*, shall be the value stored by
252 some side effect *A* that modifies *M*, where *B* does not happen before *A*.

253 **Note:** The set of side effects from which a given evaluation might take its value is also restricted by
254 the rest of the rules described here, and in particular, by the coherence requirements below.

255 If an operation *A* that modifies an atomic object *M* happens before an operation *B* that
256 modifies *M*, then *A* shall be earlier than *B* in the modification order of *M*. (This is known as
257 “write-write coherence”.)

258 If a value computation *A* of an atomic object *M* happens before a value computation *B* of *M*,
259 and *A* takes its value from a side effect *X* on *M*, then the value computed by *B* shall either be
260 the value stored by *X* or the value stored by a side effect *Y* on *M*, where *Y* follows *X* in the
261 modification order of *M*. (This is known as “read-read coherence”.)

262 If a value computation *A* of an atomic object *M* happens before an operation *B* on *M*, then *A*
263 shall take its value from a side effect *X* on *M*, where *X* precedes *B* in the modification order
264 of *M*. (This is known as “read-write coherence”.)

265 If a side effect X on an atomic object M happens before a value computation B of M , then the
266 evaluation B shall take its value from X or from a side effect Y that follows X in the
267 modification order of M . (This is known as “write-read coherence”.)

268 **Note:** This effectively disallows implementation reordering of atomic operations to a single object,
269 even if both operations are “relaxed” loads. By doing so, it effectively makes the “cache
270 coherence” guarantee provided by most hardware available to POSIX atomic operations.

271 **Note:** The value observed by a load of an atomic object depends on the “happens before” relation,
272 which in turn depends on the values observed by loads of atomic objects. The intended
273 reading is that there must exist an association of atomic loads with modifications they
274 observe that, together with suitably chosen modification orders and the “happens before”
275 relation derived as described above, satisfy the resulting constraints as imposed here.

276 An application contains a data race if it contains two conflicting actions in different threads,
277 at least one of which is not atomic, and neither happens before the other. Any such data
278 race results in undefined behavior.

279 4.12.1.2 Memory Order and Consistency

280 The enumerated type **memory_order**, defined in `<stdatomic.h>` (if supported), specifies
281 the detailed regular (non-atomic) memory synchronization operations as defined in [xref to
282 4.12.1.1] and may provide for operation ordering. Its enumeration constants specify memory
283 order as follows:

284 For `memory_order_relaxed`, no operation orders memory.

285 For `memory_order_release`, `memory_order_acq_rel`, and
286 `memory_order_seq_cst`, a store operation performs a release operation on the affected
287 memory location.

288 For `memory_order_acquire`, `memory_order_acq_rel`, and
289 `memory_order_seq_cst`, a load operation performs an acquire operation on the affected
290 memory location.

291 For `memory_order_consume`, a load operation performs a consume operation on the
292 affected memory location.

293 There shall be a single total order S on all `memory_order_seq_cst` operations, consistent
294 with the “happens before” order and modification orders for all affected locations, such that
295 each `memory_order_seq_cst` operation B that loads a value from an atomic object M
296 observes one of the following values:

- 297 • the result of the last modification A of M that precedes B in S , if it exists, or
- 298 • if A exists, the result of some modification of M that is not
299 `memory_order_seq_cst` and that does not happen before A , or
- 300 • if A does not exist, the result of some modification of M that is not
301 `memory_order_seq_cst`.

302 **Note:** Although it is not explicitly required that S include lock operations, it can always be
303 extended to an order that does include lock and unlock operations, since the ordering
304 between those is already included in the “happens before” ordering.

305 **Note:** Atomic operations specifying `memory_order_relaxed` are relaxed only with respect to
306 memory ordering. Implementations must still guarantee that any given atomic access to a
307 particular atomic object be indivisible with respect to all other atomic accesses to that object.

308 For an atomic operation *B* that reads the value of an atomic object *M*, if there is a
309 `memory_order_seq_cst` fence *X* sequenced before *B*, then *B* observes either the last
310 `memory_order_seq_cst` modification of *M* preceding *X* in the total order *S* or a later
311 modification of *M* in its modification order.

312 For atomic operations *A* and *B* on an atomic object *M*, where *A* modifies *M* and *B* takes its
313 value, if there is a `memory_order_seq_cst` fence *X* such that *A* is sequenced before *X* and
314 *B* follows *X* in *S*, then *B* observes either the effects of *A* or a later modification of *M* in its
315 modification order.

316 For atomic modifications *A* and *B* of an atomic object *M*, *B* occurs later than *A* in the
317 modification order of *M* if:

- 318 • there is a `memory_order_seq_cst` fence *X* such that *A* is sequenced before *X*, and
319 *X* precedes *B* in *S*, or
- 320 • there is a `memory_order_seq_cst` fence *Y* such that *Y* is sequenced before *B*, and
321 *A* precedes *Y* in *S*, or
- 322 • there are `memory_order_seq_cst` fences *X* and *Y* such that *A* is sequenced before
323 *X*, *Y* is sequenced before *B*, and *X* precedes *Y* in *S*.

324 Atomic read-modify-write operations shall always read the last value (in the modification
325 order) stored before the write associated with the read-modify-write operation.

326 An atomic store shall only store a value that has been computed from constants and input
327 values by a finite sequence of evaluations, such that each evaluation observes the values of
328 variables as computed by the last prior assignment in the sequence. The ordering of
329 evaluations in this sequence shall be such that:

- 330 • If an evaluation *B* observes a value computed by *A* in a different thread, then *B* does
331 not happen before *A*.
- 332 • If an evaluation *A* is included in the sequence, then all evaluations that assign to the
333 same variable and happen before *A* are also included.

334 **Note:** The second requirement disallows “out-of-thin-air”, or “speculative” stores of atomics when
335 relaxed atomics are used. Since unordered operations are involved, evaluations can appear in
336 this sequence out of thread order.

337 4.12.2 Memory Synchronization

338 In order to avoid data races, applications shall ensure that non-lock-free access to any
339 memory location by more than one thread of control (threads or processes) is restricted such
340 that no thread of control can read or modify a memory location while another thread of
341 control may be modifying it. Such access can be restricted using functions that synchronize
342 thread execution and also synchronize memory with respect to other threads. The following
343 functions shall synchronize memory with respect to other threads:

344 Ref 7.26.3, 7.26.4

345 On page 111 line 3066-3075 section 4.12 Memory Synchronization, add the following to the list of
346 functions that synchronize memory:

347	<i>cnd_broadcast()</i>	<i>mtx_lock()</i>	<i>thrd_create()</i>
348	<i>cnd_signal()</i>	<i>mtx_timedlock()</i>	<i>thrd_join()</i>
349	<i>cnd_timedwait()</i>	<i>mtx_trylock()</i>	
350	<i>cnd_wait()</i>	<i>mtx_unlock()</i>	

351 Ref 7.26.2.1, 7.26.4

352 On page 111 line 3076 section 4.12 Memory Synchronization, change:

353 The *pthread_once()* function shall synchronize memory for the first call in each thread for a
354 given **pthread_once_t** object. If the *init_routine* called by *pthread_once()* is a cancellation
355 point and is canceled, a call to *pthread_once()* for the same **pthread_once_t** object made
356 from a cancellation cleanup handler shall also synchronize memory.

357 The *pthread_mutex_lock()* function need not synchronize memory if the mutex type is
358 PTHREAD_MUTEX_RECURSIVE and the calling thread already owns the mutex. The
359 *pthread_mutex_unlock()* function need not synchronize memory if the mutex type is
360 PTHREAD_MUTEX_RECURSIVE and the mutex has a lock count greater than one.

361 to:

362 The *pthread_once()* and *call_once()* functions shall synchronize memory for the first call in
363 each thread for a given **pthread_once_t** or **once_flag** object, respectively. If the *init_routine*
364 called by *pthread_once()* or *call_once()* is a cancellation point and is canceled, a call to
365 *pthread_once()* for the same **pthread_once_t** object, or to *call_once()* for the same
366 **once_flag** object, made from a cancellation cleanup handler shall also synchronize memory.

367 The *pthread_mutex_lock()* and *thrd_lock()* functions, and their related “timed” and “try”
368 variants, need not synchronize memory if the mutex is a recursive mutex and the calling
369 thread already owns the mutex. The *pthread_mutex_unlock()* and *thrd_unlock()* functions
370 need not synchronize memory if the mutex is a recursive mutex and has a lock count greater
371 than one.

372 Ref 7.12.1 para 7

373 On page 117 line 3319 section 4.20 Treatment of Error Conditions for Mathematical Functions,
374 change:

375 The following error conditions are defined for all functions in the **<math.h>** header.

376 to:

377 The error conditions defined for all functions in the **<math.h>** header are domain, pole and
378 range errors, described below. If a domain, pole, or range error occurs and the integer
379 expression (*math_errhandling* & MATH_ERRNO) is zero, then *errno* shall either be set to
380 the value corresponding to the error, as specified below, or be left unmodified. If no such
381 error occurs, *errno* shall be left unmodified regardless of the setting of *math_errhandling*.

382 Ref 7.12.1 para 3

383 On page 117 line 3330 section 4.20.2 Pole Error, change:

384 A “pole error” occurs if the mathematical result of the function is an exact infinity (for
385 example, `log(0.0)`).

386 to:

387 A “pole error” shall occur if the mathematical result of the function has an exact infinite
388 result as the finite input argument(s) are approached in the limit (for example, `log(0.0)`). The
389 description of each function lists any required pole errors; an implementation may define
390 additional pole errors, provided that such errors are consistent with the mathematical
391 definition of the function.

392 Ref 7.12.1 para 4

393 On page 118 line 3339 section 4.20.3 Range Error, after:

394 A “range error” shall occur if the finite mathematical result of the function cannot be
395 represented in an object of the specified type, due to extreme magnitude.

396 add:

397 The description of each function lists any required range errors; an implementation may
398 define additional range errors, provided that such errors are consistent with the mathematical
399 definition of the function and are the result of either overflow or underflow.

400 Ref 7.29.1 para 5

401 On page 129 line 3749 section 6.3 C Language Wide-Character Codes, add a new paragraph:

402 Arguments to the functions declared in the `<wchar.h>` header can point to arrays containing
403 `wchar_t` values that do not correspond to valid wide character codes according to the
404 `LC_CTYPE` category of the locale being used. Such values shall be processed according to
405 the specified semantics for the function in the System Interfaces volume of POSIX.1-20xx,
406 except that it is unspecified whether an encoding error occurs if such a value appears in the
407 format string of a function that has a format string as a parameter and the specified
408 semantics do not require that value to be processed as if by `wcrtomb()`.

409 Ref 7.3.1 para 2

410 On page 224 line 7541 section `<complex.h>`, add a new paragraph:

411 [CX] Implementations shall not define the macro `__STDC_NO_COMPLEX__`, except for
412 profile implementations that define `_POSIX_SUBPROFILE` (see [xref to 2.1.5.1
413 Subprofiling Considerations]) in `<unistd.h>`, which may define
414 `__STDC_NO_COMPLEX__` and, if they do so, need not provide this header nor support
415 any of its facilities.[/CX]

416 Ref G.6 para 1

417 On page 224 line 7551 section `<complex.h>`, after:

418 The macros `imaginary` and `_Imaginary_I` shall be defined if and only if the implementation
419 supports imaginary types.

420 add:

421 [MXC]Implementations that support the IEC 60559 Complex Floating-Point option shall

422 define the macros `imaginary` and `_Imaginary_I`, and the macro `I` shall expand to
423 `_Imaginary_I.[MXC]`

424 Ref 7.3.9.3
425 On page 224 line 7553 section `<complex.h>`, add:

426 The following shall be defined as macros.

```
427           double complex        CMPLX(double x, double y);  
428           float complex         CMPLXF(float x, float y);  
429           long double complex CMPLXL(long double x, long double y);
```

430 Ref 7.3.1 para 2
431 On page 226 line 7623 section `<complex.h>`, add a new first paragraph to APPLICATION USAGE:

432 The `<complex.h>` header is optional in the ISO C standard but is mandated by POSIX.1-
433 20xx. Note however that subprofiles can choose to make this header optional (see [xref to
434 2.1.5.1 Subprofiling Considerations]), and therefore application portability to subprofile
435 implementations would benefit from checking whether `__STDC_NO_COMPLEX__` is
436 defined before inclusion of `<complex.h>`.

437 Ref 7.3.9.3
438 On page 226 line 7649 section `<complex.h>`, add `CMPLX()` to the SEE ALSO list before `cabs()`.

439 Ref 7.5 para 2
440 On page 234 line 7876 section `<errno.h>`, change:

441 The `<errno.h>` header shall provide a declaration or definition for `errno`. The symbol `errno`
442 shall expand to a modifiable lvalue of type `int`. It is unspecified whether `errno` is a macro or
443 an identifier declared with external linkage.

444 to:
445 The `<errno.h>` header shall provide a definition for the macro `errno`, which shall expand to
446 a modifiable lvalue of type `int` and thread local storage duration.

447 Ref (none)
448 On page 245 line 8290 section `<fenv.h>`, change:

449 the ISO/IEC 9899: 1999 standard

450 to:

451 the ISO C standard

452 Ref 5.2.4.2.2 para 11
453 On page 248 line 8369 section `<float.h>`, add the following new paragraphs:

454 The presence or absence of subnormal numbers is characterized by the implementation-
455 defined values of `FLT_HAS_SUBNORM`, `DBL_HAS_SUBNORM`, and
456 `LDBL_HAS_SUBNORM`:

 -1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

457 **Note:** Characterization as indeterminable is intended if floating-point operations do not consistently
458 interpret subnormal representations as zero, nor as non-zero. Characterization as absent is
459 intended if no floating-point operations produce subnormal results from non-subnormal
460 inputs, even if the type format includes representations of subnormal numbers.

461 Ref 5.2.4.2.2 para 12

462 On page 248 line 8378 section <float.h>, add a new bullet item:

463 Number of decimal digits, n , such that any floating-point number with p radix b digits can
464 be rounded to a floating-point number with n decimal digits and back again without change
465 to the value.

466 [math stuff]

467 FLT_DECIMAL_DIG 6

468 DBL_DECIMAL_DIG 10

469 LDBL_DECIMAL_DIG 10

470 where [math stuff] is a copy of the math stuff that follows line 8381, with the “max” suffixes
471 removed.

472 Ref 5.2.4.2.2 para 14

473 On page 250 line 8429 section <float.h>, add a new bullet item:

474 Minimum positive floating-point number.

475 FLT_TRUE_MIN 1E-37

476 DBL_TRUE_MIN 1E-37

477 LDBL_TRUE_MIN 1E-37

478 **Note:** If the presence or absence of subnormal numbers is indeterminable, then the value is
479 intended to be a positive number no greater than the minimum normalized positive number
480 for the type.

481 Ref (none)

482 On page 270 line 8981 section <limits.h>, change:

483 the ISO/IEC 9899: 1999 standard

484 to:

485 the ISO C standard

486 Ref 7.22.4.3

487 On page 271 line 9030 section <limits.h>, change:

488 Maximum number of functions that may be registered with *atexit()*.

489 to:

490 Maximum number of functions that can be registered with *atexit()* or *at_quick_exit()*. The
491 limit shall apply independently to each function.

492 Ref 5.2.4.2.1 para 2

493 On page 280 line 9419 section <limits.h>, change:

494 If the value of an object of type **char** is treated as a signed integer when used in an
495 expression, the value of {CHAR_MIN} is the same as that of {SCHAR_MIN} and the value
496 of {CHAR_MAX} is the same as that of {SCHAR_MAX}. Otherwise, the value of
497 {CHAR_MIN} is 0 and the value of {CHAR_MAX} is the same as that of
498 {UCHAR_MAX}.

499 to:

500 If an object of type **char** can hold negative values, the value of {CHAR_MIN} shall be the
501 same as that of {SCHAR_MIN} and the value of {CHAR_MAX} shall be the same as that
502 of {SCHAR_MAX}. Otherwise, the value of {CHAR_MIN} shall be 0 and the value of
503 {CHAR_MAX} shall be the same as that of {UCHAR_MAX}.

504 Ref (none)

505 On page 294 line 10016 section <math.h>, change:

506 the ISO/IEC 9899: 1999 standard provides for ...

507 to:

508 the ISO/IEC 9899: 1999 standard provided for ...

509 Ref 7.26.5.5

510 On page 317 line 10742 section <pthread.h>, change:

511 void pthread_exit(void *);

512 to:

513 _Noreturn void pthread_exit(void *);

514 Ref 7.13.2.1 para 1

515 On page 331 line 11204 section <setjmp.h>, change:

516 void longjmp(jmp_buf, int);

517 [CX] void siglongjmp(sigjmp_buf, int);[/CX]

518 to:

519 _Noreturn void longjmp(jmp_buf, int);

520 [CX] _Noreturn void siglongjmp(sigjmp_buf, int);[/CX]

521 Ref 7.15
522 On page 343 line 11647 insert a new <stdalign.h> section:

523 **NAME**

524 `stdalign.h` — alignment macros

525 **SYNOPSIS**

526 `#include <stdalign.h>`

527 **DESCRIPTION**

528 [CX] The functionality described on this reference page is aligned with the ISO C standard.
529 Any conflict between the requirements described here and the ISO C standard is
530 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

531 The <**stdalign.h**> header shall define the following macros:

532 `alignas` Expands to **`_Alignas`**

533 `alignof` Expands to **`_Alignof`**

534 `__alignas_is_defined`

535 Expands to the integer constant 1

536 `__alignof_is_defined`

537 Expands to the integer constant 1

538 The `__alignas_is_defined` and `__alignof_is_defined` macros shall be suitable for use in **`#if`**
539 preprocessing directives.

540 **APPLICATION USAGE**

541 None.

542 **RATIONALE**

543 None.

544 **FUTURE DIRECTIONS**

545 None.

546 **SEE ALSO**

547 None.

548 **CHANGE HISTORY**

549 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

550 Ref 7.17, 7.31.8 para 2

551 On page 345 line 11733 insert a new <stdatomic.h> section:

552 **NAME**

553 `stdatomic.h` — atomics

554 **SYNOPSIS**
555 `#include <stdatomic.h>`

556 **DESCRIPTION**

557 [CX] The functionality described on this reference page is aligned with the ISO C standard.
558 Any conflict between the requirements described here and the ISO C standard is
559 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

560 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide this
561 header nor support any of its facilities.

562 The `<stdatomic.h>` header shall define the **atomic_flag** type as a structure type. This type
563 provides the classic test-and-set functionality. It shall have two states, set and clear.
564 Operations on an object of type **atomic_flag** shall be lock free.

565 The `<stdatomic.h>` header shall define each of the atomic integer types in the following
566 table as a type that has the same representation and alignment requirements as the
567 corresponding direct type.

568 **Note:** The same representation and alignment requirements are meant to imply interchangeability
569 as arguments to functions, return values from functions, and members of unions.

Atomic type name	Direct type
atomic_bool	_Atomic _Bool
atomic_char	_Atomic char
atomic_schar	_Atomic signed char
atomic_uchar	_Atomic unsigned char
atomic_short	_Atomic short
atomic_ushort	_Atomic unsigned short
atomic_int	_Atomic int
atomic_uint	_Atomic unsigned int
atomic_long	_Atomic long
atomic_ulong	_Atomic unsigned long
atomic_llong	_Atomic long long
atomic_ullong	_Atomic unsigned long long
atomic_char16_t	_Atomic char16_t
atomic_char32_t	_Atomic char32_t
atomic_wchar_t	_Atomic wchar_t
atomic_int_least8_t	_Atomic int_least8_t
atomic_uint_least8_t	_Atomic uint_least8_t
atomic_int_least16_t	_Atomic int_least16_t
atomic_uint_least16_t	_Atomic uint_least16_t
atomic_int_least32_t	_Atomic int_least32_t
atomic_uint_least32_t	_Atomic uint_least32_t
atomic_int_least64_t	_Atomic int_least64_t
atomic_uint_least64_t	_Atomic uint_least64_t
atomic_int_fast8_t	_Atomic int_fast8_t
atomic_uint_fast8_t	_Atomic uint_fast8_t
atomic_int_fast16_t	_Atomic int_fast16_t
atomic_uint_fast16_t	_Atomic uint_fast16_t
atomic_int_fast32_t	_Atomic int_fast32_t
atomic_uint_fast32_t	_Atomic uint_fast32_t

<code>atomic_int_fast64_t</code>	<code>_Atomic int_fast64_t</code>
<code>atomic_uint_fast64_t</code>	<code>_Atomic uint_fast64_t</code>
<code>atomic_intptr_t</code>	<code>_Atomic intptr_t</code>
<code>atomic_uintptr_t</code>	<code>_Atomic uintptr_t</code>
<code>atomic_size_t</code>	<code>_Atomic size_t</code>
<code>atomic_ptrdiff_t</code>	<code>_Atomic ptrdiff_t</code>
<code>atomic_intmax_t</code>	<code>_Atomic intmax_t</code>
<code>atomic_uintmax_t</code>	<code>_Atomic uintmax_t</code>

570 The `<stdatomic.h>` header shall define the **memory_order** type as an enumerated type
571 whose enumerators shall include at least the following:

572 `memory_order_relaxed`
573 `memory_order_consume`
574 `memory_order_acquire`
575 `memory_order_release`
576 `memory_order_acq_rel`
577 `memory_order_seq_cst`

578 The `<stdatomic.h>` header shall define the following atomic lock-free macros:

579 `ATOMIC_BOOL_LOCK_FREE`
580 `ATOMIC_CHAR_LOCK_FREE`
581 `ATOMIC_CHAR16_T_LOCK_FREE`
582 `ATOMIC_CHAR32_T_LOCK_FREE`
583 `ATOMIC_WCHAR_T_LOCK_FREE`
584 `ATOMIC_SHORT_LOCK_FREE`
585 `ATOMIC_INT_LOCK_FREE`
586 `ATOMIC_LONG_LOCK_FREE`
587 `ATOMIC_LLONG_LOCK_FREE`
588 `ATOMIC_POINTER_LOCK_FREE`

589 which shall expand to constant expressions suitable for use in `#if` preprocessing directives
590 and which shall indicate the lock-free property of the corresponding atomic types (both
591 signed and unsigned). A value of 0 shall indicate that the type is never lock-free; a value of 1
592 shall indicate that the type is sometimes lock-free; a value of 2 shall indicate that the type is
593 always lock-free.

594 The `<stdatomic.h>` header shall define the macro `ATOMIC_FLAG_INIT` which shall
595 expand to an initializer for an object of type **atomic_flag**. This macro shall initialize an
596 **atomic_flag** to the clear state. An **atomic_flag** that is not explicitly initialized with
597 `ATOMIC_FLAG_INIT` is initially in an indeterminate state.

598 [OB]The `<stdatomic.h>` header shall define the macro `ATOMIC_VAR_INIT(value)` which
599 shall expand to a token sequence suitable for initializing an atomic object of a type that is
600 initialization-compatible with the non-atomic type of its *value* argument.[/OB] An atomic
601 object with automatic storage duration that is not explicitly initialized is initially in an
602 indeterminate state.

603 The `<stdatomic.h>` header shall define the macro `kill_dependency()` which shall behave as
604 described in [xref to XSH `kill_dependency()`].

605 The <stdatomic.h> header shall declare the following generic functions, where *A* refers to
606 an atomic type, *C* refers to its corresponding non-atomic type, and *M* is *C* for atomic integer
607 types or `ptrdiff_t` for atomic pointer types.

```
608     _Bool    atomic_compare_exchange_strong(volatile A *, C *, C);
609     _Bool    atomic_compare_exchange_strong_explicit(volatile A *,
610             C *, C, memory_order, memory_order);
611     _Bool    atomic_compare_exchange_weak(volatile A *, C *, C);
612     _Bool    atomic_compare_exchange_weak_explicit(volatile A *, C *,
613             C, memory_order, memory_order);
614     C        atomic_exchange(volatile A *, C);
615     C        atomic_exchange_explicit(volatile A *, C, memory_order);
616     C        atomic_fetch_add(volatile A *, M);
617     C        atomic_fetch_add_explicit(volatile A *, M,
618             memory_order);
619     C        atomic_fetch_and(volatile A *, M);
620     C        atomic_fetch_and_explicit(volatile A *, M,
621             memory_order);
622     C        atomic_fetch_or(volatile A *, M);
623     C        atomic_fetch_or_explicit(volatile A *, M, memory_order);
624     C        atomic_fetch_sub(volatile A *, M);
625     C        atomic_fetch_sub_explicit(volatile A *, M,
626             memory_order);
627     C        atomic_fetch_xor(volatile A *, M);
628     C        atomic_fetch_xor_explicit(volatile A *, M,
629             memory_order);
630     void     atomic_init(volatile A *, C);
631     _Bool    atomic_is_lock_free(const volatile A *);
632     C        atomic_load(const volatile A *);
633     C        atomic_load_explicit(const volatile A *, memory_order);
634     void     atomic_store(volatile A *, C);
635     void     atomic_store_explicit(volatile A *, C, memory_order);
```

636 It is unspecified whether any generic function declared in <stdatomic.h> is a macro or an
637 identifier declared with external linkage. If a macro definition is suppressed in order to
638 access an actual function, or a program defines an external identifier with the name of a
639 generic function, the behavior is undefined.

640 The following shall be declared as functions and may also be defined as macros. Function
641 prototypes shall be provided.

```
642     void     atomic_flag_clear(volatile atomic_flag *);
643     void     atomic_flag_clear_explicit(volatile atomic_flag *,
644             memory_order);
645     _Bool    atomic_flag_test_and_set(volatile atomic_flag *);
646     _Bool    atomic_flag_test_and_set_explicit(
647             volatile atomic_flag *, memory_order);
648     void     atomic_signal_fence(memory_order);
649     void     atomic_thread_fence(memory_order);
```

650 APPLICATION USAGE

651 None.

652 RATIONALE

653 Since operations on the `atomic_flag` type are lock free, the operations should also be
654 address-free. No other type requires lock-free operations, so the `atomic_flag` type is the

655 minimum hardware-implemented type needed to conform to this standard. The remaining
656 types can be emulated with **atomic_flag**, though with less than ideal properties.

657 The representation of atomic integer types need not have the same size as their
658 corresponding regular types. They should have the same size whenever possible, as it eases
659 effort required to port existing code.

660 **FUTURE DIRECTIONS**

661 The ISO C standard states that the macro `ATOMIC_VAR_INIT` is an obsolescent feature.
662 This macro may be removed in a future version of this standard.

663 **SEE ALSO**

664 Section 4.12.1

665 *XSH* `atomic_compare_exchange_strong()`, `atomic_compare_exchange_weak()`,
666 `atomic_exchange()`, `atomic_fetch_key()`, `atomic_flag_clear()`, `atomic_flag_test_and_set()`,
667 `atomic_init()`, `atomic_is_lock_free()`, `atomic_load()`, `atomic_signal_fence()`, `atomic_store()`,
668 `atomic_thread_fence()`, `kill_dependency()`.

669 **CHANGE HISTORY**

670 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

671 Ref 7.31.9

672 On page 345 line 11747 section `<stdbool.h>`, add OB shading to:

673 An application may undefine and then possibly redefine the macros `bool`, `true`, and `false`.

674 Ref 7.19 para 2

675 On page 346 line 11774 section `<stddef.h>`, add:

676 **max_align_t** Object type whose alignment is the greatest fundamental alignment.

677 Ref (none)

678 On page 348 line 11834 section `<stdint.h>`, change:

679 the ISO/IEC 9899: 1999 standard

680 to:

681 the ISO C standard

682 Ref 7.20.1.1 para 1

683 On page 348 line 11841 section `<stdint.h>`, change:

684 denotes a signed integer type

685 to:

686 denotes such a signed integer type

687 Ref 7.20.1.1 para 2

688 On page 348 line 11843 section <stdint.h>, change:

689 ... designates an unsigned integer type with width N . Thus, **uint24_t** denotes an unsigned
690 integer type ...

691 to:

692 ... designates an unsigned integer type with width N and no padding bits. Thus, **uint24_t**
693 denotes such an unsigned integer type ...

694 Ref 7.21.1 para 2

695 On page 355 line 12064 section <stdio.h>, change:

696 A non-array type containing all information needed to specify uniquely every position
697 within a file.

698 to:

699 A complete object type, other than an array type, capable of recording all the information
700 needed to specify uniquely every position within a file.

701 Ref 7.21.1 para 3

702 On page 357 line 12186 section <stdio.h>, change RATIONALE from:

703 There is a conflict between the ISO C standard and the POSIX definition of the
704 {TMP_MAX} macro that is addressed by ISO/IEC 9899: 1999 standard, Defect Report 336.
705 The POSIX standard is in alignment with the public record of the response to the Defect
706 Report. This change has not yet been published as part of the ISO C standard.

707 to:

708 None.

709 Ref 7.22.4.5 para 1

710 On page 359 line 12267 section <stdlib.h>, change:

711 void _Exit(int);

712 to:

713 _Noreturn void _Exit(int);

714 Ref 7.22.4.1 para 1

715 On page 359 line 12269 section <stdlib.h>, change:

716 void abort(void);

717 to:

718 _Noreturn void abort(void);

719 Ref 7.22.3.1, 7.22.4.3

720 On page 359 line 12270 section <stdlib.h>, add:

721 void *aligned_alloc(size_t, size_t);
722 int at_quick_exit(void (*)(void));

723 Ref 7.22.4.4 para 1

724 On page 360 line 12282 section <stdlib.h>, change:

725 void exit(int);

726 to:

727 _Noreturn void exit(int);

728 Ref 7.22.4.7

729 On page 360 line 12309 section <stdlib.h>, add:

730 _Noreturn void quick_exit(int);

731 Ref 7.23

732 On page 363 line 12380 insert a new <stdnoreturn.h> section:

733 **NAME**

734 stdnoreturn.h — noreturn macro

735 **SYNOPSIS**

736 #include <stdnoreturn.h>

737 **DESCRIPTION**

738 [CX] The functionality described on this reference page is aligned with the ISO C standard.
739 Any conflict between the requirements described here and the ISO C standard is
740 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

741 The <stdnoreturn.h> header shall define the macro noreturn which shall expand to
742 **_Noreturn**.

743 **APPLICATION USAGE**

744 None.

745 **RATIONALE**

746 None.

747 **FUTURE DIRECTIONS**

748 None.

749 **SEE ALSO**

750 None.

751 **CHANGE HISTORY**

752 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

753 Ref G.7

754 On page 422 line 14340 section <tgmath.h>, add two new paragraphs:

755 [MXC]Type-generic macros that accept complex arguments shall also accept imaginary
756 arguments. If an argument is imaginary, the macro shall expand to an expression whose type
757 is real, imaginary, or complex, as appropriate for the particular function: if the argument is
758 imaginary, then the types of *cos()*, *cosh()*, *fabs()*, *carg()*, *cimag()*, and *creal()* shall be real;
759 the types of *sin()*, *tan()*, *sinh()*, *tanh()*, *asin()*, *atan()*, *asinh()*, and *atanh()* shall be imaginary;
760 and the types of the others shall be complex.

761 Given an imaginary argument, each of the type-generic macros *cos()*, *sin()*, *tan()*, *cosh()*,
762 *sinh()*, *tanh()*, *asin()*, *atan()*, *asinh()*, *atanh()* is specified by a formula in terms of real
763 functions:

764 *cos(iy)* = *cosh(y)*
765 *sin(iy)* = *i sinh(y)*
766 *tan(iy)* = *i tanh(y)*
767 *cosh(iy)* = *cos(y)*
768 *sinh(iy)* = *i sin(y)*
769 *tanh(iy)* = *i tan(y)*
770 *asin(iy)* = *i asinh(y)*
771 *atan(iy)* = *i atanh(y)*
772 *asinh(iy)* = *i asin(y)*
773 *atanh(iy)* = *i atan(y)*
774 [/MXC]

775 Ref (none)

776 On page 423 line 14404 section <tgmath.h>, change:

777 the ISO/IEC 9899: 1999 standard

778 to:

779 the ISO C standard

780 Ref 7.26

781 On page 424 line 14425 insert a new <threads.h> section:

782 **NAME**

783 threads.h — ISO C threads

784 **SYNOPSIS**

785 #include <threads.h>

786 **DESCRIPTION**

787 [CX] The functionality described on this reference page is aligned with the ISO C standard.
788 Any conflict between the requirements described here and the ISO C standard is
789 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

790 [CX] Implementations shall not define the macro `__STDC_NO_THREADS__`, except for
791 profile implementations that define `_POSIX_SUBPROFILE` (see [xref to 2.1.5.1
792 Subprofiling Considerations]) in <unistd.h>, which may define `__STDC_NO_THREADS__`
793 and, if they do so, need not provide this header nor support any of its facilities.[/CX]

794 The <**threads.h**> header shall define the macros `thread_local` which shall expand to
795 `_Thread_local`, `ONCE_FLAG_INIT` which shall expand to a value that can be used to
796 initialize an object of type `once_flag`, and `TSS_DTOR_ITERATIONS` which shall expand to
797 an integer constant expression representing the maximum number of times that destructors
798 will be called when a thread terminates and shall be suitable for use in `#if` preprocessing
799 directives. [CX]If `{PTHREAD_DESTRUCTOR_ITERATIONS}` is defined in <**limits.h**>,
800 the value of `TSS_DTOR_ITERATIONS` shall be equal to
801 `{PTHREAD_DESTRUCTOR_ITERATIONS}`; otherwise, the value of
802 `TSS_DTOR_ITERATIONS` shall be greater than or equal to the value of
803 `{_POSIX_THREAD_DESTRUCTOR_ITERATIONS}` and shall be less than or equal to the
804 maximum positive value that can be returned by a call to
805 `sysconf(_SC_THREAD_DESTRUCTOR_ITERATIONS)` in any process.[/CX]

806 The <**threads.h**> header shall define the types `cond_t`, `mtx_t`, `once_flag`, `thrd_t`, and `tss_t`
807 as complete object types, the type `thrd_start_t` as the function pointer type `int (*)(void*)`,
808 and the type `tss_dtor_t` as the function pointer type `void (*)(void*)`. [CX]The type `thrd_t`
809 shall be defined to be the same type that `pthread_t` is defined to be in <**pthread.h**>.[/CX]

810 The <**threads.h**> header shall define the enumeration constants `mtx_plain`,
811 `mtx_recursive`, `mtx_timed`, `thrd_busy`, `thrd_error`, `thrd_nomem`, `thrd_success`
812 and `thrd_timedout`.

813 The following shall be declared as functions and may also be defined as macros. Function
814 prototypes shall be provided.

```
815 void      call_once(once_flag *, void (*)(void));
816 int       cnd_broadcast(cond_t *);
817 void      cnd_destroy(cond_t *);
818 int       cnd_init(cond_t *);
819 int       cnd_signal(cond_t *);
820 int       cnd_timedwait(cond_t * restrict, mtx_t * restrict,
821                        const struct timespec * restrict);
822 int       cnd_wait(cond_t *, mtx_t *);
823 void      mtx_destroy(mtx_t *);
824 int       mtx_init(mtx_t *, int);
825 int       mtx_lock(mtx_t *);
826 int       mtx_timedlock(mtx_t * restrict,
827                        const struct timespec * restrict);
828 int       mtx_trylock(mtx_t *);
829 int       mtx_unlock(mtx_t *);
830 int       thrd_create(thrd_t *, thrd_start_t, void *);
831 thrd_t    thrd_current(void);
832 int       thrd_detach(thrd_t);
833 int       thrd_equal(thrd_t, thrd_t);
834 _Noreturn void thrd_exit(int);
835 int       thrd_join(thrd_t, int *);
836 int       thrd_sleep(const struct timespec *,
837                    struct timespec *);
838 void      thrd_yield(void);
839 int       tss_create(tss_t *, tss_dtor_t);
840 void      tss_delete(tss_t);
841 void      *tss_get(tss_t);
842 int       tss_set(tss_t, void *);
```

843 Inclusion of the <**threads.h**> header shall make symbols defined in the header <**time.h**>

844 visible.

845 APPLICATION USAGE

846 The `<threads.h>` header is optional in the ISO C standard but is mandated by POSIX.1-
847 20xx. Note however that subprofiles can choose to make this header optional (see [xref to
848 2.1.5.1 Subprofiling Considerations]), and therefore application portability to subprofile
849 implementations would benefit from checking whether `__STDC_NO_THREADS__` is
850 defined before inclusion of `<threads.h>`.

851 The features provided by `<threads.h>` are not as extensive as those provided by
852 `<pthread.h>`. It is present on POSIX implementations in order to facilitate porting of ISO C
853 programs that use it. It is recommended that applications intended for use on POSIX
854 implementations use `<pthread.h>` rather than `<threads.h>` even if none of the additional
855 features are needed initially, to save the need to convert should the need to use them arise
856 later in the application's lifecycle.

857 RATIONALE

858 Although the `<threads.h>` header is optional in the ISO C standard, it is mandated by
859 POSIX.1-20xx because `<pthread.h>` is mandatory and the interfaces in `<threads.h>` can
860 easily be implemented as a thin wrapper for interfaces in `<pthread.h>`.

861 The type `thrd_t` is required to be defined as the same type that `pthread_t` is defined to be in
862 `<pthread.h>` because `thrd_current()` and `pthread_self()` need to return the same thread ID
863 when called from the initial thread. However, these types are not fully interchangeable (that
864 is, it is not always possible to pass a thread ID obtained as a `thrd_t` to a function that takes a
865 `pthread_t`, and vice versa) because threads created using `thrd_create()` have a different exit
866 status than `pthread` threads, which is reflected in differences between the prototypes for
867 `thrd_create()` and `pthread_create()`, `thrd_exit()` and `pthread_exit()`, and `thrd_join()` and
868 `pthread_join()`; also, `thrd_join()` has no way to indicate that a thread was cancelled.

869 The standard developers considered making it implementation-defined whether the types
870 `cond_t`, `mtx_t` and `tss_t` are interchangeable with the corresponding types `pthread_cond_t`,
871 `pthread_mutex_t` and `pthread_key_t` defined in `<pthread.h>` (that is, whether any
872 function that can be called with a valid `cond_t` can also be called with a valid
873 `pthread_cond_t`, and vice versa, and likewise for the other types). However, this would
874 have meant extending `mtx_lock()` to provide a way for it to indicate that the owner of a
875 mutex has terminated (equivalent to [EOWNERDEAD]). It was felt that such an extension
876 would be invention. Although there was no similar concern for `cond_t` and `tss_t`, they were
877 treated the same way as `mtx_t` for consistency.

878 FUTURE DIRECTIONS

879 None.

880 SEE ALSO

881 `<limits.h>`, `<pthread.h>`, `<time.h>`

882 XSH Section 2.9, `call_once()`, `cond_broadcast()`, `cond_destroy()`, `cond_timedwait()`,
883 `mtx_destroy()`, `mtx_lock()`, `sysconf()`, `thrd_create()`, `thrd_current()`, `thrd_detach()`,
884 `thrd_equal()`, `thrd_exit()`, `thrd_join()`, `thrd_sleep()`, `thrd_yield()`, `tss_create()`, `tss_delete()`,
885 `tss_get()`.

886 CHANGE HISTORY

887 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

888 Ref 7.27.1 para 4

889 On page 425 line 14453 section <time.h>, remove the CX shading from:

890 The <**time.h**> header shall declare the **timespec** structure, which shall include at least the
891 following members:

892	<code>time_t</code>	<code>tv_sec</code>	Seconds.
893	<code>long</code>	<code>tv_nsec</code>	Nanoseconds.

894 and change the members to:

895	<code>time_t</code>	<code>tv_sec</code>	Whole seconds.
896	<code>long</code>	<code>tv_nsec</code>	Nanoseconds [0, 999 999 999].

897 Ref 7.27.1 para 2

898 On page 426 line 14467 section <time.h>, add to the list of macros:

899	<code>TIME_UTC</code>	An integer constant greater than 0 that designates the UTC time base 900 in calls to <i>timespec_get()</i> . The value shall be suitable for use in #if 901 preprocessing directives.
-----	-----------------------	--

902 Ref 7.27.2.5

903 On page 427 line 14524 section <time.h>, add to the list of functions:

```
904     int          timespec_get(struct timespec *, int);
```

905 Ref 7.28

906 On page 433 line 14736 insert a new <uchar.h> section:

907 **NAME**

908 uchar.h — Unicode character handling

909 **SYNOPSIS**

```
910     #include <uchar.h>
```

911 **DESCRIPTION**

912 [CX] The functionality described on this reference page is aligned with the ISO C standard.
913 Any conflict between the requirements described here and the ISO C standard is
914 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

915 The <**uchar.h**> header shall define the following types:

916 **mbstate_t** As described in <**wchar.h**>.

917 **size_t** As described in <**stddef.h**>.

918 **char16_t** The same type as **uint_least16_t**, described in <**stdint.h**>.

919 **char32_t** The same type as **uint_least32_t**, described in <**stdint.h**>.

920 The following shall be declared as functions and may also be defined as macros. Function
921 prototypes shall be provided.

```
922 size_t    c16rtomb(char *restrict, char16_t,  
923             mbstate_t *restrict);  
924 size_t    c32rtomb(char *restrict, char32_t,  
925             mbstate_t *restrict);  
926 size_t    mbrtoc16(char16_t *restrict, const char *restrict,  
927                 size_t, mbstate_t *restrict);  
928 size_t    mbrtoc32(char32_t *restrict, const char *restrict,  
929                 size_t, mbstate_t *restrict);
```

930 [CX]Inclusion of the <uchar.h> header may make visible all symbols from the headers
931 <stddef.h>, <stdint.h> and <wchar.h>.[/CX]

932 APPLICATION USAGE

933 None.

934 RATIONALE

935 None.

936 FUTURE DIRECTIONS

937 None.

938 SEE ALSO

939 <stddef.h>, <stdint.h>, <wchar.h>

940 XSH *c16rtomb()*, *c32rtomb()*, *mbrtoc16()*, *mbrtoc32()*

941 CHANGE HISTORY

942 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

943 Ref 7.22.4.5 para 1

944 On page 447 line 15388 section <unistd.h>, change:

```
945 void      _exit(int);
```

946 to:

```
947 _Noreturn void _exit(int);
```

948 Ref 7.29.1 para 2

949 On page 458 line 15801 section <wchar.h>, change:

950 **mbstate_t** An object type other than an array type ...

951 to:

952 **mbstate_t** A complete object type other than an array type ...

953 Changes to XSH

954 Ref 7.1.4 paras 5, 6

955 On page 471 line 16224 section 2.1.1 Use and Implementation of Functions, add two numbered list
956 items:

957 6. Functions shall prevent data races as follows: A function shall not directly or indirectly
958 access objects accessible by threads other than the current thread unless the objects are
959 accessed directly or indirectly via the function's arguments. A function shall not directly or
960 indirectly modify objects accessible by threads other than the current thread unless the
961 objects are accessed directly or indirectly via the function's non-const arguments.
962 Implementations may share their own internal objects between threads if the objects are not
963 visible to applications and are protected against data races.

964 7. Functions shall perform all operations solely within the current thread if those operations
965 have effects that are visible to applications.

966 Ref K.3.1.1

967 On page 473 line 16283 section 2.2.1, add a new subsection:

968 2.2.1.3 *The `__STDC_WANT_LIB_EXT1__` Feature Test Macro*

969 A POSIX-conforming [XSI] or XSI-conforming [/XSI] application can define the feature test
970 macro `__STDC_WANT_LIB_EXT1__` before inclusion of any header.

971 When an application includes a header described by POSIX.1-20xx, and when this feature
972 test macro is defined to have the value 1, the header may make visible those symbols
973 specified for the header in Annex K of the ISO C standard that are not already explicitly
974 permitted by POSIX.1-20xx to be made visible in the header. These symbols are listed in
975 [xref to 2.2.2].

976 When an application includes a header described by POSIX.1-20xx, and when this feature
977 test macro is either undefined or defined to have the value 0, the header shall not make any
978 additional symbols visible that are not already made visible by the feature test macro
979 `_POSIX_C_SOURCE` [XSI] or `_XOPEN_SOURCE` [/XSI] as described above, except when
980 enabled by another feature test macro.

981 Ref 7.31.8 para 1

982 On page 475 line 16347 section 2.2.2, insert a row in the table:

<code><stdatomic.h></code>	<code>atomic_[a-z], memory_[a-z]</code>		
----------------------------------	---	--	--

983 Ref 7.31.15 para 1

984 On page 476 line 16373 section 2.2.2, insert a row in the table:

<code><threads.h></code>	<code>cnd_[a-z], mtx_[a-z], thrd_[a-z], tss_[a-z]</code>		
--------------------------------	--	--	--

985 Ref 7.31.8 para 1

986 On page 477 line 16410 section 2.2.2, insert a row in the table:

<stdatomic.h>	ATOMIC_[A-Z]
---------------	--------------

987 Ref 7.31.14 para 1

988 On page 477 line 16417 section 2.2.2, insert a row in the table:

<time.h>	TIME_[A-Z]
----------	------------

989 Ref K.3.4 - K.3.9

990 On page 477 line 16436 section 2.2.2 The Name Space, add:

991 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
 992 (see [xref to 2.2.1]), implementations may add symbols to the headers shown in the
 993 following table provided the identifiers for those symbols have one of the corresponding
 994 complete names in the table.

Header	Complete Name
<stdio.h>	fopen_s, fprintf_s, freopen_s, fscanf_s, gets_s, printf_s, scanf_s, snprintf_s, sprintf_s, sscanf_s, tmpfile_s, tmpnam_s, vfprintf_s, vfscanf_s, vprintf_s, vscanf_s, vsnprintf_s, vsprintf_s, vsscanf_s
<stdlib.h>	abort_handler_s, bsearch_s, getenv_s, ignore_handler_s, mbstowcs_s, qsort_s, set_constraint_handler_s, wcstombs_s, wctomb_s
<time.h>	asctime_s, ctime_s, gmtime_s, localtime_s
<wchar.h>	fwprintf_s, fwscanf_s, mbsrtowcs_s, snwprintf_s, swprintf_s, swscanf_s, vfwprintf_s, vfwscanf_s, vsnwprintf_s, vswprintf_s, vswscanf_s, vwprintf_s, vwscanf_s, wcrctomb_s, wmemcpy_s, wmemmove_s, wprintf_s, wscanf_s

995 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
 996 (see [xref to 2.2.1]), if any header in the following table is included, macros with the
 997 complete names shown may be defined.

Header	Complete Name
<stdint.h>	RSIZE_MAX
<stdio.h>	L_tmpnam_s, TMP_MAX_S

998 **Note:** The above two tables only include those symbols from Annex K of the ISO C standard that
 999 are not already allowed to be visible by entries in earlier tables in this section.

1000 Ref 7.1.3 para 1

1001 On page 478 line 16438 section 2.2.2, change:

1002 With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that
 1003 begin with an <underscore> and either an uppercase letter or another <underscore> are
 1004 always reserved for any use by the implementation.

1005 to:

1006 With the exception of identifiers beginning with the prefix `_POSIX_` and those identifiers

1007 which are lexically identical to keywords defined by the ISO C standard (for example
1008 **_Bool**), all identifiers that begin with an <underscore> and either an uppercase letter or
1009 another <underscore> are always reserved for any use by the implementation.

1010 Ref 7.1.3 para 1

1011 On page 478 line 16448 section 2.2.2, change:

1012 that have external linkage are always reserved

1013 to:

1014 that have external linkage and *errno* are always reserved

1015 Ref 7.1.3 para 1

1016 On page 479 line 16453 section 2.2.2, add the following in the appropriate place in the list:

1017	aligned_alloc	c32rtomb
1018	at_quick_exit	call_once
1019	atomic_compare_exchange_strong	cnd_broadcast
1020	atomic_compare_exchange_strong_explicit	cnd_destroy
1021	atomic_compare_exchange_weak	cnd_init
1022	atomic_compare_exchange_weak_explicit	cnd_signal
1023	atomic_exchange	cnd_timedwait
1024	atomic_exchange_explicit	cnd_wait
1025	atomic_fetch_add	kill_dependency
1026	atomic_fetch_add_explicit	mbrtoc16
1027	atomic_fetch_and	mbrtoc32
1028	atomic_fetch_and_explicit	mtx_destroy
1029	atomic_fetch_or	mtx_init
1030	atomic_fetch_or_explicit	mtx_lock
1031	atomic_fetch_sub	mtx_timedlock
1032	atomic_fetch_sub_explicit	mtx_trylock
1033	atomic_fetch_xor	mtx_unlock
1034	atomic_fetch_xor_explicit	quick_exit
1035	atomic_flag_clear	thrd_create
1036	atomic_flag_clear_explicit	thrd_current
1037	atomic_flag_test_and_set	thrd_detach
1038	atomic_flag_test_and_set_explicit	thrd_equal
1039	atomic_init	thrd_exit
1040	atomic_is_lock_free	thrd_join
1041	atomic_load	thrd_sleep
1042	atomic_load_explicit	thrd_yield
1043	atomic_signal_fence	timespec_get
1044	atomic_store	tss_create
1045	atomic_store_explicit	tss_delete
1046	atomic_thread_fence	tss_get
1047	c16rtomb	tss_set

1048 Ref 7.1.2 para 4

1049 On page 480 line 16551 section 2.2.2, change:

1050 Prior to the inclusion of a header, the application shall not define any macros with names

1051 lexically identical to symbols defined by that header.

1052 to:

1053 Prior to the inclusion of a header, or when any macro defined in the header is expanded, the
1054 application shall not define any macros with names lexically identical to symbols defined by
1055 that header.

1056 Ref 7.26.5.1
1057 On page 490 line 16980 section 2.4.2 Realtime Signal Generation and Delivery, change:

1058 The function shall be executed in an environment as if it were the *start_routine* for a newly
1059 created thread with thread attributes specified by *sigev_notify_attributes*.

1060 to:

1061 The function shall be executed in a newly created thread as if it were the *start_routine* for a
1062 call to *pthread_create()* with the thread attributes specified by *sigev_notify_attributes*.

1063 Ref 7.14.1.1 para 5
1064 On page 493 line 17088 section 2.4.3 Signal Actions, change:

1065 with static storage duration

1066 to:

1067 with static or thread storage duration that is not a lock-free atomic object

1068 Ref 7.14.1.1 para 5
1069 On page 493 line 17090 section 2.4.3 Signal Actions, after applying bug 711 change:

1070 other than one of the functions and macros listed in the following table

1071 to:

1072 other than one of the functions and macros specified below as being async-signal-safe

1073 Ref 7.14.1.1 para 5
1074 On page 494 line 17133 section 2.4.3 Signal Actions, add *quick_exit()* to the table of async-signal-
1075 safe functions.

1076 Ref 7.14.1.1 para 5
1077 On page 494 line 17147 section 2.4.3 Signal Actions, change:

1078 Any function or function-like macro not in the above table may be unsafe with respect to
1079 signals.

1080 to:

1081 In addition, the functions in `<stdatomic.h>` other than *atomic_init()* shall be async-signal-
1082 safe when the atomic arguments are lock-free, and the *atomic_is_lock_free()* function shall
1083 be async-signal-safe when called with an atomic argument.

- 1084 All other functions (including generic functions) and function-like macros may be unsafe
1085 with respect to signals.
- 1086 Ref 7.21.2 para 7,8
1087 On page 496 line 17228 section 2.5 Standard I/O Streams, add a new paragraph:
- 1088 Each stream shall have an associated lock that is used to prevent data races when multiple
1089 threads of execution access a stream, and to restrict the interleaving of stream operations
1090 performed by multiple threads. Only one thread can hold this lock at a time. The lock shall
1091 be reentrant: a single thread can hold the lock multiple times at a given time. All functions
1092 that read, write, position, or query the position of a stream, [CX]except those with names
1093 ending *_unlocked*[/CX], shall lock the stream [CX] as if by a call to *flockfile()*[/CX] before
1094 accessing it and release the lock [CX] as if by a call to *funlockfile()*[/CX] when the access is
1095 complete.
- 1096 Ref (none)
1097 On page 498 line 17312 section 2.5.2 Stream Orientation and Encoding Rules, change:
- 1098 For conformance to the ISO/IEC 9899: 1999 standard, the definition of a stream includes an
1099 “orientation”.
- 1100 to:
- 1101 The definition of a stream includes an “orientation”.
- 1102 Ref 7.26.5.8
1103 On page 508 line 17720 section 2.8.4 Process Scheduling, change:
- 1104 When a running thread issues the *sched_yield()* function
- 1105 to:
- 1106 When a running thread issues the *sched_yield()* or *thrd_yield()* function
- 1107 Ref 7.17.2.2 para 3, 7.22.2.2 para 3
1108 On page 513 line 17907,17916 section 2.9.1 Thread-Safety, add *atomic_init()* and *srand()* to the list
1109 of functions that need not be thread-safe.
- 1110 Ref 7.12.8.3, 7.22.4.8
1111 On page 513 line 17907-17927 section 2.9.1 Thread-Safety, delete the following from the list of
1112 functions that need not be thread-safe:
- 1113 *lgamma()*, *lgammaf()*, *lgammal()*, *system()*
- 1114 [Note to reviewers: deletion of *mblen\(\)*, *mbtowc\(\)*, and *wctomb\(\)* from this list is the subject of](#)
1115 [Mantis bug 708.](#)
- 1116 Ref 7.28.1 para 1
1117 On page 513 line 17928 section 2.9.1 Thread-Safety, change:
- 1118 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument.

1119 The *mbrlen()*, *mbrtowc()*, *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and
1120 *wcsrtombs()* functions need not be thread-safe if passed a NULL *ps* argument.

1121 to:

1122 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a null pointer
1123 argument. The *c16rtomb()*, *c32rtomb()*, *mbrlen()*, *mbrtoc16()*, *mbrtoc32()*, *mbrtowc()*,
1124 *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and *wcsrtombs()* functions need not
1125 be thread-safe if passed a null *ps* argument. The *lgamma()*, *lgammaf()*, and *lgammal()*
1126 functions shall be thread-safe [XSI]except that they need not avoid data races when storing a
1127 value in the *signgam* variable[/XSI].

1128 Ref 7.1.4 para 5

1129 On page 513 line 17934 section 2.9.1 Thread-Safety, change:

1130 Implementations shall provide internal synchronization as necessary in order to satisfy this
1131 requirement.

1132 to:

1133 Some functions that are not required to be thread-safe are nevertheless required to avoid data
1134 races with either all or some other functions, as specified on their individual reference pages.

1135 Implementations shall provide internal synchronization as necessary in order to satisfy
1136 thread-safety requirements.

1137 Ref 7.26.5

1138 On page 513 line 17944 section 2.9.2 Thread IDs, change:

1139 The lifetime of a thread ID ends after the thread terminates if it was created with the
1140 *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or if *pthread_detach()* or
1141 *pthread_join()* has been called for that thread.

1142 to:

1143 The lifetime of a thread ID ends after the thread terminates if it was created using
1144 *pthread_create()* with the *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or
1145 if *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()* has been called for that
1146 thread.

1147 Ref 7.26.5

1148 On page 514 line 17950 section 2.9.2 Thread IDs, change:

1149 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1150 *pthread_detach()* or *pthread_join()*.

1151 to:

1152 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1153 *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()*.

1154 Ref 7.26.4

1155 On page 514 line 17956 section 2.9.3 Thread Mutexes, change:

1156 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

1157 to:

1158 A thread shall become the owner of a mutex, *m*, of type **pthread_mutex_t** when one of the
1159 following occurs:

1160 Ref 7.26.3, 7.26.4

1161 On page 514 line 17972 section 2.9.3 Thread Mutexes, add two new paragraphs and lists:

1162 A thread shall become the owner of a mutex, *m*, of type **mtx_t** when one of the following
1163 occurs:

- 1164 • It calls *mtx_lock()* with *m* as the *mtx* argument and the call returns *thrd_success*.
- 1165 • It calls *mtx_trylock()* with *m* as the *mtx* argument and the call returns
1166 *thrd_success*.
- 1167 • It calls *mtx_timedlock()* with *m* as the *mtx* argument and the call returns
1168 *thrd_success*.
- 1169 • It calls *cond_wait()* with *m* as the *mtx* argument and the call returns *thrd_success*.
- 1170 • It calls *cond_timedwait()* with *m* as the *mtx* argument and the call returns
1171 *thrd_success* or *thrd_timedout*.

1172 The thread shall remain the owner of *m* until one of the following occurs:

- 1173 • It executes *mtx_unlock()* with *m* as the *mtx* argument.
- 1174 • It blocks in a call to *cond_wait()* with *m* as the *mtx* argument.
- 1175 • It blocks in a call to *cond_timedwait()* with *m* as the *mtx* argument.

1176 Ref 7.26.4

1177 On page 514 line 17980 section 2.9.3 Thread Mutexes, change:

1178 Robust mutexes provide a means to enable the implementation to notify other threads in the
1179 event of a process terminating while one of its threads holds a mutex lock.

1180 to:

1181 Robust mutexes provide a means to enable the implementation to notify other threads in the
1182 event of a process terminating while one of its threads holds a lock on a mutex of type
1183 **pthread_mutex_t**.

1184 Ref 7.26.5

1185 On page 517 line 18085 section 2.9.5 Thread Cancellation, change:

1186 The thread cancellation mechanism allows a thread to terminate the execution of any other
1187 thread in the process in a controlled manner.

1188 to:

1189 The thread cancellation mechanism allows a thread to terminate the execution of any thread

1190 in the process, except for threads created using *thrd_create()*, in a controlled manner.

1191 Ref 7.26.3, 7.26.5.6

1192 On page 518 line 18119-18137 section 2.9.5.2 Cancellation Points, add the following to the list of
1193 functions that are required to be cancellation points:

1194 *cnd_timedwait()*, *cnd_wait()*, *thrd_join()*, *thrd_sleep()*

1195 Ref 7.26.5

1196 On page 520 line 18225 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1197 Each thread maintains a list of cancellation cleanup handlers.

1198 to:

1199 Each thread that was not created using *thrd_create()* maintains a list of cancellation cleanup
1200 handlers.

1201 Ref 7.26.6.1

1202 On page 521 line 18240 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1203 as described for *pthread_key_create()*

1204 to:

1205 as described for *pthread_key_create()* and *tss_create()*

1206 Ref 7.26

1207 On page 523 line 18337 section 2.9.9 Synchronization Object Copies and Alternative Mappings,
1208 add a new sentence:

1209 For ISO C functions declared in `<threads.h>`, the above requirements shall apply as if
1210 condition variables of type **cnd_t** and mutexes of type **mtx_t** have a process-shared attribute
1211 that is set to `PTHREAD_PROCESS_PRIVATE`.

1212 Ref 7.26.3

1213 On page 547 line 19279 section 2.12.1 Defined Types, change:

1214 **pthread_cond_t**

1215 to

1216 **pthread_cond_t, cnd_t**

1217 Ref 7.26.6, 7.26.4

1218 On page 547 line 19281 section 2.12.1 Defined Types, change:

1219 **pthread_key_t**

1220 **pthread_mutex_t**

1221 to

1222 **pthread_key_t, tss_t**
1223 **pthread_mutex_t, mtx_t**

1224 Ref 7.26.2.1
1225 On page 547 line 19284 section 2.12.1 Defined Types, change:

1226 **pthread_once_t**

1227 to

1228 **pthread_once_t, once_flag**

1229 Ref 7.26.5
1230 On page 547 line 19287 section 2.12.1 Defined Types, change:

1231 **pthread_t**

1232 to

1233 **pthread_t, thrd_t**

1234 Ref 7.3.9.3
1235 On page 552 line 19370 insert a new CMPLX() section:

1236 **NAME**

1237 CMPLX — make a complex value

1238 **SYNOPSIS**

1239 #include <complex.h>

1240 double complex CMPLX(double x, double y);
1241 float complex CMPLXF(float x, float y);
1242 long double complex CMPLXL(long double x, long double y);

1243 **DESCRIPTION**

1244 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1245 Any conflict between the requirements described here and the ISO C standard is
1246 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1247 The CMPLX macros shall expand to an expression of the specified complex type, with the
1248 real part having the (converted) value of x and the imaginary part having the (converted)
1249 value of y . The resulting expression shall be suitable for use as an initializer for an object
1250 with static or thread storage duration, provided both arguments are likewise suitable.

1251 **RETURN VALUE**

1252 The CMPLX macros return the complex value $x + iy$ (where i is the imaginary unit).

1253 These macros shall behave as if the implementation supported imaginary types and the
1254 definitions were:

```
1255 #define CMPLX(x, y) ((double complex)((double)(x) + \  
1256     _Imaginary_I * (double)(y)))  
1257 #define CMPLXF(x, y) ((float complex)((float)(x) + \  

```

```
1258         _Imaginary_I * (float)(y)))
1259     #define CmplXL(x, y) ((long double complex)((long double)(x) + \
1260         _Imaginary_I * (long double)(y)))
```

1261 **ERRORS**

1262 No errors are defined.

1263 **EXAMPLES**

1264 None.

1265 **APPLICATION USAGE**

1266 None.

1267 **RATIONALE**

1268 None.

1269 **FUTURE DIRECTIONS**

1270 None.

1271 **SEE ALSO**

1272 XBD <complex.h>

1273 **CHANGE HISTORY**

1274 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1275 Ref 7.22.4.5 para 1

1276 On page 553 line 19384 section `_Exit()`, change:

```
1277     void _Exit(int status);
```

```
1278     #include <unistd.h>
```

```
1279     void _exit(int status);
```

1280 to:

```
1281     _Noreturn void _Exit(int status);
```

```
1282     #include <unistd.h>
```

```
1283     _Noreturn void _exit(int status);
```

1284 Ref 7.22.4.5 para 2

1285 On page 553 line 19396 section `_Exit()`, change:

1286 shall not call functions registered with `atexit()` nor any registered signal handlers

1287 to:

1288 shall not call functions registered with `atexit()` nor `at_quick_exit()`, nor any registered signal
1289 handlers

1290 Ref (none)

1291 On page 557 line 19562 section `_Exit()`, change:

1292 The ISO/IEC 9899: 1999 standard adds the `_Exit()` function

1293 to:

1294 The ISO/IEC 9899: 1999 standard added the `_Exit()` function

1295 Ref 7.22.4.3, 7.22.4.7

1296 On page 557 line 19568 section `_Exit()`, add `at_quick_exit` and `quick_exit` to the SEE ALSO section.

1297 Ref 7.22.4.1 para 1

1298 On page 565 line 19761 section `abort()`, change:

1299 `void abort(void);`

1300 to:

1301 `_Noreturn void abort(void);`

1302 Ref (none)

1303 On page 565 line 19785 section `abort()`, change:

1304 The ISO/IEC 9899: 1999 standard requires the `abort()` function to be async-signal-safe.

1305 to:

1306 The ISO/IEC 9899: 1999 standard required (and the current standard still requires) the
1307 `abort()` function to be async-signal-safe.

1308 Ref 7.22.3.1

1309 On page 597 line 20771 insert the following new `aligned_alloc()` section:

1310 **NAME**

1311 `aligned_alloc` — allocate memory with a specified alignment

1312 **SYNOPSIS**

1313 `#include <stdlib.h>`

1314 `void *aligned_alloc(size_t alignment, size_t size);`

1315 **DESCRIPTION**

1316 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1317 Any conflict between the requirements described here and the ISO C standard is
1318 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1319 The `aligned_alloc()` function shall allocate unused space for an object whose alignment is
1320 specified by `alignment`, whose size in bytes is specified by `size` and whose value is
1321 indeterminate.

1322 The order and contiguity of storage allocated by successive calls to `aligned_alloc()` is
1323 unspecified. Each such allocation shall yield a pointer to an object disjoint from any other
1324 object. The pointer returned shall point to the start (lowest byte address) of the allocated

1325 space. If the value of *alignment* is not a valid alignment supported by the implementation, a
1326 null pointer shall be returned. If the space cannot be allocated, a null pointer shall be
1327 returned. If the size of the space requested is 0, the behavior is implementation-defined:
1328 either a null pointer shall be returned to indicate an error, or the behavior shall be as if the
1329 size were some non-zero value, except that the behavior is undefined if the returned pointer
1330 is used to access an object.

1331 For purposes of determining the existence of a data race, *aligned_alloc()* shall behave as
1332 though it accessed only memory locations accessible through its arguments and not other
1333 static duration storage. The function may, however, visibly modify the storage that it
1334 allocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix_memalign()*,
1335 [/ADV] and *realloc()* that allocate or deallocate a particular region of memory shall occur in
1336 a single total order (see [xref to XBD 4.12.1]), and each such deallocation call shall
1337 synchronize with the next allocation (if any) in this order.

1338 RETURN VALUE

1339 Upon successful completion with *size* not equal to 0, *aligned_alloc()* shall return a pointer to
1340 the allocated space. If *size* is 0, either:

- 1341 • A null pointer shall be returned [CX]and *errno* may be set to an implementation-
1342 defined value,[/CX] or
- 1343 • A pointer to the allocated space shall be returned. The application shall ensure that
1344 the pointer is not used to access an object.

1345 Otherwise, it shall return a null pointer [CX]and set *errno* to indicate the error[/CX].

1346 ERRORS

1347 The *aligned_alloc()* function shall fail if:

- | | | |
|------|--------------|---|
| 1348 | [CX][EINVAL] | The value of <i>alignment</i> is not a valid alignment supported by the |
| 1349 | | implementation. |
| 1350 | [ENOMEM] | Insufficient storage space is available.[/CX] |

1351 EXAMPLES

1352 None.

1353 APPLICATION USAGE

1354 None.

1355 RATIONALE

1356 None.

1357 FUTURE DIRECTIONS

1358 None.

1359 SEE ALSO

1360 *calloc*, *free*, *getrlimit*, *malloc*, *posix_memalign*, *realloc*

1361 XBD <stdlib.h>

1362 **CHANGE HISTORY**

1363 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1364 Ref 7.27.3, 7.1.4 para 5

1365 On page 600 line 20911 section `asctime()`, change:

1366 [CX]The `asctime()` function need not be thread-safe.[/CX]

1367 to:

1368 The `asctime()` function need not be thread-safe; however, `asctime()` shall avoid data races
1369 with all functions other than itself, `ctime()`, `gmtime()` and `localtime()`.

1370 Ref 7.22.4.3

1371 On page 618 line 21380 insert the following new `at_quick_exit()` section:

1372 **NAME**

1373 `at_quick_exit` — register a function to be called from `quick_exit()`

1374 **SYNOPSIS**

1375 `#include <stdlib.h>`

1376 `int at_quick_exit(void (*func)(void));`

1377 **DESCRIPTION**

1378 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1379 Any conflict between the requirements described here and the ISO C standard is
1380 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1381 The `at_quick_exit()` function shall register the function pointed to by `func`, to be called
1382 without arguments should `quick_exit()` be called. It is unspecified whether a call to the
1383 `at_quick_exit()` function that does not happen before the `quick_exit()` function is called will
1384 succeed.

1385 At least 32 functions can be registered with `at_quick_exit()`.

1386 [CX]After a successful call to any of the `exec` functions, any functions previously registered
1387 by `at_quick_exit()` shall no longer be registered.[/CX]

1388 **RETURN VALUE**

1389 Upon successful completion, `at_quick_exit()` shall return 0; otherwise, it shall return a non-
1390 zero value.

1391 **ERRORS**

1392 No errors are defined.

1393 **EXAMPLES**

1394 None.

1395 **APPLICATION USAGE**

1396 The `at_quick_exit()` function registrations are distinct from the `atexit()` registrations, so
1397 applications might need to call both registration functions with the same argument.

1398 The functions registered by a call to *at_quick_exit()* must return to ensure that all registered
1399 functions are called.

1400 The application should call *sysconf()* to obtain the value of {ATEXIT_MAX}, the number of
1401 functions that can be registered. There is no way for an application to tell how many
1402 functions have already been registered with *at_quick_exit()*.

1403 Since the behavior is undefined if the *quick_exit()* function is called more than once,
1404 portable applications calling *at_quick_exit()* must ensure that the *quick_exit()* function is not
1405 called when the functions registered by the *at_quick_exit()* function are called.

1406 If a function registered by the *at_quick_exit()* function is called and a portable application
1407 needs to stop further *quick_exit()* processing, it must call the *_exit()* function or the *_Exit()*
1408 function or one of the functions which cause abnormal process termination.

1409 **RATIONALE**

1410 None.

1411 **FUTURE DIRECTIONS**

1412 None.

1413 **SEE ALSO**

1414 *atexit*, *exec*, *exit*, *quick_exit*, *sysconf*

1415 XBD <stdlib.h>

1416 **CHANGE HISTORY**

1417 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1418 Ref 7.22.4.3

1419 On page 618 line 21381 section *atexit()*, change:

1420 *atexit* — register a function to run at process termination

1421 to:

1422 *atexit* — register a function to be called from *exit()* or after return from *main()*

1423 Ref 7.22.4.2 para 2, 7.22.4.3

1424 On page 618 line 21389 section *atexit()*, change:

1425 The *atexit()* function shall register the function pointed to by *func*, to be called without
1426 arguments at normal program termination. At normal program termination, all functions
1427 registered by the *atexit()* function shall be called, in the reverse order of their registration,
1428 except that a function is called after any previously registered functions that had already
1429 been called at the time it was registered. Normal termination occurs either by a call to *exit()*
1430 or a return from *main()*.

1431 to:

1432 The *atexit()* function shall register the function pointed to by *func*, to be called without
1433 arguments from *exit()*, or after return from the initial call to *main()*, or on the last thread
1434 termination. If the *exit()* function is called, it is unspecified whether a call to the *atexit()*
1435 function that does not happen before *exit()* is called will succeed.

1436 [Note to reviewers: the part about all registered functions being called in reverse order is duplicated](#)
1437 [on the exit\(\) page and is not needed here.](#)

1438 Ref 7.22.4.2 para 2

1439 On page 618 line 21405 section *atexit()*, insert a new first APPLICATION USAGE paragraph:

1440 The *atexit()* function registrations are distinct from the *at_quick_exit()* registrations, so
1441 applications might need to call both registration functions with the same argument.

1442 Ref 7.22.4.3

1443 On page 618 line 21410 section *atexit()*, change:

1444 Since the behavior is undefined if the *exit()* function is called more than once, portable
1445 applications calling *atexit()* must ensure that the *exit()* function is not called at normal
1446 process termination when all functions registered by the *atexit()* function are called.

1447 All functions registered by the *atexit()* function are called at normal process termination,
1448 which occurs by a call to the *exit()* function or a return from *main()* or on the last thread
1449 termination, when the behavior is as if the implementation called *exit()* with a zero argument
1450 at thread termination time.

1451 If, at normal process termination, a function registered by the *atexit()* function is called and a
1452 portable application needs to stop further *exit()* processing, it must call the *_exit()* function
1453 or the *_Exit()* function or one of the functions which cause abnormal process termination.

1454 to:

1455 Since the behavior is undefined if the *exit()* function is called more than once, portable
1456 applications calling *atexit()* must ensure that the *exit()* function is not called when the
1457 functions registered by the *atexit()* function are called.

1458 If a function registered by the *atexit()* function is called and a portable application needs to
1459 stop further *exit()* processing, it must call the *_exit()* function or the *_Exit()* function or one
1460 of the functions which cause abnormal process termination.

1461 Ref 7.22.4.3

1462 On page 619 line 21425 section *atexit()*, add *at_quick_exit* to the SEE ALSO section.

1463 Ref 7.16

1464 On page 624 line 21548 insert the following new *atomic_**() sections:

1465 **NAME**

1466 *atomic_compare_exchange_strong*, *atomic_compare_exchange_strong_explicit*,
1467 *atomic_compare_exchange_weak*, *atomic_compare_exchange_weak_explicit* — atomically
1468 compare and exchange the values of two objects

1469 **SYNOPSIS**

```

1470 #include <stdatomic.h>
1471 _Bool atomic_compare_exchange_strong(volatile A *object,
1472     C *expected, C desired);
1473 _Bool atomic_compare_exchange_strong_explicit(volatile A *object,
1474     C *expected, C desired, memory_order success,
1475     memory_order failure);
1476 _Bool atomic_compare_exchange_weak(volatile A *object,
1477     C *expected, C desired);
1478 _Bool atomic_compare_exchange_weak_explicit(volatile A *object,
1479     C *expected, C desired, memory_order success,
1480     memory_order failure);

```

1481 DESCRIPTION

1482 [CX] The functionality described on this reference page is aligned with the ISO C standard.
 1483 Any conflict between the requirements described here and the ISO C standard is
 1484 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1485 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
 1486 `<stdatomic.h>` header nor support these generic functions.

1487 The `atomic_compare_exchange_strong_explicit()` generic function shall atomically compare
 1488 the contents of the memory pointed to by *object* for equality with that pointed to by
 1489 *expected*, and if true, shall replace the contents of the memory pointed to by *object*
 1490 with *desired*, and if false, shall update the contents of the memory pointed to by *expected*
 1491 with that pointed to by *object*. This operation shall be an atomic read-modify-write operation
 1492 (see [xref to XBD 4.12.1]). If the comparison is true, memory shall be affected according to
 1493 the value of *success*, and if the comparison is false, memory shall be affected according to
 1494 the value of *failure*. The application shall ensure that *failure* is not
 1495 `memory_order_release` nor `memory_order_acq_rel`, and shall ensure that *failure* is
 1496 no stronger than *success*.

1497 The `atomic_compare_exchange_strong()` generic function shall be equivalent to
 1498 `atomic_compare_exchange_strong_explicit()` called with *success* and *failure* both set to
 1499 `memory_order_seq_cst`.

1500 The `atomic_compare_exchange_weak_explicit()` generic function shall be equivalent to
 1501 `atomic_compare_exchange_strong_explicit()`, except that the compare-and-exchange
 1502 operation may fail spuriously. That is, even when the contents of memory referred to by
 1503 *expected* and *object* are equal, it may return zero and store back to *expected* the same
 1504 memory contents that were originally there.

1505 The `atomic_compare_exchange_weak()` generic function shall be equivalent to
 1506 `atomic_compare_exchange_weak_explicit()` called with *success* and *failure* both set to
 1507 `memory_order_seq_cst`.

1508 RETURN VALUE

1509 These generic functions shall return the result of the comparison.

1510 ERRORS

1511 No errors are defined.

1512 EXAMPLES

1513 None.

1514 **APPLICATION USAGE**

1515 A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will
1516 be in a loop. For example:

```
1517     exp = atomic_load(&cur);  
1518     do {  
1519         des = function(exp);  
1520     } while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

1521 When a compare-and-exchange is in a loop, the weak version will yield better performance
1522 on some platforms. When a weak compare-and-exchange would require a loop and a strong
1523 one would not, the strong one is preferable.

1524 **RATIONALE**

1525 None.

1526 **FUTURE DIRECTIONS**

1527 None.

1528 **SEE ALSO**

1529 XBD Section 4.12.1, <**stdatomic.h**>

1530 **CHANGE HISTORY**

1531 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1532 **NAME**

1533 `atomic_exchange`, `atomic_exchange_explicit` — atomically exchange the value of an object

1534 **SYNOPSIS**

```
1535     #include <stdatomic.h>  
1536     C atomic_exchange(volatile A *object, C desired);  
1537     C atomic_exchange_explicit(volatile A *object,  
1538         C desired, memory_order order);
```

1539 **DESCRIPTION**

1540 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1541 Any conflict between the requirements described here and the ISO C standard is
1542 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[CX]

1543 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1544 <**stdatomic.h**> header nor support these generic functions.

1545 The `atomic_exchange_explicit()` generic function shall atomically replace the value pointed
1546 to by *object* with *desired*. This operation shall be an atomic read-modify-write operation (see
1547 [xref to XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1548 The `atomic_exchange()` generic function shall be equivalent to `atomic_exchange_explicit()`
1549 called with *order* set to `memory_order_seq_cst`.

1550 **RETURN VALUE**

1551 These generic functions shall return the value pointed to by *object* immediately before the

1552 effects.

1553 **ERRORS**

1554 No errors are defined.

1555 **EXAMPLES**

1556 None.

1557 **APPLICATION USAGE**

1558 None.

1559 **RATIONALE**

1560 None.

1561 **FUTURE DIRECTIONS**

1562 None.

1563 **SEE ALSO**

1564 XBD Section 4.12.1, <**stdatomic.h**>

1565 **CHANGE HISTORY**

1566 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1567 **NAME**

1568 `atomic_fetch_add`, `atomic_fetch_add_explicit`, `atomic_fetch_and`,
1569 `atomic_fetch_and_explicit`, `atomic_fetch_or`, `atomic_fetch_or_explicit`, `atomic_fetch_sub`,
1570 `atomic_fetch_sub_explicit`, `atomic_fetch_xor`, `atomic_fetch_xor_explicit` — atomically
1571 replace the value of an object with the result of a computation

1572 **SYNOPSIS**

```
1573 #include <stdatomic.h>
1574 C atomic_fetch_add(volatile A *object, M operand);
1575 C atomic_fetch_add_explicit(volatile A *object, M operand,
1576 memory_order order);
1577 C atomic_fetch_and(volatile A *object, M operand);
1578 C atomic_fetch_and_explicit(volatile A *object, M operand,
1579 memory_order order);
1580 C atomic_fetch_or(volatile A *object, M operand);
1581 C atomic_fetch_or_explicit(volatile A *object, M operand,
1582 memory_order order);
1583 C atomic_fetch_sub(volatile A *object, M operand);
1584 C atomic_fetch_sub_explicit(volatile A *object, M operand,
1585 memory_order order);
1586 C atomic_fetch_xor(volatile A *object, M operand);
1587 C atomic_fetch_xor_explicit(volatile A *object, M operand,
1588 memory_order order);
```

1589 **DESCRIPTION**

1590 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1591 Any conflict between the requirements described here and the ISO C standard is
1592 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1593 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the

1594 <**stdatomic.h**> header nor support these generic functions.

1595 The *atomic_fetch_add_explicit()* generic function shall atomically replace the value pointed
1596 to by *object* with the result of adding *operand* to this value. This operation shall be an
1597 atomic read-modify-write operation (see [xref to XBD 4.12.1]). Memory shall be affected
1598 according to the value of *order*.

1599 The *atomic_fetch_add()* generic function shall be equivalent to *atomic_fetch_add_explicit()*
1600 called with *order* set to *memory_order_seq_cst*.

1601 The other *atomic_fetch_**() generic functions shall be equivalent to
1602 *atomic_fetch_add_explicit()* if their name ends with *explicit*, or to *atomic_fetch_add()* if it
1603 does not, respectively, except that they perform the computation indicated in their name,
1604 instead of addition:

1605 *sub* subtraction
1606 *or* bitwise inclusive OR
1607 *xor* bitwise exclusive OR
1608 *and* bitwise AND

1609 For addition and subtraction, the application shall ensure that *A* is an atomic integer type or
1610 an atomic pointer type and is not **atomic_bool**. For the other operations, the application
1611 shall ensure that *A* is an atomic integer type and is not **atomic_bool**.

1612 For signed integer types, the computation shall silently wrap around on overflow; there are
1613 no undefined results. For pointer types, the result can be an undefined address, but the
1614 computations otherwise have no undefined behavior.

1615 **RETURN VALUE**

1616 These generic functions shall return the value pointed to by *object* immediately before the
1617 effects.

1618 **ERRORS**

1619 No errors are defined.

1620 **EXAMPLES**

1621 None.

1622 **APPLICATION USAGE**

1623 The operation of these generic functions is nearly equivalent to the operation of the
1624 corresponding compound assignment operators +=, -=, etc. The only differences are that the
1625 compound assignment operators are not guaranteed to operate atomically, and the value
1626 yielded by a compound assignment operator is the updated value of the object, whereas the
1627 value returned by these generic functions is the previous value of the atomic object.

1628 **RATIONALE**

1629 None.

1630 **FUTURE DIRECTIONS**

1631 None.

1632 **SEE ALSO**

1633 XBD Section 4.12.1, <stdatomic.h>

1634 **CHANGE HISTORY**

1635 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1636 **NAME**

1637 `atomic_flag_clear`, `atomic_flag_clear_explicit` — clear an atomic flag

1638 **SYNOPSIS**

```
1639 #include <stdatomic.h>
1640 void atomic_flag_clear(volatile atomic_flag *object);
1641 void atomic_flag_clear_explicit(
1642     volatile atomic_flag *object, memory_order order);
```

1643 **DESCRIPTION**

1644 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1645 Any conflict between the requirements described here and the ISO C standard is
1646 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1647 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1648 `<stdatomic.h>` header nor support these functions.

1649 The `atomic_flag_clear_explicit()` function shall atomically place the atomic flag pointed to
1650 by `object` into the clear state. Memory shall be affected according to the value of `order`,
1651 which the application shall ensure is not `memory_order_acquire` nor
1652 `memory_order_acq_rel`.

1653 The `atomic_flag_clear()` function shall be equivalent to `atomic_flag_clear_explicit()` called
1654 with `order` set to `memory_order_seq_cst`.

1655 **RETURN VALUE**

1656 These functions shall not return a value.

1657 **ERRORS**

1658 No errors are defined.

1659 **EXAMPLES**

1660 None.

1661 **APPLICATION USAGE**

1662 None.

1663 **RATIONALE**

1664 None.

1665 **FUTURE DIRECTIONS**

1666 None.

1667 **SEE ALSO**

1668 XBD Section 4.12.1, <stdatomic.h>

1669 **CHANGE HISTORY**

1670 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1671 **NAME**

1672 `atomic_flag_test_and_set`, `atomic_flag_test_and_set_explicit` — test and set an atomic flag

1673 **SYNOPSIS**

```
1674 #include <stdatomic.h>
1675 _Bool atomic_flag_test_and_set(volatile atomic_flag *object);
1676 _Bool atomic_flag_test_and_set_explicit(
1677     volatile atomic_flag *object, memory_order order);
```

1678 **DESCRIPTION**

1679 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1680 Any conflict between the requirements described here and the ISO C standard is
1681 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1682 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1683 `<stdatomic.h>` header nor support these functions.

1684 The `atomic_flag_test_and_set_explicit()` function shall atomically place the atomic flag
1685 pointed to by `object` into the set state and return the value corresponding to the immediately
1686 preceding state. This operation shall be an atomic read-modify-write operation (see [xref to
1687 XBD 4.12.1]). Memory shall be affected according to the value of `order`.

1688 The `atomic_flag_test_and_set()` function shall be equivalent to
1689 `atomic_flag_test_and_set_explicit()` called with `order` set to `memory_order_seq_cst`.

1690 **RETURN VALUE**

1691 These functions shall return the value that corresponds to the state of the atomic flag
1692 immediately before the effects. The return value `true` shall correspond to the set state and the
1693 return value `false` shall correspond to the clear state.

1694 **ERRORS**

1695 No errors are defined.

1696 **EXAMPLES**

1697 None.

1698 **APPLICATION USAGE**

1699 None.

1700 **RATIONALE**

1701 None.

1702 **FUTURE DIRECTIONS**

1703 None.

1704 **SEE ALSO**

1705 XBD Section 4.12.1, `<stdatomic.h>`

1706 **CHANGE HISTORY**

1707 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1708 **NAME**

1709 `atomic_init` — initialize an atomic object

1710 **SYNOPSIS**

```
1711 #include <stdatomic.h>
1712 void atomic_init(volatile A *obj, C value);
```

1713 **DESCRIPTION**

1714 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1715 Any conflict between the requirements described here and the ISO C standard is
1716 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1717 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1718 `<stdatomic.h>` header nor support this generic function.

1719 The `atomic_init()` generic function shall initialize the atomic object pointed to by `obj` to the
1720 value `value`, while also initializing any additional state that the implementation might need
1721 to carry for the atomic object.

1722 Although this function initializes an atomic object, it does not avoid data races; concurrent
1723 access to the variable being initialized, even via an atomic operation, constitutes a data race.

1724 **RETURN VALUE**

1725 The `atomic_init()` generic function shall not return a value.

1726 **ERRORS**

1727 No errors are defined.

1728 **EXAMPLES**

```
1729 atomic_int guide;
1730 atomic_init(&guide, 42);
```

1731 **APPLICATION USAGE**

1732 None.

1733 **RATIONALE**

1734 None.

1735 **FUTURE DIRECTIONS**

1736 None.

1737 **SEE ALSO**

1738 XBD `<stdatomic.h>`

1739 **CHANGE HISTORY**

1740 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1741 **NAME**

1742 `atomic_is_lock_free` — indicate whether or not atomic operations are lock-free

1743 **SYNOPSIS**

```
1744     #include <stdatomic.h>
1745     _Bool atomic_is_lock_free(const volatile A *obj);
```

1746 **DESCRIPTION**

1747 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1748 Any conflict between the requirements described here and the ISO C standard is
1749 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1750 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1751 `<stdatomic.h>` header nor support this generic function.

1752 The `atomic_is_lock_free()` generic function shall indicate whether or not atomic operations
1753 on objects of the type pointed to by `obj` are lock-free; `obj` can be a null pointer.

1754 **RETURN VALUE**

1755 The `atomic_is_lock_free()` generic function shall return a non-zero value if and only if
1756 atomic operations on objects of the type pointed to by `obj` are lock-free. During the lifetime
1757 of the calling process, the result of the lock-free query shall be consistent for all pointers of
1758 the same type.

1759 **ERRORS**

1760 No errors are defined.

1761 **EXAMPLES**

1762 None.

1763 **APPLICATION USAGE**

1764 None.

1765 **RATIONALE**

1766 Operations that are lock-free should also be address-free. That is, atomic operations on the
1767 same memory location via two different addresses will communicate atomically. The
1768 implementation should not depend on any per-process state. This restriction enables
1769 communication via memory mapped into a process more than once and memory shared
1770 between two processes.

1771 **FUTURE DIRECTIONS**

1772 None.

1773 **SEE ALSO**

1774 XBD `<stdatomic.h>`

1775 **CHANGE HISTORY**

1776 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1777 **NAME**

1778 `atomic_load`, `atomic_load_explicit` — atomically obtain the value of an object

1779 **SYNOPSIS**

```
1780     #include <stdatomic.h>
1781     C atomic_load(const volatile A *object);
```

1782 **C** atomic_load_explicit(const volatile **A** *object,
1783 memory_order order);

1784 **DESCRIPTION**

1785 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1786 Any conflict between the requirements described here and the ISO C standard is
1787 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1788 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1789 `<stdatomic.h>` header nor support these generic functions.

1790 The `atomic_load_explicit()` generic function shall atomically obtain the value pointed to by
1791 `object`. Memory shall be affected according to the value of `order`, which the application shall
1792 ensure is not `memory_order_release` nor `memory_order_acq_rel`.

1793 The `atomic_load()` generic function shall be equivalent to `atomic_load_explicit()` called with
1794 `order` set to `memory_order_seq_cst`.

1795 **RETURN VALUE**

1796 These generic functions shall return the value pointed to by `object`.

1797 **ERRORS**

1798 No errors are defined.

1799 **EXAMPLES**

1800 None.

1801 **APPLICATION USAGE**

1802 None.

1803 **RATIONALE**

1804 None.

1805 **FUTURE DIRECTIONS**

1806 None.

1807 **SEE ALSO**

1808 XBD Section 4.12.1, `<stdatomic.h>`

1809 **CHANGE HISTORY**

1810 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1811 **NAME**

1812 `atomic_signal_fence`, `atomic_thread_fence` — fence operations

1813 **SYNOPSIS**

```
1814 #include <stdatomic.h>  
1815 void atomic_signal_fence(memory_order order);  
1816 void atomic_thread_fence(memory_order order);
```

1817 **DESCRIPTION**

1818 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1819 Any conflict between the requirements described here and the ISO C standard is
1820 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1821 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1822 `<stdatomic.h>` header nor support these functions.

1823 The `atomic_signal_fence()` and `atomic_thread_fence()` functions provide synchronization
1824 primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A
1825 fence with acquire semantics is called an *acquire fence*; a fence with release semantics is
1826 called a *release fence*.

1827 A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X*
1828 and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X*
1829 modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* or a value written
1830 by any side effect in the hypothetical release sequence *X* would head if it were a release
1831 operation.

1832 A release fence *A* synchronizes with an atomic operation *B* that performs an acquire
1833 operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is
1834 sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by
1835 any side effect in the hypothetical release sequence *X* would head if it were a release
1836 operation.

1837 An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with
1838 an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced
1839 before *B* and reads the value written by *A* or a value written by any side effect in the release
1840 sequence headed by *A*.

1841 Depending on the value of *order*, the operation performed by `atomic_thread_fence()` shall:

- 1842 • have no effects, if *order* is equal to `memory_order_relaxed`;
- 1843 • be an acquire fence, if *order* is equal to `memory_order_acquire` or
1844 `memory_order_consume`;
- 1845 • be a release fence, if *order* is equal to `memory_order_release`;
- 1846 • be both an acquire fence and a release fence, if *order* is equal to
1847 `memory_order_acq_rel`;
- 1848 • be a sequentially consistent acquire and release fence, if *order* is equal to
1849 `memory_order_seq_cst`.

1850 The `atomic_signal_fence()` function shall be equivalent to `atomic_thread_fence()`, except
1851 that the resulting ordering constraints shall be established only between a thread and a signal
1852 handler executed in the same thread.

1853 RETURN VALUE

1854 These functions shall not return a value.

1855 ERRORS

1856 No errors are defined.

1857 **EXAMPLES**

1858 None.

1859 **APPLICATION USAGE**

1860 The *atomic_signal_fence()* function can be used to specify the order in which actions
1861 performed by the thread become visible to the signal handler. Implementation reorderings of
1862 loads and stores are inhibited in the same way as with *atomic_thread_fence()*, but the
1863 hardware fence instructions that *atomic_thread_fence()* would have inserted are not
1864 emitted.

1865 **RATIONALE**

1866 None.

1867 **FUTURE DIRECTIONS**

1868 None.

1869 **SEE ALSO**

1870 XBD Section 4.12.1, <**stdatomic.h**>

1871 **CHANGE HISTORY**

1872 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1873 **NAME**

1874 *atomic_store*, *atomic_store_explicit* — atomically store a value in an object

1875 **SYNOPSIS**

```
1876       #include <stdatomic.h>  
1877       void atomic_store(volatile A *object, C desired);  
1878       void atomic_store_explicit(volatile A *object, C desired,  
1879       memory_order order);
```

1880 **DESCRIPTION**

1881 [**CX**] The functionality described on this reference page is aligned with the ISO C standard.
1882 Any conflict between the requirements described here and the ISO C standard is
1883 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[**CX**]

1884 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1885 <**stdatomic.h**> header nor support these generic functions.

1886 The *atomic_store_explicit()* generic function shall atomically replace the value pointed to by
1887 *object* with the value of *desired*. Memory shall be affected according to the value of *order*,
1888 which the application shall ensure is not `memory_order_acquire`,
1889 `memory_order_consume`, nor `memory_order_acq_rel`.

1890 The *atomic_store()* generic function shall be equivalent to *atomic_store_explicit()* called
1891 with *order* set to `memory_order_seq_cst`.

1892 **RETURN VALUE**

1893 These generic functions shall not return a value.

1894 **ERRORS**

1895 No errors are defined.

1896 **EXAMPLES**

1897 None.

1898 **APPLICATION USAGE**

1899 None.

1900 **RATIONALE**

1901 None.

1902 **FUTURE DIRECTIONS**

1903 None.

1904 **SEE ALSO**

1905 XBD Section 4.12.1, <stdatomic.h>

1906 **CHANGE HISTORY**

1907 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1908 Ref 7.28.1, 7.1.4 para 5

1909 On page 633 line 21891 insert a new `c16rtomb()` section:

1910 **NAME**

1911 `c16rtomb`, `c32rtomb` — convert a Unicode character code to a character (restartable)

1912 **SYNOPSIS**

1913 `#include <uchar.h>`

1914 `size_t c16rtomb(char *restrict s, char16_t c16,`

1915 `mbstate_t *restrict ps);`

1916 `size_t c32rtomb(char *restrict s, char32_t c32,`

1917 `mbstate_t *restrict ps);`

1918 **DESCRIPTION**

1919 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1920 Any conflict between the requirements described here and the ISO C standard is

1921 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1922 If *s* is a null pointer, the `c16rtomb()` function shall be equivalent to the call:

1923 `c16rtomb(buf, L'\0', ps)`

1924 where *buf* is an internal buffer.

1925 If *s* is not a null pointer, the `c16rtomb()` function shall determine the number of bytes needed

1926 to represent the character that corresponds to the wide character given by *c16* (including any

1927 shift sequences), and store the resulting bytes in the array whose first element is pointed to

1928 by *s*. At most {MB_CUR_MAX} bytes shall be stored. If *c16* is a null wide character, a null

1929 byte shall be stored, preceded by any shift sequence needed to restore the initial shift state;

1930 the resulting state described shall be the initial conversion state.

1931 If *ps* is a null pointer, the `c16rtomb()` function shall use its own internal `mbstate_t` object,

1932 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
1933 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current
1934 conversion state of the associated character sequence.

1935 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

1936 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

1937 The *c32rtomb()* function shall behave the same way as *c16rtomb()* except that the second
1938 parameter shall be an object of type **char32_t** instead of **char16_t**. References to *c16* in the
1939 above description shall apply as if they were *c32* when they are being read as describing
1940 *c32rtomb()*.

1941 If called with a null *ps* argument, the *c16rtomb()* function need not be thread-safe; however,
1942 such calls shall avoid data races with calls to *c16rtomb()* with a non-null argument and with
1943 calls to all other functions.

1944 If called with a null *ps* argument, the *c32rtomb()* function need not be thread-safe; however,
1945 such calls shall avoid data races with calls to *c32rtomb()* with a non-null argument and with
1946 calls to all other functions.

1947 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
1948 calls *c16rtomb()* or *c32rtomb()* with a null pointer for *ps*.

1949 **RETURN VALUE**

1950 These functions shall return the number of bytes stored in the array object (including any
1951 shift sequences). When *c16* or *c32* is not a valid wide character, an encoding error shall
1952 occur. In this case, the function shall store the value of the macro [EILSEQ] in *errno* and
1953 shall return (**size_t**)-1; the conversion state is unspecified.

1954 **ERRORS**

1955 These function shall fail if:

1956 [EILSEQ] An invalid wide-character code is detected.

1957 These functions may fail if:

1958 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

1959 **EXAMPLES**

1960 None.

1961 **APPLICATION USAGE**

1962 None.

1963 **RATIONALE**

1964 None.

1965 **FUTURE DIRECTIONS**

1966 None.

1967 **SEE ALSO**

1968 *mbrtoc16*

- 1969 XBD <uchar.h>
- 1970 **CHANGE HISTORY**
- 1971 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.
- 1972 Ref G.6 para 6, F.10.4.3, F.10.4.2, F.10 para 11
- 1973 On page 633 line 21905 section `cabs()`, add:
- 1974 [MXC]`cabs(x + iy)`, `cabs(y + ix)`, and `cabs(x - iy)` shall return exactly the same value.
- 1975 If z is $\pm 0 \pm i0$, `+0` shall be returned.
- 1976 If the real or imaginary part of z is $\pm\text{Inf}$, `+Inf` shall be returned, even if the other part is NaN.
- 1977 If the real or imaginary part of z is NaN and the other part is not $\pm\text{Inf}$, NaN shall be returned.
- 1978 [/MXC]
- 1979 Ref G.6.1.1
- 1980 On page 634 line 21935 section `cacos()`, add:
- 1981 [MXC]`cacos(conj(z))`, `cacosf(conjf(z))` and `cacosl(conjl(z))` shall return exactly the same
- 1982 value as `conj(cacos(z))`, `conjf(cacosf(z))` and `conjl(cacosl(z))`, respectively, including for the
- 1983 special values of z below.
- 1984 If z is $\pm 0 + i0$, $\pi/2 - i0$ shall be returned.
- 1985 If z is $\pm 0 + i\text{NaN}$, $\pi/2 + i\text{NaN}$ shall be returned.
- 1986 If z is $x + i\text{Inf}$ where x is finite, $\pi/2 - i\text{Inf}$ shall be returned.
- 1987 If z is $x + i\text{NaN}$ where x is non-zero and finite, `NaN + iNaN` shall be returned and the invalid
- 1988 floating-point exception may be raised.
- 1989 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $\pi - i\text{Inf}$ shall be returned.
- 1990 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, `+0 - iInf` shall be returned.
- 1991 If z is $-\text{Inf} + i\text{Inf}$, $3\pi/4 - i\text{Inf}$ shall be returned.
- 1992 If z is $+\text{Inf} + i\text{Inf}$, $\pi/4 - i\text{Inf}$ shall be returned.
- 1993 If z is $\pm\text{Inf} + i\text{NaN}$, `NaN \pm iInf` shall be returned; the sign of the imaginary part of the result
- 1994 is unspecified.
- 1995 If z is `NaN + iy` where y is finite, `NaN + iNaN` shall be returned and the invalid floating-
- 1996 point exception may be raised.
- 1997 If z is `NaN + iInf`, `NaN - iInf` shall be returned.
- 1998 If z is `NaN + iNaN`, `NaN - iNaN` shall be returned.[/MXC]

1999 Ref G.6.2.1
2000 On page 635 line 21966 section `cacosh()`, add:

2001 [MXC]`cacosh(conj(z))`, `cacoshf(conjf(z))` and `cacoshl(conjl(z))` shall return exactly the same
2002 value as `conj(cacosh(z))`, `conjf(cacoshf(z))` and `conjl(cacoshl(z))`, respectively, including for
2003 the special values of z below.

2004 If z is $\pm 0 + i0$, $+0 + i\pi/2$ shall be returned.

2005 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2006 If z is $0 + i\text{NaN}$, $\text{NaN} \pm i\pi/2$ shall be returned; the sign of the imaginary part of the result is
2007 unspecified.

2008 If z is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2009 floating-point exception may be raised.

2010 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2011 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2012 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

2013 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2014 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2015 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2016 point exception may be raised.

2017 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2018 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2019 Ref 7.26.2.1
2020 On page 637 line 21989 insert the following new `call_once()` section:

2021 **NAME**
2022 `call_once` — dynamic package initialization

2023 **SYNOPSIS**
2024 `#include <threads.h>`

2025 `void call_once(once_flag *flag, void (*init_routine)(void));`
2026 `once_flag flag = ONCE_FLAG_INIT;`

2027 **DESCRIPTION**
2028 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2029 Any conflict between the requirements described here and the ISO C standard is
2030 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2031 The `call_once()` function shall use the **once_flag** pointed to by `flag` to ensure that
2032 `init_routine` is called exactly once, the first time the `call_once()` function is called with that

2033 value of *flag*. Completion of an effective call to the *call_once()* function shall synchronize
2034 with all subsequent calls to the *call_once()* function with the same value of *flag*.

2035 [CX]The *call_once()* function is not a cancellation point. However, if *init_routine* is a
2036 cancellation point and is canceled, the effect on *flag* shall be as if *call_once()* was never
2037 called.

2038 If the call to *init_routine* is terminated by a call to *longjmp()* or *siglongjmp()*, the behavior is
2039 undefined.

2040 The behavior of *call_once()* is undefined if *flag* has automatic storage duration or is not
2041 initialized by `ONCE_FLAG_INIT`.

2042 The *call_once()* function shall not be affected if the calling thread executes a signal handler
2043 during the call.[/CX]

2044 **RETURN VALUE**

2045 The *call_once()* function shall not return a value.

2046 **ERRORS**

2047 No errors are defined.

2048 **EXAMPLES**

2049 None.

2050 **APPLICATION USAGE**

2051 If *init_routine* recursively calls *call_once()* with the same *flag*, the recursive call will not call
2052 the specified *init_routine*, and thus the specified *init_routine* will not complete, and thus the
2053 recursive call to *call_once()* will not return. Use of *longjmp()* or *siglongjmp()* within an
2054 *init_routine* to jump to a point outside of *init_routine* prevents *init_routine* from returning.

2055 **RATIONALE**

2056 For dynamic library initialization in a multi-threaded process, if an initialization flag is used
2057 the flag needs to be protected against modification by multiple threads simultaneously
2058 calling into the library. This can be done by using a statically-initialized mutex. However,
2059 the better solution is to use *call_once()* or *pthread_once()* which are designed for exactly
2060 this purpose, for example:

```

2061 #include <threads.h>
2062 static once_flag random_is_initialized = ONCE_FLAG_INIT;
2063 extern void initialize_random(void);

2064 int random_function()
2065 {
2066     call_once(&random_is_initialized, initialize_random);
2067     ...
2068     /* Operations performed after initialization. */
2069 }
```

2070 The *call_once()* function is not affected by signal handlers for the reasons stated in [xref to
2071 XRAT B.2.3].

2072 **FUTURE DIRECTIONS**

2073 None.

2074 **SEE ALSO**

2075 *pthread_once*

2076 XBD Section 4.12.2, <**threads.h**>

2077 **CHANGE HISTORY**

2078 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2079 Ref 7.22.3 para 1

2080 On page 637 line 22002 section `calloc()`, change:

2081 a pointer to any type of object

2082 to:

2083 a pointer to any type of object with a fundamental alignment requirement

2084 Ref 7.22.3 para 1

2085 On page 637 line 22007 section `calloc()`, change:

2086 either a null pointer shall be returned, or ...

2087 to:

2088 either a null pointer shall be returned to indicate an error, or ...

2089 Ref 7.22.3 para 2

2090 On page 637 line 22008 section `calloc()`, add a new paragraph:

2091 For purposes of determining the existence of a data race, `calloc()` shall behave as though it
2092 accessed only memory locations accessible through its arguments and not other static
2093 duration storage. The function may, however, visibly modify the storage that it allocates.
2094 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `[ADV]posix_memalign()`, `[/ADV]` and
2095 `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total
2096 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
2097 next allocation (if any) in this order.

2098 Ref 7.22.3.1

2099 On page 637 line 22029 section `calloc()`, add `aligned_alloc` to the SEE ALSO section.

2100 Ref G.6 para 6, F.10.1.4, F.10 para 11

2101 On page 639 line 22055 section `carg()`, add:

2102 [MXC]If z is $-0 \pm i0$, $\pm\pi$ shall be returned.

2103 If z is $+0 \pm i0$, ± 0 shall be returned.

2104 If z is $x \pm i0$ where x is negative, $\pm\pi$ shall be returned.

- 2105 If z is $x \pm i0$ where x is positive, ± 0 shall be returned.
- 2106 If z is $\pm 0 + iy$ where y is negative, $-\pi/2$ shall be returned.
- 2107 If z is $\pm 0 + iy$ where y is positive, $\pi/2$ shall be returned.
- 2108 If z is $-\text{Inf} \pm iy$ where y is positive and finite, $\pm\pi$ shall be returned.
- 2109 If z is $+\text{Inf} \pm iy$ where y is positive and finite, ± 0 shall be returned.
- 2110 If z is $x \pm i\text{Inf}$ where x is finite, $\pm\pi/2$ shall be returned.
- 2111 If z is $-\text{Inf} \pm i\text{Inf}$, $\pm 3\pi/4$ shall be returned.
- 2112 If z is $+\text{Inf} \pm i\text{Inf}$, $\pm\pi/4$ shall be returned.
- 2113 If the real or imaginary part of z is NaN, NaN shall be returned.[/MXC]
- 2114 Ref G.6 para 7, G.6.2.2
2115 On page 640 line 22086 section `casin()`, add:
- 2116 [MXC]`casin(conj(iz))`, `casinf(conjf(iz))` and `casinl(conjl(iz))` shall return exactly the same
2117 value as `conj(casin(iz))`, `conjf(casinf(iz))` and `conjl(casinl(iz))`, respectively, and `casin(-iz)`,
2118 `casinf(-iz)` and `casinl(-iz)` shall return exactly the same value as `-casin(iz)`, `-casinf(iz)` and
2119 `-casinl(iz)`, respectively, including for the special values of iz below.
- 2120 If iz is $+0 + i0$, $-i(0 + i0)$ shall be returned.
- 2121 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i(+\text{Inf} + i\pi/2)$ shall be returned.
- 2122 If iz is $x + i\text{NaN}$ where x is finite, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the invalid
2123 floating-point exception may be raised.
- 2124 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i(+\text{Inf} + i0)$ shall be returned.
- 2125 If iz is $+\text{Inf} + i\text{Inf}$, $-i(+\text{Inf} + i\pi/4)$ shall be returned.
- 2126 If iz is $+\text{Inf} + i\text{NaN}$, $-i(+\text{Inf} + i\text{NaN})$ shall be returned.
- 2127 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.
- 2128 If iz is $\text{NaN} + iy$ where y is non-zero and finite, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
2129 invalid floating-point exception may be raised.
- 2130 If iz is $\text{NaN} + i\text{Inf}$, $-i(\pm\text{Inf} + i\text{NaN})$ shall be returned; the sign of the imaginary part of the
2131 result is unspecified.
- 2132 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2133 Ref G.6 para 7
2134 On page 640 line 22094 section `casin()`, change RATIONALE from:

2135 None.

2136 to:

2137 The MXC special cases for *casin()* are derived from those for *casinh()* by applying the
2138 formula $casin(z) = -i casinh(iz)$.

2139 Ref G.6.2.2
2140 On page 641 line 22118 section *casinh()*, add:

2141 [MXC]*casinh(conj(z))*, *casinhf(conjf(z))* and *casinhl(conjl(z))* shall return exactly the same
2142 value as *conj(casinh(z))*, *conjf(casinhf(z))* and *conjl(casinhl(z))*, respectively, and *casinh(-z)*,
2143 *casinhf(-z)* and *casinhl(-z)* shall return exactly the same value as $-casinh(z)$, $-casinhf(z)$
2144 and $-casinhl(z)$, respectively, including for the special values of *z* below.

2145 If *z* is $+0 + i0$, $0 + i0$ shall be returned.

2146 If *z* is $x + iInf$ where *x* is positive-signed and finite, $+Inf + i\pi/2$ shall be returned.

2147 If *z* is $x + iNaN$ where *x* is finite, $NaN + iNaN$ shall be returned and the invalid floating-
2148 point exception may be raised.

2149 If *z* is $+Inf + iy$ where *y* is positive-signed and finite, $+Inf + i0$ shall be returned.

2150 If *z* is $+Inf + iInf$, $+Inf + i\pi/4$ shall be returned.

2151 If *z* is $+Inf + iNaN$, $+Inf + iNaN$ shall be returned.

2152 If *z* is $NaN + i0$, $NaN + i0$ shall be returned.

2153 If *z* is $NaN + iy$ where *y* is non-zero and finite, $NaN + iNaN$ shall be returned and the invalid
2154 floating-point exception may be raised.

2155 If *z* is $NaN + iInf$, $\pm Inf + iNaN$ shall be returned; the sign of the real part of the result is
2156 unspecified.

2157 If *z* is $NaN + iNaN$, $NaN + iNaN$ shall be returned.[/MXC]

2158 Ref G.6 para 7, G.6.2.3
2159 On page 643 line 22157 section *catan*, add:

2160 [MXC]*catan(conj(iz))*, *catanf(conjf(iz))* and *catanl(conjl(iz))* shall return exactly the same
2161 value as *conj(catan(iz))*, *conjf(catanf(iz))* and *conjl(catanl(iz))*, respectively, and *catan(-iz)*,
2162 *catanf(-iz)* and *catanl(-iz)* shall return exactly the same value as $-catan(iz)$, $-catanf(iz)$ and
2163 $-catanl(iz)$, respectively, including for the special values of *iz* below.

2164 If *iz* is $+0 + i0$, $-i (+0 + i0)$ shall be returned.

2165 If *iz* is $+0 + iNaN$, $-i (+0 + iNaN)$ shall be returned.

2166 If *iz* is $+1 + i0$, $-i (+Inf + i0)$ shall be returned and the divide-by-zero floating-point

2167 exception shall be raised.

2168 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.

2169 If iz is $x + i\text{NaN}$ where x is non-zero and finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
2170 invalid floating-point exception may be raised.

2171 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.

2172 If iz is $+\text{Inf} + i\text{Inf}$, $-i (+0 + i\pi/2)$ shall be returned.

2173 If iz is $+\text{Inf} + i\text{NaN}$, $-i (+0 + i\text{NaN})$ shall be returned.

2174 If iz is $\text{NaN} + iy$ where y is finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the invalid
2175 floating-point exception may be raised.

2176 If iz is $\text{NaN} + i\text{Inf}$, $-i (\pm 0 + i\pi/2)$ shall be returned; the sign of the imaginary part of the
2177 result is unspecified.

2178 If iz is $\text{NaN} + i\text{NaN}$, $-i (\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2179 Ref G.6 para 7
2180 On page 643 line 22165 section `catan()`, change RATIONALE from:

2181 None.

2182 to:

2183 The MXC special cases for `catan()` are derived from those for `catanh()` by applying the
2184 formula $\text{catan}(z) = -i \text{catanh}(iz)$.

2185 Ref G.6.2.3
2186 On page 644 line 22189 section `catanh`, add:

2187 [MXC]`catanh(conj(z))`, `catanhf(conjf(z))` and `catanhl(conjl(z))` shall return exactly the same
2188 value as `conj(catanh(z))`, `conjf(catanhf(z))` and `conjl(catanhl(z))`, respectively, and
2189 `catanh(-z)`, `catanhf(-z)` and `catanhl(-z)` shall return exactly the same value as $-\text{catanh}(z)$,
2190 $-\text{catanhf}(z)$ and $-\text{catanhl}(z)$, respectively, including for the special values of z below.

2191 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2192 If z is $+0 + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.

2193 If z is $+1 + i0$, $+\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2194 shall be raised.

2195 If z is $x + i\text{Inf}$ where x is positive-signed and finite, $+0 + i\pi/2$ shall be returned.

2196 If z is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2197 floating-point exception may be raised.

2198 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\pi/2$ shall be returned.

- 2199 If z is $+\text{Inf} + i\text{Inf}$, $+0 + i\pi/2$ shall be returned.
- 2200 If z is $+\text{Inf} + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.
- 2201 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2202 point exception may be raised.
- 2203 If z is $\text{NaN} + i\text{Inf}$, $\pm 0 + i\pi/2$ shall be returned; the sign of the real part of the result is
2204 unspecified.
- 2205 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2206 Ref G.6 para 7, G.6.2.4
2207 On page 652 line 22426 section `ccos()`, add:
- 2208 [MXC]`ccos(conj(iz))`, `ccosf(conj(iz))` and `ccosl(conj(iz))` shall return exactly the same value
2209 as `conj(ccos(iz))`, `conj(ccosf(iz))` and `conjl(ccosl(iz))`, respectively, and `ccos(-iz)`, `ccosf(-iz)`
2210 and `ccosl(-iz)` shall return exactly the same value as `ccos(iz)`, `ccosf(iz)` and `ccosl(iz)`,
2211 respectively, including for the special values of iz below.
- 2212 If iz is $+0 + i0$, $1 + i0$ shall be returned.
- 2213 If iz is $+0 + i\text{Inf}$, $\text{NaN} \pm i0$ shall be returned and the invalid floating-point exception shall be
2214 raised; the sign of the imaginary part of the result is unspecified.
- 2215 If iz is $+0 + i\text{NaN}$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2216 unspecified.
- 2217 If iz is $x + i\text{Inf}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2218 floating-point exception shall be raised.
- 2219 If iz is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the
2220 invalid floating-point exception may be raised.
- 2221 If iz is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.
- 2222 If iz is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.
- 2223 If iz is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2224 shall be raised; the sign of the real part of the result is unspecified.
- 2225 If iz is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2226 If iz is $\text{NaN} + i0$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2227 unspecified.
- 2228 If iz is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2229 invalid floating-point exception may be raised.
- 2230 If iz is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2231 Ref G.6 para 7
2232 On page 652 line 22434 section `ccos()`, change RATIONALE from:

2233 None.

2234 to:

2235 The MXC special cases for `ccos()` are derived from those for `ccosh()` by applying the
2236 formula $ccos(z) = ccosh(iz)$.

2237 Ref G.6.2.4
2238 On page 653 line 22455 section `ccosh()`, add:

2239 [MXC]`ccosh(conj(z))`, `ccoshf(conjf(z))` and `ccoshl(conjl(z))` shall return exactly the same
2240 value as `conj(ccosh(z))`, `conjf(ccoshf(z))` and `conjl(ccoshl(z))`, respectively, and `ccosh(-z)`,
2241 `ccoshf(-z)` and `ccoshl(-z)` shall return exactly the same value as `ccosh(z)`, `ccoshf(z)` and
2242 `ccoshl(z)`, respectively, including for the special values of z below.

2243 If z is $+0 + i0$, $1 + i0$ shall be returned.

2244 If z is $+0 + i\text{Inf}$, $\text{NaN} \pm i0$ shall be returned and the invalid floating-point exception shall be
2245 raised; the sign of the imaginary part of the result is unspecified.

2246 If z is $+0 + i\text{NaN}$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2247 unspecified.

2248 If z is $x + i\text{Inf}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2249 floating-point exception shall be raised.

2250 If z is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2251 floating-point exception may be raised.

2252 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2253 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2254 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2255 shall be raised; the sign of the real part of the result is unspecified.

2256 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2257 If z is $\text{NaN} + i0$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2258 unspecified.

2259 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2260 invalid floating-point exception may be raised.

2261 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2262 Ref F.10.6.1 para 4
2263 On page 655 line 22489 section `ceil()`, add a new paragraph:

2264 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2265 arguments.[/MX]

2266 Ref F.10.6.1 para 2
2267 On page 655 line 22491 section `ceil()`, change:

2268 [MX]The result shall have the same sign as x .[/MX]

2269 to:

2270 [MX]The returned value shall be independent of the current rounding direction mode and
2271 shall have the same sign as x .[/MX]

2272 Ref F.10.6.1 para 4
2273 On page 655 line 22504 section `ceil()`, delete from APPLICATION USAGE:

2274 These functions may raise the inexact floating-point exception if the result differs in value
2275 from the argument.

2276 Ref G.6.3.1
2277 On page 657 line 22539 section `cexp()`, add:

2278 [MXC] $cexp(\text{conj}(z))$, $cexpf(\text{conjf}(z))$ and $cexpl(\text{conjl}(z))$ shall return exactly the same value
2279 as $\text{conj}(cexp(z))$, $\text{conjf}(cexpf(z))$ and $\text{conjl}(cexpl(z))$, respectively, including for the special
2280 values of z below.

2281 If z is $\pm 0 + i0$, $1 + i0$ shall be returned.

2282 If z is $x + i\text{Inf}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-point
2283 exception shall be raised.

2284 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2285 point exception may be raised.

2286 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2287 If z is $-\text{Inf} + iy$ where y is finite, $+0 (\cos(y) + i\sin(y))$ shall be returned.

2288 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2289 If z is $-\text{Inf} + i\text{Inf}$, $\pm 0 \pm i0$ shall be returned; the signs of the real and imaginary parts of the
2290 result are unspecified.

2291 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2292 shall be raised; the sign of the real part of the result is unspecified.

2293 If z is $-\text{Inf} + i\text{NaN}$, $\pm 0 \pm i0$ shall be returned; the signs of the real and imaginary parts of the
2294 result are unspecified.

2295 If z is $+\text{Inf} + i\text{NaN}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned; the sign of the real part of the result is
2296 unspecified.

- 2297 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.
- 2298 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2299 invalid floating-point exception may be raised.
- 2300 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2301 Ref 7.26.5.7
2302 On page 679 line 23268 section `clock_getres()`, change:
- 2303 including the `nanosleep()` function
- 2304 to:
- 2305 including the `nanosleep()` and `thrd_sleep()` functions
- 2306 Ref G.6.3.2
2307 On page 687 line 23495 section `clog()`, add:
- 2308 [MXC]`clog(conj(z))`, `clogf(conjf(z))` and `clogl(conjl(z))` shall return exactly the same value as
2309 `conj(clog(z))`, `conjf(clogf(z))` and `conjl(clogl(z))`, respectively, including for the special
2310 values of z below.
- 2311 If z is $-0 + i0$, $-\text{Inf} + i\pi$ shall be returned and the divide-by-zero floating-point exception
2312 shall be raised.
- 2313 If z is $+0 + i0$, $-\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2314 shall be raised.
- 2315 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.
- 2316 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2317 point exception may be raised.
- 2318 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.
- 2319 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.
- 2320 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.
- 2321 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.
- 2322 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2323 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2324 point exception may be raised.
- 2325 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2326 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2327 Ref 7.26.3

2328 On page 698 line 23854 insert the following new `cond_*`() sections:

2329 [Note to reviewers: changes to `cond_broadcast` and `cond_signal` may be needed depending on the](#)
2330 [outcome of Mantis bug 609.](#)

2331 **NAME**

2332 `cond_broadcast`, `cond_signal` — broadcast or signal a condition

2333 **SYNOPSIS**

2334 `#include <threads.h>`

2335 `int cond_broadcast(cond_t *cond);`

2336 `int cond_signal(cond_t *cond);`

2337 **DESCRIPTION**

2338 [CX] The functionality described on this reference page is aligned with the ISO C standard.

2339 Any conflict between the requirements described here and the ISO C standard is

2340 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2341 The `cond_broadcast()` function shall unblock all of the threads that are blocked on the
2342 condition variable pointed to by `cond` at the time of the call.

2343 The `cond_signal()` function shall unblock one of the threads that are blocked on the condition
2344 variable pointed to by `cond` at the time of the call (if any threads are blocked on `cond`).

2345 If no threads are blocked on the condition variable pointed to by `cond` at the time of the call,
2346 these functions shall have no effect and shall return `thrd_success`.

2347 [CX]If more than one thread is blocked on a condition variable, the scheduling policy shall
2348 determine the order in which threads are unblocked. When each thread unblocked as a result
2349 of a `cond_broadcast()` or `cond_signal()` returns from its call to `cond_wait()` or `cond_timedwait()`,
2350 the thread shall own the mutex with which it called `cond_wait()` or `cond_timedwait()`. The
2351 thread(s) that are unblocked shall contend for the mutex according to the scheduling policy
2352 (if applicable), and as if each had called `mtx_lock()`.

2353 The `cond_broadcast()` and `cond_signal()` functions can be called by a thread whether or not it
2354 currently owns the mutex that threads calling `cond_wait()` or `cond_timedwait()` have associated
2355 with the condition variable during their waits; however, if predictable scheduling behavior is
2356 required, then that mutex shall be locked by the thread calling `cond_broadcast()` or
2357 `cond_signal()`.

2358 These functions shall not be affected if the calling thread executes a signal handler during
2359 the call.[/CX]

2360 The behavior is undefined if the value specified by the `cond` argument to `cond_broadcast()` or
2361 `cond_signal()` does not refer to an initialized condition variable.

2362 **RETURN VALUE**

2363 These functions shall return `thrd_success` on success, or `thrd_error` if the request
2364 could not be honored.

2365 **ERRORS**

2366 No errors are defined.

2367 **EXAMPLES**

2368 None.

2369 **APPLICATION USAGE**

2370 See the APPLICATION USAGE section for *pthread_cond_broadcast()*, substituting
2371 *cnd_broadcast()* for *pthread_cond_broadcast()* and *cnd_signal()* for *pthread_cond_signal()*.

2372 **RATIONALE**

2373 As for *pthread_cond_broadcast()* and *pthread_cond_signal()*, spurious wakeups may occur
2374 with *cnd_broadcast()* and *cnd_signal()*, necessitating that applications code a predicate-
2375 testing-loop around the condition wait. (See the RATIONALE section for
2376 *pthread_cond_broadcast()*.)

2377 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2378 B.2.3].

2379 **FUTURE DIRECTIONS**

2380 None.

2381 **SEE ALSO**

2382 *cnd_destroy*, *cnd_timedwait*, *pthread_cond_broadcast*

2383 XBD Section 4.12.2, <**threads.h**>

2384 **CHANGE HISTORY**

2385 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2386 **NAME**

2387 *cnd_destroy*, *cnd_init* — destroy and initialize condition variables

2388 **SYNOPSIS**

2389 `#include <threads.h>`

2390 `void cnd_destroy(cnd_t *cond);`

2391 `int cnd_init(cnd_t *cond);`

2392 **DESCRIPTION**

2393 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2394 Any conflict between the requirements described here and the ISO C standard is
2395 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2396 The *cnd_destroy()* function shall release all resources used by the condition variable pointed
2397 to by *cond*. It shall be safe to destroy an initialized condition variable upon which no threads
2398 are currently blocked. Attempting to destroy a condition variable upon which other threads
2399 are currently blocked results in undefined behavior. A destroyed condition variable object
2400 can be reinitialized using *cnd_init()*; the results of otherwise referencing the object after it
2401 has been destroyed are undefined. The behavior is undefined if the value specified by the
2402 *cond* argument to *cnd_destroy()* does not refer to an initialized condition variable.

2403 The *cnd_init()* function shall initialize a condition variable. If it succeeds it shall set the
2404 variable pointed to by *cond* to a value that uniquely identifies the newly initialized condition

2405 variable. Attempting to initialize an already initialized condition variable results in
2406 undefined behavior. A thread that calls *cnd_wait()* on a newly initialized condition variable
2407 shall block.

2408 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
2409 further requirements.

2410 These functions shall not be affected if the calling thread executes a signal handler during
2411 the call.[/CX]

2412 **RETURN VALUE**

2413 The *cnd_destroy()* function shall not return a value.

2414 The *cnd_init()* function shall return *thrd_success* on success, or *thrd_nomem* if no
2415 memory could be allocated for the newly created condition, or *thrd_error* if the request
2416 could not be honored.

2417 **ERRORS**

2418 See RETURN VALUE.

2419 **EXAMPLES**

2420 None.

2421 **APPLICATION USAGE**

2422 None.

2423 **RATIONALE**

2424 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2425 B.2.3].

2426 **FUTURE DIRECTIONS**

2427 None.

2428 **SEE ALSO**

2429 *cnd_broadcast*, *cnd_timedwait*

2430 XBD <**threads.h**>

2431 **CHANGE HISTORY**

2432 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2433 **NAME**

2434 *cnd_timedwait*, *cnd_wait* — wait on a condition

2435 **SYNOPSIS**

```
2436 #include <threads.h>  
2437 int cnd_timedwait(cnd_t * restrict cond, mtx_t * restrict mtx,  
2438                 const struct timespec * restrict ts);  
2439 int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

2440 **DESCRIPTION**

2441 [CX] The functionality described on this reference page is aligned with the ISO C standard.

2442 Any conflict between the requirements described here and the ISO C standard is
2443 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2444 The *cond_timedwait()* function shall atomically unlock the mutex pointed to by *mtx* and block
2445 until the condition variable pointed to by *cond* is signaled by a call to *cond_signal()* or to
2446 *cond_broadcast()*, or until after the TIME_UTC-based calendar time pointed to by *ts*, or until
2447 it is unblocked due to an unspecified reason.

2448 The *cond_wait()* function shall atomically unlock the mutex pointed to by *mtx* and block until
2449 the condition variable pointed to by *cond* is signaled by a call to *cond_signal()* or to
2450 *cond_broadcast()*, or until it is unblocked due to an unspecified reason.

2451 [CX]Atomically here means "atomically with respect to access by another thread to the
2452 mutex and then the condition variable". That is, if another thread is able to acquire the mutex
2453 after the about-to-block thread has released it, then a subsequent call to *cond_broadcast()* or
2454 *cond_signal()* in that thread shall behave as if it were issued after the about-to-block thread
2455 has blocked.[/CX]

2456 When the calling thread becomes unblocked, these functions shall lock the mutex pointed to
2457 by *mtx* before they return. The application shall ensure that the mutex pointed to by *mtx* is
2458 locked by the calling thread before it calls these functions.

2459 When using condition variables there is always a Boolean predicate involving shared
2460 variables associated with each condition wait that is true if the thread should proceed.
2461 Spurious wakeups from the *cond_timedwait()* and *cond_wait()* functions may occur. Since the
2462 return from *cond_timedwait()* or *cond_wait()* does not imply anything about the value of this
2463 predicate, the predicate should be re-evaluated upon such return.

2464 When a thread waits on a condition variable, having specified a particular mutex to either
2465 the *cond_timedwait()* or the *cond_wait()* operation, a dynamic binding is formed between that
2466 mutex and condition variable that remains in effect as long as at least one thread is blocked
2467 on the condition variable. During this time, the effect of an attempt by any thread to wait on
2468 that condition variable using a different mutex is undefined. Once all waiting threads have
2469 been unblocked (as by the *cond_broadcast()* operation), the next wait operation on
2470 that condition variable shall form a new dynamic binding with the mutex specified by that
2471 wait operation. Even though the dynamic binding between condition variable and mutex
2472 might be removed or replaced between the time a thread is unblocked from a wait on the
2473 condition variable and the time that it returns to the caller or begins cancellation cleanup, the
2474 unblocked thread shall always re-acquire the mutex specified in the condition wait operation
2475 call from which it is returning.

2476 [CX]A condition wait (whether timed or not) is a cancellation point. When the cancelability
2477 type of a thread is set to PTHREAD_CANCEL_DEFERRED, a side-effect of acting upon a
2478 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired
2479 before calling the first cancellation cleanup handler. The effect is as if the thread were
2480 unblocked, allowed to execute up to the point of returning from the call to *cond_timedwait()*
2481 or *cond_wait()*, but at that point notices the cancellation request and instead of returning to
2482 the caller of *cond_timedwait()* or *cond_wait()*, starts the thread cancellation activities, which
2483 includes calling cancellation cleanup handlers.

2484 A thread that has been unblocked because it has been canceled while blocked in a call to
2485 *cond_timedwait()* or *cond_wait()* shall not consume any condition signal that may be directed

2486 concurrently at the condition variable if there are other threads blocked on the condition
2487 variable.[/CX]

2488 When *cond_timedwait()* times out, it shall nonetheless release and re-acquire the mutex
2489 referenced by *mutex*, and may consume a condition signal directed concurrently at the
2490 condition variable.

2491 [CX]These functions shall not be affected if the calling thread executes a signal handler
2492 during the call, except that if a signal is delivered to a thread waiting for a condition
2493 variable, upon return from the signal handler either the thread shall resume waiting for the
2494 condition variable as if it was not interrupted, or it shall return *thrd_success* due to
2495 spurious wakeup.[/CX]

2496 The behavior is undefined if the value specified by the *cond* or *mtx* argument to these
2497 functions does not refer to an initialized condition variable or an initialized mutex object,
2498 respectively.

2499 **RETURN VALUE**

2500 The *cond_timedwait()* function shall return *thrd_success* upon success, or
2501 *thrd_timedout* if the time specified in the call was reached without acquiring the
2502 requested resource, or *thrd_error* if the request could not be honored.

2503 The *cond_wait()* function shall return *thrd_success* upon success or *thrd_error* if the
2504 request could not be honored.

2505 **ERRORS**

2506 See RETURN VALUE.

2507 **EXAMPLES**

2508 None.

2509 **APPLICATION USAGE**

2510 None.

2511 **RATIONALE**

2512 These functions are not affected by signal handlers (except as stated in the DESCRIPTION)
2513 for the reasons stated in [xref to XRAT B.2.3].

2514 **FUTURE DIRECTIONS**

2515 None.

2516 **SEE ALSO**

2517 *cond_broadcast*, *cond_destroy*, *timespec_get*

2518 XBD Section 4.12.2, <**threads.h**>

2519 **CHANGE HISTORY**

2520 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2521 Ref F.10.8.1 para 2

2522 On page 705 line 24155 section *copysign()*, add a new paragraph:

2523 [MX]The returned value shall be exact and shall be independent of the current rounding
2524 direction mode.[/MX]

2525 Ref G.6.4.1 para 1
2526 On page 711 line 24308 section `cpow()`, add a new paragraph:

2527 [MXC]These functions shall raise floating-point exceptions if appropriate for the calculation
2528 of the parts of the result, and may also raise spurious floating-point exceptions.[/MXC]

2529 Ref G.6.4.1 footnote 386
2530 On page 711 line 24318 section `cpow()`, change RATIONALE from:

2531 None.

2532 to:

2533 Permitting spurious floating-point exceptions allows `cpow(z, c)` to be implemented as `cexp(c`
2534 `clog(z))` without precluding implementations that treat special cases more carefully.

2535 Ref G.6 para 7, G.6.2.5
2536 On page 718 line 24545 section `csin()`, add:

2537 [MXC]`csin(conj(iz))`, `csinf(conjf(iz))` and `csinl(conjl(iz))` shall return exactly the same value
2538 as `conj(csin(iz))`, `conjf(csinf(iz))` and `conjl(csinl(iz))`, respectively, and `csin(-iz)`, `csinf(-iz)`
2539 and `csinl(-iz)` shall return exactly the same value as `-csin(iz)`, `-csinf(iz)` and `-csinl(iz)`,
2540 respectively, including for the special values of `iz` below.

2541 If `iz` is `+0 + i0`, `-i (+0 + i0)` shall be returned.

2542 If `iz` is `+0 + iInf`, `-i ($\pm 0 + iNaN$)` shall be returned and the invalid floating-point exception
2543 shall be raised; the sign of the imaginary part of the result is unspecified.

2544 If `iz` is `+0 + iNaN`, `-i ($\pm 0 + iNaN$)` shall be returned; the sign of the imaginary part of the
2545 result is unspecified.

2546 If `iz` is `x + iInf` where `x` is positive and finite, `-i (NaN + iNaN)` shall be returned and the
2547 invalid floating-point exception shall be raised.

2548 If `iz` is `x + iNaN` where `x` is non-zero and finite, `-i (NaN + iNaN)` shall be returned and the
2549 invalid floating-point exception may be raised.

2550 If `iz` is `+Inf + i0`, `-i (+Inf + i0)` shall be returned.

2551 If `iz` is `+Inf + iy` where `y` is positive and finite, `-iInf (cos(y) + isin(y))` shall be returned.

2552 If `iz` is `+Inf + iInf`, `-i ($\pm Inf + iNaN$)` shall be returned and the invalid floating-point exception
2553 shall be raised; the sign of the imaginary part of the result is unspecified.

2554 If `iz` is `+Inf + iNaN`, `-i ($\pm Inf + iNaN$)` shall be returned; the sign of the imaginary part of the
2555 result is unspecified.

- 2556 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.
- 2557 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
2558 invalid floating-point exception may be raised.
- 2559 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2560 Ref G.6 para 7
2561 On page 718 line 24553 section `csin()`, change RATIONALE from:
- 2562 None.
- 2563 to:
- 2564 The MXC special cases for `csin()` are derived from those for `csinh()` by applying the formula
2565 $csin(z) = -i csinh(iz)$.
- 2566 Ref G.6.2.5
2567 On page 719 line 24574 section `csinh()`, add:
- 2568 [MXC]`csinh(conj(z))`, `csinhf(conjf(z))` and `csinhl(conjl(z))` shall return exactly the same
2569 value as `conj(csinh(z))`, `conjf(csinhf(z))` and `conjl(csinhl(z))`, respectively, and `csinh(-z)`,
2570 `csinhf(-z)` and `csinhl(-z)` shall return exactly the same value as $-csinh(z)$, $-csinhf(z)$ and
2571 $-csinhl(z)$, respectively, including for the special values of z below.
- 2572 If z is $+0 + i0$, $+0 + i0$ shall be returned.
- 2573 If z is $+0 + i\text{Inf}$, $\pm 0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2574 raised; the sign of the real part of the result is unspecified.
- 2575 If z is $+0 + i\text{NaN}$, $\pm 0 + i\text{NaN}$ shall be returned; the sign of the real part of the result is
2576 unspecified.
- 2577 If z is $x + i\text{Inf}$ where x is positive and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2578 floating-point exception shall be raised.
- 2579 If z is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2580 floating-point exception may be raised.
- 2581 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.
- 2582 If z is $+\text{Inf} + iy$ where y is positive and finite, $+\text{Inf}(\cos(y) + i\sin(y))$ shall be returned.
- 2583 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2584 shall be raised; the sign of the real part of the result is unspecified.
- 2585 If z is $+\text{Inf} + i\text{NaN}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned; the sign of the real part of the result is
2586 unspecified.
- 2587 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.
- 2588 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the

2589 invalid floating-point exception may be raised.

2590 If z is NaN + i NaN, NaN + i NaN shall be returned.[/MXC]

2591 Ref G.6.4.2

2592 On page 721 line 24612 section `csqrt()`, add:

2593 [MXC]`csqrt(conj(z))`, `csqrtf(conjf(z))` and `csqrtl(conjl(z))` shall return exactly the same value
2594 as `conj(csqrt(z))`, `conjf(csqrtf(z))` and `conjl(csqrtl(z))`, respectively, including for the special
2595 values of z below.

2596 If z is $\pm 0 + i0$, $+0 + i0$ shall be returned.

2597 If the imaginary part of z is Inf, $+Inf + iInf$, shall be returned.

2598 If z is $x + iNaN$ where x is finite, NaN + $iNaN$ shall be returned and the invalid floating-
2599 point exception may be raised.

2600 If z is $-Inf + iy$ where y is positive-signed and finite, $+0 + iInf$ shall be returned.

2601 If z is $+Inf + iy$ where y is positive-signed and finite, $+Inf + i0$ shall be returned.

2602 If z is $-Inf + iNaN$, NaN $\pm iInf$ shall be returned; the sign of the imaginary part of the result
2603 is unspecified.

2604 If z is $+Inf + iNaN$, $+Inf + iNaN$ shall be returned.

2605 If z is NaN + iy where y is finite, NaN + $iNaN$ shall be returned and the invalid floating-
2606 point exception may be raised.

2607 If z is NaN + $iNaN$, NaN + $iNaN$ shall be returned.[/MXC]

2608 Ref G.6 para 7, G.6.2.6

2609 On page 722 line 24641 section `ctan()`, add:

2610 [MXC]`ctan(conj(iz))`, `ctanf(conjf(iz))` and `ctanl(conjl(iz))` shall return exactly the same value
2611 as `conj(ctan(iz))`, `conjf(ctanf(iz))` and `conjl(ctanl(iz))`, respectively, and `ctan(-iz)`, `ctanf(-iz)`
2612 and `ctanl(-iz)` shall return exactly the same value as `-ctan(iz)`, `-ctanf(iz)` and `-ctanl(iz)`,
2613 respectively, including for the special values of iz below.

2614 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.

2615 If iz is $0 + iInf$, $-i (0 + iNaN)$ shall be returned and the invalid floating-point exception shall
2616 be raised.

2617 If iz is $x + iInf$ where x is non-zero and finite, $-i (NaN + iNaN)$ shall be returned and the
2618 invalid floating-point exception shall be raised.

2619 If iz is $0 + iNaN$, $-i (0 + iNaN)$ shall be returned.

2620 If iz is $x + iNaN$ where x is non-zero and finite, $-i (NaN + iNaN)$ shall be returned and the
2621 invalid floating-point exception may be raised.

- 2622 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i(1 + i0 \sin(2y))$ shall be returned.
- 2623 If iz is $+\text{Inf} + i\text{Inf}$, $-i(1 \pm i0)$ shall be returned; the sign of the real part of the result is
2624 unspecified.
- 2625 If iz is $+\text{Inf} + i\text{NaN}$, $-i(1 \pm i0)$ shall be returned; the sign of the real part of the result is
2626 unspecified.
- 2627 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.
- 2628 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
2629 invalid floating-point exception may be raised.
- 2630 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2631 Ref G.6 para 7
2632 On page 722 line 24649 section `ctan()`, change RATIONALE from:
- 2633 None.
- 2634 to:
- 2635 The MXC special cases for `ctan()` are derived from those for `ctanh()` by applying the
2636 formula $ctan(z) = -i ctanh(iz)$.
- 2637 Ref G.6.2.6
2638 On page 723 line 24670 section `ctanh()`, add:
- 2639 [MXC]`ctanh(conj(z))`, `ctanhf(conjf(z))` and `ctanhl(conjl(z))` shall return exactly the same
2640 value as `conj(ctanh(z))`, `conjf(ctanhf(z))` and `conjl(ctanhl(z))`, respectively, and `ctanh(-z)`,
2641 `ctanhf(-z)` and `ctanhl(-z)` shall return exactly the same value as $-ctanh(z)$, $-ctanhf(z)$ and
2642 $-ctanhl(z)$, respectively, including for the special values of z below.
- 2643 If z is $+0 + i0$, $+0 + i0$ shall be returned.
- 2644 If z is $0 + i\text{Inf}$, $0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2645 raised.
- 2646 If z is $x + i\text{Inf}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2647 floating-point exception shall be raised.
- 2648 If z is $0 + i\text{NaN}$, $0 + i\text{NaN}$ shall be returned.
- 2649 If z is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2650 floating-point exception may be raised.
- 2651 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $1 + i0 \sin(2y)$ shall be returned.
- 2652 If z is $+\text{Inf} + i\text{Inf}$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2653 unspecified.

2654 If z is $+\text{Inf} + i\text{NaN}$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2655 unspecified.

2656 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2657 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2658 invalid floating-point exception may be raised.

2659 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2660 Ref 7.27.3, 7.1.4 para 5
2661 On page 727 line 24774 section `ctime()`, change:

2662 [CX]The `ctime()` function need not be thread-safe.[/CX]

2663 to:
2664 The `ctime()` function need not be thread-safe; however, `ctime()` shall avoid data races with all
2665 functions other than itself, `asctime()`, `gmtime()` and `localtime()`.

2666 Ref 7.5 para 2
2667 On page 781 line 26447 section `errno`, change:

2668 The lvalue `errno` is used by many functions to return error values.

2669 to:

2670 The lvalue to which the macro `errno` expands is used by many functions to return error
2671 values.

2672 Ref 7.5 para 3
2673 On page 781 line 26449 section `errno`, change:

2674 The value of `errno` shall be defined only after a call to a function for which it is explicitly
2675 stated to be set and until it is changed by the next function call or if the application assigns it
2676 a value.

2677 to:

2678 The value of `errno` in the initial thread shall be zero at program startup (the initial value of
2679 `errno` in other threads is an indeterminate value) and shall otherwise be defined only after a
2680 call to a function for which it is explicitly stated to be set and until it is changed by the next
2681 function call or if the application assigns it a value.

2682 Ref 7.5 para 2
2683 On page 781 line 26456 section `errno`, delete:

2684 It is unspecified whether `errno` is a macro or an identifier declared with external linkage.

2685 Ref 7.22.4.4 para 2
2686 On page 796 line 27057 section `exit()`, add a new (unshaded) paragraph:

2687 The `exit()` function shall cause normal process termination to occur. No functions registered

2688 by the *at_quick_exit()* function shall be called. If a process calls the *exit()* function more
2689 than once, or calls the *quick_exit()* function in addition to the *exit()* function, the behavior is
2690 undefined.

2691 Ref 7.22.4.4 para 2
2692 On page 796 line 27068 section *exit()*, delete:

2693 If *exit()* is called more than once, the behavior is undefined.

2694 Ref 7.22.4.3, 7.22.4.7
2695 On page 796 line 27086 section *exit()*, add *at_quick_exit* and *quick_exit* to the SEE ALSO section.

2696 Ref F.10.4.2 para 2
2697 On page 804 line 27323 section *fabs()*, add a new paragraph:

2698 [MX]The returned value shall be exact and shall be independent of the current rounding
2699 direction mode.[/MX]

2700 Ref 7.21.2 para 7,8
2701 On page 874 line 29483 section *flockfile()*, change:

2702 These functions shall provide for explicit application-level locking of **(FILE *)**
2703 objects.

2704 to:

2705 These functions shall provide for explicit application-level locking of the locks associated
2706 with standard I/O streams (see [xref to 2.5]).

2707 Ref 7.21.2 para 7,8
2708 On page 874 line 29499 section *flockfile()*, delete:

2709 All functions that reference **(FILE *)** objects, except those with names ending in *_unlocked*,
2710 shall behave as if they use *flockfile()* and *funlockfile()* internally to obtain ownership of these
2711 **(FILE *)** objects.

2712 Ref F.10.6.2 para 3
2713 On page 876 line 29560 section *floor()*, add a new paragraph:

2714 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2715 arguments.[/MX]

2716 Ref F.10.6.2 para 2
2717 On page 876 line 29562 section *floor()*, change:

2718 [MX]The result shall have the same sign as *x*.[/MX]

2719 to:

2720 [MX]The returned value shall be independent of the current rounding direction mode and
2721 shall have the same sign as *x*.[/MX]

- 2722 Ref F.10.6.2 para 3
2723 On page 876 line 29576 section `floor()`, delete from APPLICATION USAGE:
- 2724 These functions may raise the inexact floating-point exception if the result differs in value
2725 from the argument.
- 2726 Ref F.10.9.2 para 2
2727 On page 880 line 29695 section `fmax()`, add a new paragraph:
- 2728 [MX]The returned value shall be exact and shall be independent of the current rounding
2729 direction mode.[/MX]
- 2730 Ref F.10.9.3 para 2
2731 On page 884 line 29844 section `fmin()`, add a new paragraph:
- 2732 [MX]The returned value shall be exact and shall be independent of the current rounding
2733 direction mode.[/MX]
- 2734 Ref F.10.7.1 para 2
2735 On page 885 line 29892 section `fmod()`, change:
- 2736 [MXX]If the correct value would cause underflow, and is representable, a range error may
2737 occur and the correct value shall be returned.[/MXX]
- 2738 to:
- 2739 [MX]When subnormal results are supported, the returned value shall be exact and shall be
2740 independent of the current rounding direction mode.[/MX]
- 2741 Ref 7.21.5.3 para 3
2742 On page 892 line 30125 section `fopen()`, add:
- 2743 `wx` or `wbx` Create file for writing.
- 2744 Ref 7.21.5.3 para 3
2745 On page 892 line 30128 section `fopen()`, add:
- 2746 `w+x` or `w+bx` or `wb+x` Create file for update.
- 2747 Ref 7.21.5.3 para 5
2748 On page 892 line 30132 section `fopen()`, add a new paragraph and list:
- 2749 Opening a file with exclusive mode (*x* as the last character in the *mode* argument) shall fail
2750 if the file already exists or cannot be created. Otherwise, the file shall be created with
2751 exclusive (also known as non-shared) access to the extent that the underlying file system
2752 supports exclusive access.
- 2753 [Note to reviewers: This “exclusive access” requirement is the subject of discussions in WG14](#)
2754 [which hopefully will result in a clarification in C2x, in which case the above text will be changed to](#)
2755 [match the proposed C2x text.](#)
- 2756 Ref 7.21.5.3 para 3

2757 On page 892 line 30144 section `fopen()`, change:

2758 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and ...

2759 to:

2760 If the first character in *mode* is *w* or *a*, and ...

2761 Ref 7.21.5.3 para 3,5

2762 On page 892 line 30148 section `fopen()`, change:

2763 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously
2764 exist, the `fopen()` function shall create a file as if it called the `creat()` function with a value
2765 appropriate for the *path* argument interpreted from *pathname* and a value of `S_IRUSR` |
2766 `S_IWUSR` | `S_IRGRP` | `S_IWGRP` | `S_IROTH` | `S_IWOTH` for the *mode* argument.

2767 to:

2768 If the first character in *mode* is *w* or *a*, and the file did not previously exist, the `fopen()`
2769 function shall create a file as if it called the `open()` function with a value appropriate for the
2770 *path* argument interpreted from *pathname*, a value for the *oflag* argument as specified below,
2771 and a value of `S_IRUSR` | `S_IWUSR` | `S_IRGRP` | `S_IWGRP` | `S_IROTH` | `S_IWOTH` for
2772 the third argument.

2773 Ref 7.21.5.3 para 5

2774 On page 893 line 30158 section `fopen()`, change:

2775 The file descriptor ...

2776 to:

2777 If the last character in *mode* is not *x*, the file descriptor ...

2778 Ref 7.21.5.3 para 5

2779 On page 893 line 30166 section `fopen()`, add the following new paragraphs:

2780 [CX]If the last character in *mode* is *x* and the underlying file system does not support
2781 exclusive access, the file descriptor associated with the opened stream shall be allocated and
2782 opened as if by a call to `open()` with the following flags:

<i>fopen()</i> Mode	<i>open()</i> Flags
<i>wx</i> or <i>wbx</i>	<code>O_WRONLY</code> <code>O_CREAT</code> <code>O_EXCL</code> <code>O_TRUNC</code>
<i>w+x</i> or <i>w+bx</i> or <i>wb+x</i>	<code>O_RDWR</code> <code>O_CREAT</code> <code>O_EXCL</code> <code>O_TRUNC</code>

2783 If the last character in *mode* is *x* and the underlying file system supports exclusive access,
2784 the file descriptor associated with the opened stream shall be allocated and opened as if by a
2785 call to `open()` with the above flags or with the above flags ORed with an implementation-
2786 defined file creation flag if necessary to enable exclusive access (see above).[CX]

2787 [Note to reviewers: The above change may need to be updated depending on the outcome of WG14](#)
2788 [discussions about the “exclusive access” requirement.](#)

2789 Ref 7.21.5.3 para 5

2790 On page 893 line 30175 section `fopen()`, add (within the CX shading):

2791 [EEXIST] The last character in *mode* is *x* and the named file exists.

2792 Ref 7.21.5.3 para 5

2793 On page 895 line 30236 section `fopen()`, change APPLICATION USAGE from:

2794 None.

2795 to:

2796 If an application needs to create a file in a way that fails if the file already exists, and either
2797 requires that it does not have exclusive access to the file or does not need exclusive access, it
2798 should use `open()` with the `O_CREAT` and `O_EXCL` flags instead of using `fopen()` with an *x*
2799 in the *mode*. A stream can then be created, if needed, by calling `fdopen()` on the file
2800 descriptor returned by `open()`.

2801 [Note to reviewers: The above change may need to be updated depending on the outcome of WG14](#)
2802 [discussions about the “exclusive access” requirement.](#)

2803 Ref 7.21.5.3 para 5

2804 On page 895 line 30238 section `fopen()`, change RATIONALE from:

2805 None.

2806 to:

2807 When the last character in *mode* is *x*, the ISO C standard requires that the file is created with
2808 exclusive access to the extent that the underlying system supports exclusive access.
2809 Although POSIX.1 does not specify any method of enabling exclusive access, it allows for
2810 the existence of an implementation-defined file creation flag that enables it. Note that it must
2811 be a file creation flag, not a file access mode flag (that is, one that is included in
2812 `O_ACCMODE`) or a file status flag, so that it does not affect the value returned by `fcntl()`
2813 with `F_GETFL`. On implementations that have such a flag, if support for it is file system
2814 dependent and exclusive access is requested when using `fopen()` to create a file on a file
2815 system that does not support it, the flag must not be used if it would cause `fopen()` to fail.

2816 Some implementations support mandatory file locking as a means of enabling exclusive
2817 access to a file. Locks are set in the normal way, but instead of only preventing others from
2818 setting conflicting locks they prevent others from accessing the contents of the locked part
2819 of the file in a way that conflicts with the lock. However, unless the implementation has a
2820 way of setting a whole-file write lock on file creation, this does not satisfy the requirement
2821 in the ISO C standard that the file is “created with exclusive access to the extent that the
2822 underlying system supports exclusive access”. (Having `fopen()` create the file and set a lock
2823 on the file as two separate operations is not the same, and it would introduce a race
2824 condition whereby another process could open the file and write to it (or set a lock) in
2825 between the two operations.) However, on all implementations that support mandatory file
2826 locking, its use is discouraged; therefore, it is recommended that implementations which
2827 support mandatory file locking do **not** add a means of creating a file with a whole-file
2828 exclusive lock set, so that `fopen()` is not required to enable mandatory file locking.

2829 [Note to reviewers: The above change may need to be updated depending on the outcome of WG14](#)
2830 [discussions about the “exclusive access” requirement.](#)

2831 Ref 7.22.3.3 para 2

2832 On page 933 line 31673 section `free()`, change:

2833 Otherwise, if the argument does not match a pointer earlier returned by a function in
2834 POSIX.1-2017 that allocates memory as if by `malloc()`, or if the space has been deallocated
2835 by a call to `free()` or `realloc()`, the behavior is undefined.

2836 to:

2837 Otherwise, if the argument does not match a pointer earlier returned by `aligned_alloc()`,
2838 `calloc()`, `malloc()`, `[ADV]posix_memalign()`, `[/ADV] realloc()`, or a function in POSIX.1-
2839 20xx that allocates memory as if by `malloc()`, or if the space has been deallocated by a call
2840 to `free()` or `realloc()`, the behavior is undefined.

2841 Ref 7.22.3 para 2

2842 On page 933 line 31677 section `free()`, add a new paragraph:

2843 For purposes of determining the existence of a data race, `free()` shall behave as though it
2844 accessed only memory locations accessible through its argument and not other static
2845 duration storage. The function may, however, visibly modify the storage that it deallocates.
2846 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `[ADV]posix_memalign()`, `[/ADV]` and
2847 `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total
2848 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
2849 next allocation (if any) in this order.

2850 Ref 7.22.3.1

2851 On page 933 line 31691 section `free()`, add `aligned_alloc` to the SEE ALSO section.

2852 Ref 7.21.5.3 para 3,5; 7.21.5.4 para 2

2853 On page 942 line 32010 section `freopen()`, replace the following text:

2854 shall be allocated and opened as if by a call to `open()` with the following flags:

2855 and the table that follows it with:

2856 shall be allocated and opened as if by a call to `open()` with the flags specified for `fopen()`
2857 with the same `mode` argument.

2858 Ref (none)

2859 On page 944 line 32094 section `freopen()`, change:

2860 It is possible that these side-effects are an unintended consequence of the way the feature is
2861 specified in the ISO/IEC 9899: 1999 standard, but unless or until the ISO C standard is
2862 changed, ...

2863 to:

2864 It is possible that these side-effects are an unintended consequence of the way the feature

2865 was specified in the ISO/IEC 9899: 1999 standard (and still is in the current standard), but
2866 unless or until the ISO C standard is changed, ...

2867 [Note to reviewers: if the APPLICATION USAGE and RATIONALE additions for fopen\(\) are](#)
2868 [retained, changes should be added here to make the equivalent sections for freopen\(\) refer to those](#)
2869 [for fopen\(\).](#)

2870 Ref 7.12.6.4 para 3
2871 On page 947 line 32161 section frexp(), change:

2872 The integer exponent shall be stored in the **int** object pointed to by *exp*.

2873 to:

2874 The integer exponent shall be stored in the **int** object pointed to by *exp*; if the integer
2875 exponent is outside the range of **int**, the results are unspecified.

2876 Ref F.10.3.4 para 3
2877 On page 947 line 32164 section frexp(), add a new paragraph:

2878 [MX]When the radix of the argument is a power of 2, the returned value shall be exact and
2879 shall be independent of the current rounding direction mode.[/MX]

2880 Ref 7.21.6.2 para 4
2881 On page 950 line 32239 section fscanf(), change:

2882 If a directive fails, as detailed below, the function shall return.

2883 to:

2884 When all directives have been executed, or if a directive fails (as detailed below), the
2885 function shall return.

2886 Ref 7.21.6.2 para 5
2887 On page 950 line 32242 section fscanf(), after applying bug 1163 change:

2888 A directive composed of one or more white-space bytes shall be executed by reading input
2889 until no more valid input can be read, or up to the first non-white-space byte , which remains
2890 unread.

2891 to:

2892 A directive composed of one or more white-space bytes shall be executed by reading input
2893 up to the first non-white-space byte, which shall remain unread, or until no more bytes can
2894 be read. The directive shall never fail.

2895 Ref (none)
2896 On page 955 line 32471 section fscanf(), change:

2897 This function is aligned with the ISO/IEC 9899: 1999 standard, and in doing so a few
2898 “obvious” things were not included. Specifically, the set of characters allowed in a scanset is
2899 limited to single-byte characters. In other similar places, multi-byte characters have been

2900 permitted, but for alignment with the ISO/IEC 9899: 1999 standard, it has not been done
2901 here.

2902 to:

2903 The set of characters allowed in a scanset is limited to single-byte characters. In other
2904 similar places, multi-byte characters have been permitted, but for alignment with the ISO C
2905 standard, it has not been done here.

2906 Ref 7.29.2.2 para 4
2907 On page 1004 line 34144 section `fwscanf()`, change:

2908 If a directive fails, as detailed below, the function shall return.

2909 to:

2910 When all directives have been executed, or if a directive fails (as detailed below), the
2911 function shall return.

2912 Ref 7.29.2.2 para 5
2913 On page 1004 line 34147 section `fwscanf()`, change:

2914 A directive composed of one or more white-space wide characters is executed by reading
2915 input until no more valid input can be read, or up to the first wide character which is not a
2916 white-space wide character, which remains unread.

2917 to:

2918 A directive composed of one or more white-space wide characters shall be executed by
2919 reading input up to the first wide character that is not a white-space wide character, which
2920 shall remain unread, or until no more wide characters can be read. The directive shall never
2921 fail.

2922 Ref 7.27.3, 7.1.4 para 5
2923 On page 1113 line 37680 section `gmtime()`, change:

2924 [CX]The *gmtime()* function need not be thread-safe.[/CX]

2925 to:

2926 The *gmtime()* function need not be thread-safe; however, *gmtime()* shall avoid data races
2927 with all functions other than itself, *asctime()*, *ctime()* and *localtime()*.

2928 Ref F.10.3.5 para 1
2929 On page 1133 line 38281 section `ilogb()`, add a new paragraph:

2930 [MX]When the correct result is representable in the range of the return type, the returned
2931 value shall be exact and shall be independent of the current rounding direction mode.[/MX]

2932 Ref F.10.3.5 para 3
2933 On page 1133 line 38282,38285,38288 section `ilogb()`, change:

2934 [XSI]On XSI-conformant systems, a domain error shall occur[/XSI]

2935 to:

2936 [XSI|MX]On XSI-conformant systems and on systems that support the IEC 60559 Floating-
2937 Point option, a domain error shall occur[/XSI|MX]

2938 Ref 7.12.6.5 para 2
2939 On page 1133 line 38291 section `ilogb()`, change:

2940 If the correct value is greater than `{INT_MAX}`, [MX]a domain error shall occur and[/MX]
2941 an unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error
2942 shall occur and `{INT_MAX}` shall be returned.[/XSI]

2943 If the correct value is less than `{INT_MIN}`, [MX]a domain error shall occur and[/MX] an
2944 unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error shall
2945 occur and `{INT_MIN}` shall be returned.[/XSI]

2946 to:

2947 If the correct value is greater than `{INT_MAX}` or less than `{INT_MIN}`, an unspecified
2948 value shall be returned. [XSI]On XSI-conformant systems, a domain error shall occur and
2949 `{INT_MAX}` or `{INT_MIN}`, respectively, shall be returned;[/XSI] [MX]if the IEC 60559
2950 Floating-Point option is supported, a domain error shall occur;[/MX] otherwise, a domain
2951 error or range error may occur.

2952 Ref F.10.3.5 para 3
2953 On page 1133 line 38300 section `ilogb()`, change:

2954 [XSI]The `x` argument is zero, NaN, or $\pm\text{Inf}$.[/XSI]

2955 to:

2956 [XSI|MX]The `x` argument is zero, NaN, or $\pm\text{Inf}$.[/XSI|MX]

2957 Ref F.10.11 para 1
2958 On page 1174 line 39604 section `isgreater()`,
2959 and page 1175 line 39642 section `isgreaterequal()`,
2960 and page 1177 line 39708 section `isless()`,
2961 and page 1178 line 39746 section `islessequal()`,
2962 and page 1179 line 39784 section `islessgreater()`, add a new paragraph:

2963 [MX]Relational operators and their corresponding comparison macros shall produce
2964 equivalent result values, even if argument values are represented in wider formats. Thus,
2965 comparison macro arguments represented in formats wider than their semantic types shall
2966 not be converted to the semantic types, unless the wide evaluation method converts operands
2967 of relational operators to their semantic types. The standard wide evaluation methods
2968 characterized by `FLT_EVAL_METHOD` equal to 1 or 2 (see [xref to `<float.h>`]) do not
2969 convert operands of relational operators to their semantic types.[/MX]

2970 (The editors may wish to merge the pages for the above interfaces to reduce duplication – they have
2971 duplicate APPLICATION USAGE as well.)

2972 Ref 7.30.2.2.1 para 4
2973 On page 1202 line 40411 section `iswctype()`, remove the CX shading from:

2974 If *charclass* is (`wctype_t`)0, these functions shall return 0.

2975 Ref 7.17.3.1
2976 On page 1229 line 41126 insert a new `kill_dependency()` section:

2977 **NAME**
2978 `kill_dependency` — terminate a dependency chain

2979 **SYNOPSIS**
2980 `#include <stdatomic.h>`
2981 `type kill_dependency(type y);`

2982 **DESCRIPTION**
2983 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2984 Any conflict between the requirements described here and the ISO C standard is
2985 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2986 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
2987 `<stdatomic.h>` header nor support this macro.

2988 The `kill_dependency()` macro shall terminate a dependency chain (see [xref to XBD 4.12.1
2989 Memory Ordering]). The argument shall not carry a dependency to the return value.

2990 **RETURN VALUE**
2991 The `kill_dependency()` macro shall return the value of *y*.

2992 **ERRORS**
2993 No errors are defined.

2994 **EXAMPLES**
2995 None.

2996 **APPLICATION USAGE**
2997 None.

2998 **RATIONALE**
2999 None.

3000 **FUTURE DIRECTIONS**
3001 None.

3002 **SEE ALSO**
3003 XBD Section 4.12.1, `<stdatomic.h>`

3004 **CHANGE HISTORY**
3005 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3006 Ref 7.12.8.3, 7.1.4 para 5

3007 On page 1241 line 41433 section `lgamma()`, change:

3008 [CX]These functions need not be thread-safe.[/CX]

3009 to:

3010 [XSI]If concurrent calls are made to these functions, the value of *signgam* is indeterminate.
3011 [/XSI]

3012 Ref 7.12.8.3, 7.1.4 para 5

3013 On page 1242 line 41464 section `lgamma()`, add a new paragraph to APPLICATION USAGE:

3014 If the value of *signgam* will be obtained after a call to *lgamma()*, *lgammaf()*, or *lgammal()*,
3015 in order to ensure that the value will not be altered by another call in a different thread,
3016 applications should either restrict calls to these functions to be from a single thread or use a
3017 lock such as a mutex or spin lock to protect a critical section starting before the function call
3018 and ending after the value of *signgam* has been obtained.

3019 Ref 7.12.8.3, 7.1.4 para 5

3020 On page 1242 line 41466 section `lgamma()`, change RATIONALE from:

3021 None.

3022 to:

3023 Earlier versions of this standard did not require *lgamma()*, *lgammaf()*, and *lgammal()* to be
3024 thread-safe because *signgam* was a global variable. They are now required to be thread-safe
3025 to align with the ISO C standard (which, since the introduction of threads in 2011, requires
3026 that they avoid data races), with the exception that they need not avoid data races when
3027 storing a value in the *signgam* variable. Since *signgam* is not specified by the ISO C
3028 standard, this exception is not a conflict with that standard.

3029 Ref 7.11.2.1, 7.1.4 para 5

3030 On page 1262 line 42124 section `localeconv()`, change:

3031 [CX]The *localeconv()* function need not be thread-safe.[/CX]

3032 to:

3033 The *localeconv()* function need not be thread-safe; however, *localeconv()* shall avoid data
3034 races with all other functions.

3035 Ref 7.27.3, 7.1.4 para 5

3036 On page 1265 line 42217 section `localtime()`, change:

3037 [CX]The *localtime()* function need not be thread-safe.[/CX]

3038 to:

3039 The *localtime()* function need not be thread-safe; however, *localtime()* shall avoid data races
3040 with all functions other than itself, *asctime()*, *ctime()* and *gmtime()*.

3041 Ref F.10.3.11 para 2

3042 On page 1280 line 42723 section `logb()`, add a new paragraph:

3043 [MX]The returned value shall be exact and shall be independent of the current rounding
3044 direction mode.[/MX]

3045 Ref 7.13.2.1 para 1
3046 On page 1283 line 42780 section `longjmp()`, change:

3047 `void longjmp(jmp_buf env, int val);`

3048 to:

3049 `_Noreturn void longjmp(jmp_buf env, int val);`

3050 Ref 7.13.2.1 para 2
3051 On page 1283 line 42804 section `longjmp()`, remove the CX shading from:

3052 The effect of a call to `longjmp()` where initialization of the **`jmp_buf`** structure was not
3053 performed in the calling thread is undefined.

3054 Ref 7.13.2.1 para 4
3055 On page 1283 line 42807 section `longjmp()`, change:

3056 After `longjmp()` is completed, program execution continues ...

3057 to:

3058 After `longjmp()` is completed, thread execution shall continue ...

3059 Ref 7.22.3 para 1
3060 On page 1295 line 43144 section `malloc()`, change:

3061 a pointer to any type of object

3062 to:

3063 a pointer to any type of object with a fundamental alignment requirement

3064 Ref 7.22.3 para 1
3065 On page 1295 line 43148 section `malloc()`, change:

3066 either a null pointer shall be returned, or ...

3067 to:

3068 either a null pointer shall be returned to indicate an error, or ...

3069 Ref 7.22.3 para 2
3070 On page 1295 line 43150 section `malloc()`, add a new paragraph:

3071 For purposes of determining the existence of a data race, `malloc()` shall behave as though it
3072 accessed only memory locations accessible through its argument and not other static

3073 duration storage. The function may, however, visibly modify the storage that it allocates.
3074 Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] and
3075 *realloc()* that allocate or deallocate a particular region of memory shall occur in a single total
3076 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
3077 next allocation (if any) in this order.

3078 Ref 7.22.3.1

3079 On page 1295 line 43171 section *malloc()*, add *aligned_alloc* to the SEE ALSO section.

3080 Ref 7.22.7.1 para 2

3081 On page 1297 line 43194 section *mblen()*, change:

3082 `mbtowc((wchar_t *)0, s, n);`

3083 to:

3084 `mbtowc((wchar_t *)0, (const char *)0, 0);`

3085 `mbtowc((wchar_t *)0, s, n);`

3086 Ref 7.22.7 para 1

3087 On page 1297 line 43198 section *mblen()*, change:

3088 this function shall be placed into its initial state by a call for which

3089 to:

3090 this function shall be placed into its initial state at program startup and can be returned to
3091 that state by a call for which

3092 Ref 7.22.7 para 1, 7.1.4 para 5

3093 On page 1297 line 43206 section *mblen()*, change:

3094 [CX]The *mblen()* function need not be thread-safe.[/CX]

3095 to:

3096 The *mblen()* function need not be thread-safe; however, it shall avoid data races with all
3097 other functions.

3098 Ref 7.29.6.3 para 1, 7.1.4 para 5

3099 On page 1299 line 43254 section *mbrlen()*, change:

3100 [CX]The *mbrlen()* function need not be thread-safe if called with a NULL *ps* argument.
3101 [/CX]

3102 to:

3103 If called with a null *ps* argument, the *mbrlen()* function need not be thread-safe; however,
3104 such calls shall avoid data races with calls to *mbrlen()* with a non-null argument and with
3105 calls to all other functions.

3106 Ref 7.28.1, 7.1.4 para 5

3107 On page 1301 line 43296 insert a new *mbrtoc16()* section:

3108 **NAME**

3109 `mbrtoc16`, `mbrtoc32` — convert a character to a Unicode character code (restartable)

3110 **SYNOPSIS**

3111 `#include <uchar.h>`

```
3112     size_t mbrtoc16(char16_t *restrict pc16, const char *restrict s,  
3113                   size_t n, mbstate_t *restrict ps);  
3114     size_t mbrtoc32(char32_t *restrict pc32, const char *restrict s,  
3115                   size_t n, mbstate_t *restrict ps);
```

3116 **DESCRIPTION**

3117 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3118 Any conflict between the requirements described here and the ISO C standard is
3119 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3120 If *s* is a null pointer, the `mbrtoc16()` function shall be equivalent to the call:

```
3121 mbrtoc16(NULL, "", 1, ps)
```

3122 In this case, the values of the parameters *pc16* and *n* are ignored.

3123 If *s* is not a null pointer, the `mbrtoc16()` function shall inspect at most *n* bytes beginning with
3124 the byte pointed to by *s* to determine the number of bytes needed to complete the next
3125 character (including any shift sequences). If the function determines that the next character
3126 is complete and valid, it shall determine the values of the corresponding wide characters and
3127 then, if *pc16* is not a null pointer, shall store the value of the first (or only) such character in
3128 the object pointed to by *pc16*. Subsequent calls shall store successive wide characters
3129 without consuming any additional input until all the characters have been stored. If the
3130 corresponding wide character is the null wide character, the resulting state described shall be
3131 the initial conversion state.

3132 If *ps* is a null pointer, the `mbrtoc16()` function shall use its own internal **mbstate_t** object,
3133 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
3134 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current
3135 conversion state of the associated character sequence.

3136 The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

3137 The `mbrtoc16()` function shall not change the setting of `errno` if successful.

3138 The `mbrtoc32()` function shall behave the same way as `mbrtoc16()` except that the first
3139 parameter shall point to an object of type **char32_t** instead of **char16_t**. References to *pc16*
3140 in the above description shall apply as if they were *pc32* when they are being read as
3141 describing `mbrtoc32()`.

3142 If called with a null *ps* argument, the `mbrtoc16()` function need not be thread-safe; however,
3143 such calls shall avoid data races with calls to `mbrtoc16()` with a non-null argument and with
3144 calls to all other functions.

3145 If called with a null *ps* argument, the `mbrtoc32()` function need not be thread-safe; however,
3146 such calls shall avoid data races with calls to `mbrtoc32()` with a non-null argument and with
3147 calls to all other functions.

3148 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
3149 calls *mbrtoc16()* or *mbrtoc32()* with a null pointer for *ps*.

3150 RETURN VALUE

3151 These functions shall return the first of the following that applies:

3152 0 If the next *n* or fewer bytes complete the character that corresponds to the null
3153 wide character (which is the value stored).

3154 between 1 and *n* inclusive

3155 If the next *n* or fewer bytes complete a valid character (which is the value
3156 stored); the value returned shall be the number of bytes that complete the
3157 character.

3158 (*size_t*)-3 If the next character resulting from a previous call has been stored, in which
3159 case no bytes from the input shall be consumed by the call.

3160 (*size_t*)-2 If the next *n* bytes contribute to an incomplete but potentially valid character,
3161 and all *n* bytes have been processed (no value is stored). When *n* has at least
3162 the value of the {*MB_CUR_MAX*} macro, this case can only occur if *s*
3163 points at a sequence of redundant shift sequences (for implementations with
3164 state-dependent encodings).

3165 (*size_t*)-1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
3166 contribute to a complete and valid character (no value is stored). In this case,
3167 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

3168 ERRORS

3169 These function shall fail if:

3170 [EILSEQ] An invalid character sequence is detected. [CX]In the POSIX locale
3171 an [EILSEQ] error cannot occur since all byte values are valid
3172 characters.[/CX]

3173 These functions may fail if:

3174 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

3175 EXAMPLES

3176 None.

3177 APPLICATION USAGE

3178 None.

3179 RATIONALE

3180 None.

3181 FUTURE DIRECTIONS

3182 None.

3183 SEE ALSO

3184 *c16rtomb*

3185 XBD <uchar.h>

3186 **CHANGE HISTORY**

3187 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3188 Ref 7.29.6.3 para 1, 7.1.4 para 5

3189 On page 1301 line 43322 section `mbrtowc()`, change:

3190 [CX]The `mbrtowc()` function need not be thread-safe if called with a NULL *ps* argument.
3191 [/CX]

3192 to:

3193 If called with a null *ps* argument, the `mbrtowc()` function need not be thread-safe; however,
3194 such calls shall avoid data races with calls to `mbrtowc()` with a non-null argument and with
3195 calls to all other functions.

3196 Ref 7.29.6.4 para 1, 7.1.4 para 5

3197 On page 1304 line 43451 section `mbsrtowcs()`, change:

3198 [CX]The `mbsnrtowcs()` and `mbsrtowcs()` functions need not be thread-safe if called with a
3199 NULL *ps* argument.[/CX]

3200 to:

3201 [CX]If called with a null *ps* argument, the `mbsnrtowcs()` function need not be thread-safe;
3202 however, such calls shall avoid data races with calls to `mbsnrtowcs()` with a non-null
3203 argument and with calls to all other functions.[/CX]

3204 If called with a null *ps* argument, the `mbsrtowcs()` function need not be thread-safe; however,
3205 such calls shall avoid data races with calls to `mbsrtowcs()` with a non-null argument and with
3206 calls to all other functions.

3207 Ref 7.22.7 para 1

3208 On page 1308 line 43557 section `mbtowc()`, change:

3209 this function is placed into its initial state by a call for which

3210 to:

3211 this function shall be placed into its initial state at program startup and can be returned to
3212 that state by a call for which

3213 Ref 7.22.7 para 1, 7.1.4 para 5

3214 On page 1308 line 43567 section `mbtowc()`, change:

3215 [CX]The `mbtowc()` function need not be thread-safe.[/CX]

3216 to:

3217 The `mbtowc()` function need not be thread-safe; however, it shall avoid data races with all

3218 other functions.

3219 Ref 7.24.5.1 para 2

3220 On page 1311 line 43642 section `memchr()`, change:

3221 Implementations shall behave as if they read the memory byte by byte from the beginning of
3222 the bytes pointed to by *s* and stop at the first occurrence of *c* (if it is found in the initial *n*
3223 bytes).

3224 to:

3225 The implementation shall behave as if it reads the bytes sequentially and stops as soon as a
3226 matching byte is found.

3227 Ref F.10.3.12 para 2

3228 On page 1346 line 44854 section `modf()`, add a new paragraph:

3229 [MX]The returned value shall be exact and shall be independent of the current rounding
3230 direction mode.[/MX]

3231 Ref 7.26.4

3232 On page 1384 line 46032 insert the following new `mtx_*` sections:

3233 **NAME**

3234 `mtx_destroy`, `mtx_init` — destroy and initialize a mutex

3235 **SYNOPSIS**

3236 `#include <threads.h>`

3237 `void mtx_destroy(mtx_t *mtx);`
3238 `int mtx_init(mtx_t *mtx, int type);`

3239 **DESCRIPTION**

3240 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3241 Any conflict between the requirements described here and the ISO C standard is
3242 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3243 The `mtx_destroy()` function shall release any resources used by the mutex pointed to by *mtx*.
3244 A destroyed mutex object can be reinitialized using `mtx_init()`; the results of otherwise
3245 referencing the object after it has been destroyed are undefined. It shall be safe to destroy an
3246 initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that
3247 another thread is attempting to lock, or a mutex that is being used in a `cond_timedwait()` or
3248 `cond_wait()` call by another thread, results in undefined behavior. The behavior is undefined if
3249 the value specified by the *mtx* argument to `mtx_destroy()` does not refer to an initialized
3250 mutex.

3251 The `mtx_init()` function shall initialize a mutex object with properties indicated by *type*,
3252 whose valid values include:

3253 `mtx_plain` for a simple non-recursive mutex,

3254 `mtx_timed` for a non-recursive mutex that supports timeout,

3255 `mtx_plain` | `mtx_recursive` for a simple recursive mutex, or

3256 `mtx_timed` | `mtx_recursive` for a recursive mutex that supports timeout.

3257 If the `mtx_init()` function succeeds, it shall set the mutex pointed to by `mtx` to a value that
3258 uniquely identifies the newly initialized mutex. Upon successful initialization, the state of
3259 the mutex becomes initialized and unlocked. Attempting to initialize an already initialized
3260 mutex results in undefined behavior.

3261 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
3262 further requirements.

3263 These functions shall not be affected if the calling thread executes a signal handler during
3264 the call.[/CX]

3265 **RETURN VALUE**

3266 The `mtx_destroy()` function shall not return a value.

3267 The `mtx_init()` function shall return `thrd_success` on success or `thrd_error` if the
3268 request could not be honored.

3269 **ERRORS**

3270 No errors are defined.

3271 **EXAMPLES**

3272 None.

3273 **APPLICATION USAGE**

3274 A mutex can be destroyed immediately after it is unlocked. However, since attempting to
3275 destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that
3276 is being used in a `cond_timedwait()` or `cond_wait()` call by another thread results in undefined
3277 behavior, care must be taken to ensure that no other thread may be referencing the mutex.

3278 **RATIONALE**

3279 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3280 B.2.3].

3281 **FUTURE DIRECTIONS**

3282 None.

3283 **SEE ALSO**

3284 `mtx_lock`

3285 XBD <**threads.h**>

3286 **CHANGE HISTORY**

3287 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3288 **NAME**

3289 `mtx_lock`, `mtx_timedlock`, `mtx_trylock`, `mtx_unlock` — lock and unlock a mutex

3290 **SYNOPSIS**

```
3291     #include <threads.h>

3292     int mtx_lock(mtx_t *mtx);
3293     int mtx_timedlock(mtx_t * restrict mtx,
3294                      const struct timespec * restrict ts);
3295     int mtx_trylock(mtx_t *mtx);
3296     int mtx_unlock(mtx_t *mtx);
```

3297 **DESCRIPTION**

3298 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3299 Any conflict between the requirements described here and the ISO C standard is
3300 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3301 The *mtx_lock()* function shall block until it locks the mutex pointed to by *mtx*. If the mutex
3302 is non-recursive, the application shall ensure that it is not already locked by the calling
3303 thread.

3304 The *mtx_timedlock()* function shall block until it locks the mutex pointed to by *mtx* or until
3305 after the *TIME_UTC* -based calendar time pointed to by *ts*. The application shall ensure that
3306 the specified mutex supports timeout. [CX]Under no circumstance shall the function fail
3307 with a timeout if the mutex can be locked immediately. The validity of the *ts* parameter need
3308 not be checked if the mutex can be locked immediately.[/CX]

3309 The *mtx_trylock()* function shall endeavor to lock the mutex pointed to by *mtx*. If the mutex
3310 is already locked (by any thread, including the current thread), the function shall return
3311 without blocking. If the mutex is recursive and the mutex is currently owned by the calling
3312 thread, the mutex lock count (see below) shall be incremented by one and the *mtx_trylock()*
3313 function shall immediately return success.

3314 [CX]These functions shall not be affected if the calling thread executes a signal handler
3315 during the call; if a signal is delivered to a thread waiting for a mutex, upon return from the
3316 signal handler the thread shall resume waiting for the mutex as if it was not interrupted.
3317 [/CX]

3318 If a call to *mtx_lock()*, *mtx_timedlock()* or *mtx_trylock()* locks the mutex, prior calls to
3319 *mtx_unlock()* on the same mutex shall synchronize with this lock operation.

3320 The *mtx_unlock()* function shall unlock the mutex pointed to by *mtx* . The application shall
3321 ensure that the mutex pointed to by *mtx* is locked by the calling thread. [CX]If there are
3322 threads blocked on the mutex object referenced by *mtx* when *mtx_unlock()* is called,
3323 resulting in the mutex becoming available, the scheduling policy shall determine which
3324 thread shall acquire the mutex.[/CX]

3325 A recursive mutex shall maintain the concept of a lock count. When a thread successfully
3326 acquires a mutex for the first time, the lock count shall be set to one. Every time a thread
3327 relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks
3328 the mutex, the lock count shall be decremented by one. When the lock count reaches zero,
3329 the mutex shall become available for other threads to acquire.

3330 For purposes of determining the existence of a data race, mutex lock and unlock operations
3331 on mutexes of type **mtx_t** behave as atomic operations. All lock and unlock operations on a
3332 particular mutex occur in some particular total order.

3333 If *mtx* does not refer to an initialized mutex object, the behavior of these functions is
3334 undefined.

3335 RETURN VALUE

3336 The *mtx_lock()* and *mtx_unlock()* functions shall return `thrd_success` on success, or
3337 `thrd_error` if the request could not be honored.

3338 The *mtx_timedlock()* function shall return `thrd_success` on success, or `thrd_timedout`
3339 if the time specified was reached without acquiring the requested resource, or `thrd_error`
3340 if the request could not be honored.

3341 The *mtx_trylock()* function shall return `thrd_success` on success, or `thrd_busy` if the
3342 resource requested is already in use, or `thrd_error` if the request could not be honored.
3343 The *mtx_trylock()* function can spuriously fail to lock an unused resource, in which case it
3344 shall return `thrd_busy`.

3345 ERRORS

3346 See RETURN VALUE.

3347 EXAMPLES

3348 None.

3349 APPLICATION USAGE

3350 None.

3351 RATIONALE

3352 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3353 B.2.3].

3354 FUTURE DIRECTIONS

3355 None.

3356 SEE ALSO

3357 *mtx_destroy*, *timespec_get*

3358 XBD Section 4.12.2, <**threads.h**>

3359 CHANGE HISTORY

3360 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3361 Ref F.10.8.2 para 2

3362 On page 1388 line 46143 section `nan()`, add a new paragraph:

3363 [MX]The returned value shall be exact and shall be independent of the current rounding
3364 direction mode.[/MX]

3365 Ref F.10.8.3 para 2, F.10.8.4 para 2

3366 On page 1395 line 46388 section `nextafter()`, add a new paragraph:

3367 [MX]Even though underflow or overflow can occur, the returned value shall be independent
3368 of the current rounding direction mode.[/MX]

3369 Ref 7.22.3 para 2

3370 On page 1448 line 48069 section `posix_memalign()`, add a new (unshaded) paragraph:

3371 For purposes of determining the existence of a data race, `posix_memalign()` shall behave as
3372 though it accessed only memory locations accessible through its arguments and not other
3373 static duration storage. The function may, however, visibly modify the storage that it
3374 allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `posix_memalign()`, and `realloc()`
3375 that allocate or deallocate a particular region of memory shall occur in a single total order
3376 (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the next
3377 allocation (if any) in this order.

3378 Ref 7.22.3.1

3379 On page 1449 line 48107 section `posix_memalign()`, add `aligned_alloc` to the SEE ALSO section.

3380 Ref F.10.4.4 para 1

3381 On page 1548 line 50724 section `pow()`, change:

3382 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 , a pole error shall
3383 occur and `pow()`, `powf()`, and `powl()` shall return `±HUGE_VAL`, `±HUGE_VALF`, and
3384 `±HUGE_VALL`, respectively if y is an odd integer, or `HUGE_VAL`, `HUGE_VALF`, and
3385 `HUGE_VALL`, respectively if y is not an odd integer.

3386 to:

3387 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 :

- 3388 • if y is an odd integer, a pole error shall occur and `pow()`, `powf()`, and `powl()` shall
3389 return `±HUGE_VAL`, `±HUGE_VALF`, and `±HUGE_VALL`, respectively;
- 3390 • if y is finite and is not an odd integer, a pole error shall occur and `pow()`, `powf()`, and
3391 `powl()` shall return `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively;
- 3392 • if y is `-Inf`, a pole error may occur and `pow()`, `powf()`, and `powl()` shall return
3393 `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

3394 Ref 7.26

3395 On page 1603 line 52244 section `pthread_cancel()`, add a new paragraph:

3396 If `thread` refers to a thread that was created using `thrd_create()`, the behavior is undefined.

3397 Ref 7.26.5.6

3398 On page 1603 line 52277 section `pthread_cancel()`, add a new RATIONALE paragraph:

3399 Use of `pthread_cancel()` to cancel a thread that was created using `thrd_create()` is undefined
3400 because `thrd_join()` has no way to indicate a thread was cancelled. The standard developers
3401 considered adding a `thrd_canceled` enumeration constant that `thrd_join()` would return in
3402 this case. However, this return would be unexpected in code that is written to conform to the
3403 ISO C standard, and it would also not solve the problem that threads which use only ISO C
3404 `<threads.h>` interfaces (such as ones created by third party libraries written to conform to

3405 the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not
3406 provide cancellation cleanup handlers.

3407 Ref 7.26.5.5
3408 On page 1639 line 53422 section `pthread_exit()`, change:

3409 `void pthread_exit(void *value_ptr);`

3410 to:

3411 `_Noreturn void pthread_exit(void *value_ptr);`

3412 Ref 7.26.6
3413 On page 1639 line 53427 section `pthread_exit()`, change:

3414 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3415 specific data, appropriate destructor functions shall be called in an unspecified order.

3416 to:

3417 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3418 specific data (whether associated with key type `tss_t` or `pthread_key_t`), appropriate
3419 destructor functions shall be called in an unspecified order.

3420 Ref 7.26.5.5
3421 On page 1639 line 53432 section `pthread_exit()`, change:

3422 An implicit call to `pthread_exit()` is made when a thread other than the thread in which
3423 `main()` was first invoked returns from the start routine that was used to create it.

3424 to:

3425 An implicit call to `pthread_exit()` is made when a thread that was not created using
3426 `thrd_create()`, and is not the thread in which `main()` was first invoked, returns from the start
3427 routine that was used to create it.

3428 Ref 7.26.5.5
3429 On page 1639 line 53451 section `pthread_exit()`, change APPLICATION USAGE from:

3430 None.

3431 to:

3432 Calls to `pthread_exit()` should not be made from threads created using `thrd_create()`, as their
3433 exit status has a different type (`int` instead of `void *`). If `pthread_exit()` is called from the
3434 initial thread and it is not the last thread to terminate, other threads should not try to obtain
3435 its exit status using `thrd_join()`.

3436 Ref 7.26.5.5
3437 On page 1639 line 53453 section `pthread_exit()`, change:

3438 The normal mechanism by which a thread terminates is to return from the routine that was

3439 specified in the *pthread_create()* call that started it.

3440 to:

3441 The normal mechanism by which a thread that was started using *pthread_create()* terminates
3442 is to return from the routine that was specified in the *pthread_create()* call that started it.

3443 Ref 7.26.5.5, 7.26.6
3444 On page 1640 line 53470 section *pthread_exit()*, add *pthread_key_create*, *thrd_create*, *thrd_exit* and
3445 *tss_create* to the SEE ALSO section.

3446 Ref 7.26.5.5
3447 On page 1649 line 53748 section *pthread_join()*, add a new paragraph:

3448 If *thread* refers to a thread that was created using *thrd_create()* and the thread terminates, or
3449 has already terminated, by returning from its start routine, the behavior of *pthread_join()* is
3450 undefined. If *thread* refers to a thread that terminates, or has already terminated, by calling
3451 *thrd_exit()*, the behavior of *pthread_join()* is undefined.

3452 Ref 7.26.5.5
3453 On page 1651 line 53819 section *pthread_join()*, add a new RATIONALE paragraph:

3454 The *pthread_join()* function cannot be used to obtain the exit status of a thread that was
3455 created using *thrd_create()* and which terminates by returning from its start routine, or of a
3456 thread that terminates by calling *thrd_exit()*, because such threads have an **int** exit status,
3457 instead of the **void *** that *pthread_join()* returns via its *value_ptr* argument.

3458 Ref 7.22.4.7
3459 On page 1765 line 57040 insert the following new *quick_exit()* section:

3460 **NAME**
3461 *quick_exit* — terminate a process

3462 **SYNOPSIS**
3463 `#include <stdlib.h>`

3464 `_Noreturn void quick_exit(int status);`

3465 **DESCRIPTION**
3466 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3467 Any conflict between the requirements described here and the ISO C standard is
3468 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3469 The *quick_exit()* function shall cause normal process termination to occur. It shall not call
3470 functions registered with *atexit()* nor any registered signal handlers. If a process calls the
3471 *quick_exit()* function more than once, or calls the *exit()* function in addition to the
3472 *quick_exit()* function, the behavior is undefined. If a signal is raised while the *quick_exit()*
3473 function is executing, the behavior is undefined.

3474 The *quick_exit()* function shall first call all functions registered by *at_quick_exit()*, in the
3475 reverse order of their registration, except that a function is called after any previously
3476 registered functions that had already been called at the time it was registered. If, during the

3477 call to any such function, a call to the *longjmp()* [CX] or *siglongjmp()*[/CX] function is made
3478 that would terminate the call to the registered function, the behavior is undefined.

3479 If a function registered by a call to *at_quick_exit()* fails to return, the remaining registered
3480 functions shall not be called and the rest of the *quick_exit()* processing shall not be
3481 completed.

3482 Finally, the *quick_exit()* function shall terminate the process as if by a call to *_Exit(status)*.

3483 **RETURN VALUE**

3484 The *quick_exit()* function does not return.

3485 **ERRORS**

3486 No errors are defined.

3487 **EXAMPLES**

3488 None.

3489 **APPLICATION USAGE**

3490 None.

3491 **RATIONALE**

3492 None.

3493 **FUTURE DIRECTIONS**

3494 None.

3495 **SEE ALSO**

3496 *_Exit*, *at_quick_exit*, *atexit*, *exit*

3497 XBD <stdlib.h>

3498 **CHANGE HISTORY**

3499 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3500 Ref 7.22.2.1 para 3, 7.1.4 para 5

3501 On page 1767 line 57095 section *rand()*, change:

3502 [CX]The *rand()* function need not be thread-safe.[/CX]

3503 to:

3504 The *rand()* function need not be thread-safe; however, *rand()* shall avoid data races with all
3505 functions other than non-thread-safe pseudo-random sequence generation functions.

3506 Ref 7.22.2.2 para 3, 7.1.4 para 5

3507 On page 1767 line 57105 section *rand()*, add a new paragraph:

3508 The *srand()* function need not be thread-safe; however, *srand()* shall avoid data races with
3509 all functions other than non-thread-safe pseudo-random sequence generation functions.

3510 Ref 7.22.3 para 1,2; 7.22.3.5 para 2,3,4; 7.31.12 para 2
3511 On page 1788 line 57862-57892 section `realloc()`, replace the DESCRIPTION and RETURN
3512 VALUE sections with:

3513 **DESCRIPTION**

3514 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3515 Any conflict between the requirements described here and the ISO C standard is
3516 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3517 The `realloc()` function shall deallocate the old object pointed to by *ptr* and return a pointer to
3518 a new object that has the size specified by *size*. The contents of the new object shall be the
3519 same as that of the old object prior to deallocation, up to the lesser of the new and old sizes.
3520 Any bytes in the new object beyond the size of the old object have indeterminate values.

3521 If *ptr* is a null pointer, `realloc()` shall be equivalent to `malloc()` function for the specified
3522 size. Otherwise, if *ptr* does not match a pointer returned earlier by `aligned_alloc()`, `calloc()`,
3523 `malloc()`, `[ADV]posix_memalign()`,`[/ADV] realloc()`, or a function in POSIX.1-20xx that
3524 allocates memory as if by `malloc()`, or if the space has been deallocated by a call to `free()` or
3525 `realloc()`, the behavior is undefined.

3526 If *size* is non-zero and memory for the new object is not allocated, the old object shall not be
3527 deallocated. [OB]If *size* is zero and memory for the new object is not allocated, it is
3528 implementation-defined whether the old object is deallocated; if the old object is not
3529 deallocated, its value shall be unchanged.[/OB]

3530 The order and contiguity of storage allocated by successive calls to `realloc()` is unspecified.
3531 The pointer returned if the allocation succeeds shall be suitably aligned so that it may be
3532 assigned to a pointer to any type of object with a fundamental alignment requirement and
3533 then used to access such an object in the space allocated (until the space is explicitly freed or
3534 reallocated). Each such allocation shall yield a pointer to an object disjoint from any other
3535 object. The pointer returned shall point to the start (lowest byte address) of the allocated
3536 space. If the space cannot be allocated, a null pointer shall be returned. [OB]If the size of the
3537 space requested is 0, the behavior is implementation-defined: either a null pointer shall be
3538 returned to indicate an error, or the behavior shall be as if the size were some non-zero
3539 value, except that the behavior is undefined if the returned pointer is used to access an
3540 object.[/OB]

3541 For purposes of determining the existence of a data race, `realloc()` shall behave as though it
3542 accessed only memory locations accessible through its arguments and not other static
3543 duration storage. The function may, however, visibly modify the storage that it allocates or
3544 deallocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `[ADV]posix_memalign()`,
3545 `[/ADV]` and `realloc()` that allocate or deallocate a particular region of memory shall occur in
3546 a single total order (see [xref to XBD 4.12.1]), and each such deallocation call shall
3547 synchronize with the next allocation (if any) in this order.

3548 **RETURN VALUE**

3549 The `realloc()` function shall return a pointer to the new object (which can have the same
3550 value as a pointer to the old object), or a null pointer if the new object has not been
3551 allocated.

3552 [OB]If *size* is zero, either:

- 3553 • A null pointer shall be returned [CX]and, if *ptr* is not a null pointer, *errno* shall be set
3554 to an implementation-defined value.[/CX]
3555 • A pointer to the allocated space shall be returned, and the memory object pointed to
3556 by *ptr* shall be freed. The application shall ensure that the pointer is not used to
3557 access an object.[/OB]

3558 If there is not enough available memory, *realloc()* shall return a null pointer [CX]and set
3559 *errno* to [ENOMEM][/CX].

3560 Ref 7.22.3.5 para 3,4

3561 On page 1789 line 57899 section *realloc()*, change:

3562 The description of *realloc()* has been modified from previous versions of this standard to
3563 align with the ISO/IEC 9899: 1999 standard. Previous versions explicitly permitted a call to
3564 *realloc(p, 0)* to free the space pointed to by *p* and return a null pointer. While this behavior
3565 could be interpreted as permitted by this version of the standard, the C language committee
3566 have indicated that this interpretation is incorrect. Applications should assume that if
3567 *realloc()* returns a null pointer, the space pointed to by *p* has not been freed. Since this could
3568 lead to double-frees, implementations should also set *errno* if a null pointer actually
3569 indicates a failure, and applications should only free the space if *errno* was changed.

3570 to:

3571 The ISO C standard makes it implementation-defined whether a call to *realloc(p, 0)* frees the
3572 space pointed to by *p* if it returns a null pointer because memory for the new object was not
3573 allocated. POSIX.1 instead requires that implementations set *errno* if a null pointer is
3574 returned and the space has not been freed, and POSIX applications should only free the
3575 space if *errno* was changed.

3576 Ref 7.31.12 para 2

3577 On page 1789 line 57909-57912 section *realloc()*, change FUTURE DIRECTIONS to:

3578 The ISO C standard states that invoking *realloc()* with a *size* argument equal to zero is an
3579 obsolescent feature. This feature may be removed in a future version of this standard.

3580 Ref 7.22.3.1

3581 On page 1789 line 57914 section *realloc()*, add *aligned_alloc* to the SEE ALSO section.

3582 Ref F.10.7.2 para 2

3583 On page 1809 line 58638 section *remainder()*, add a new paragraph:

3584 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3585 Ref F.10.7.3 para 2

3586 On page 1814 line 58758 section *remquo()*, add a new paragraph:

3587 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3588 Ref F.10.6.6 para 3

3589 On page 1828 line 59258 section *round()*, add a new paragraph:

3590 [MX]These functions may raise the inexact floating-point exception for finite non-integer

3591 arguments.[/MX]

3592 Ref F.10.6.6 para 3
3593 On page 1828 line 59272 section round(), delete from APPLICATION USAGE:

3594 These functions may raise the inexact floating-point exception if the result differs in value
3595 from the argument.

3596 Ref F.10.3.13 para 2
3597 On page 1829 line 59306 section scalbln(), add a new paragraph:

3598 [MX]If the calculation does not overflow or underflow, the returned value shall be exact and
3599 shall be independent of the current rounding direction mode.[/MX]

3600 Ref 7.11.1.1 para 5
3601 On page 1903 line 61520 section setlocale(), remove the CX shading from:

3602 The *setlocale()* function need not be thread-safe; however, it shall avoid data races with all
3603 function calls that do not affect and are not affected by the global locale.

3604 Ref 7.13.2.1 para 1
3605 On page 1970 line 63497 section siglongjmp(), change:

3606 `void siglongjmp(sigjmp_buf env, int val);`

3607 to:

3608 `_Noreturn void siglongjmp(sigjmp_buf env, int val);`

3609 Ref 7.13.2.1 para 4
3610 On page 1970 line 63504 section siglongjmp(), change:

3611 *After siglongjmp() is completed, program execution shall continue ...*

3612 to:

3613 *After siglongjmp() is completed, thread execution shall continue ...*

3614 Ref 7.14.1.1 para 5
3615 On page 1971 line 63564 section signal(), change:

3616 with static storage duration

3617 to:

3618 with static or thread storage duration that is not a lock-free atomic object

3619 Ref F.10.4.5 para 1
3620 On page 2009 line 64624 section sqrt(), add:

3621 [MX]The returned value shall be dependent on the current rounding direction mode.[/MX]

3622 Ref 7.24.6.2 para 3, 7.1.4 para 5
3623 On page 2035 line 65231 section `strerror()`, change:

3624 [CX]The `strerror()` function need not be thread-safe.[/CX]

3625 to:

3626 The `strerror()` function need not be thread-safe; however, `strerror()` shall avoid data races
3627 with all other functions.

3628 Ref 7.22.1.3 para 10
3629 On page 2073 line 66514 section `strtod()`, change:

3630 If the correct value is outside the range of representable values

3631 to:

3632 If the correct value would cause an overflow and default rounding is in effect

3633 Ref 7.24.5.8 para 6, 7.1.4 para 5
3634 On page 2078 line 66674 section `strtok()`, change:

3635 [CX]The `strtok()` function need not be thread-safe.[/CX]

3636 to:

3637 The `strtok()` function need not be thread-safe; however, `strtok()` shall avoid data races with
3638 all other functions.

3639 Ref 7.22.4.8, 7.1.4 para 5
3640 On page 2107 line 67579 section `system()`, change:

3641 The `system()` function need not be thread-safe.

3642 to:

3643 [CX]If concurrent calls to `system()` are made from multiple threads, it is unspecified
3644 whether:

- 3645 • each call saves and restores the dispositions of the SIGINT and SIGQUIT signals
3646 independently, or
- 3647 • in a set of concurrent calls the dispositions in effect after the last call returns are
3648 those that were in effect on entry to the first call.

3649 If a thread is cancelled while it is in a call to `system()`, it is unspecified whether the child
3650 process is terminated and waited for, or is left running.[/CX]

3651 Ref 7.22.4.8, 7.1.4 para 5
3652 On page 2108 line 67627 section `system()`, change:

3653 Using the `system()` function in more than one thread in a process or when the SIGCHLD
3654 signal is being manipulated by more than one thread in a process may produce unexpected
3655 results.

3656 to:

3657 Although *system()* is required to be thread-safe, it is recommended that concurrent calls
3658 from multiple threads are avoided, since *system()* is not required to coordinate the saving
3659 and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of
3660 overlapping calls, and therefore the signals might end up being set to ignored after the last
3661 call returns. Applications should also avoid cancelling a thread while it is in a call to
3662 *system()* as the child process may be left running in that event. In addition, if another thread
3663 alters the disposition of the SIGCHLD signal, a call to *signal()* may produce unexpected
3664 results.

3665 Ref 7.22.4.8, 7.1.4 para 5

3666 On page 2109 line 67675 section *system()*, delete:

```
3667     #include <signal.h>
```

3668 Ref 7.22.4.8, 7.1.4 para 5

3669 On page 2109 line 67692,67696,67712 section *system()*, change *sigprocmask* to
3670 *pthread_sigmask*.

3671 Ref 7.22.4.8, 7.1.4 para 5

3672 On page 2110 line 67718 section *system()*, change:

3673 Note also that the above example implementation is not thread-safe. Implementations can
3674 provide a thread-safe *system()* function, but doing so involves complications such as how to
3675 restore the signal dispositions for SIGINT and SIGQUIT correctly if there are overlapping
3676 calls, and how to deal with cancellation. The example above would not restore the signal
3677 dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-
3678 safe implementation since canceling a non-thread-safe function results in undefined behavior
3679 (see Section 2.9.5.2, on page 518). To avoid leaking a process ID, a thread-safe
3680 implementation would need to terminate the child process when acting on a cancellation.

3681 to:

3682 Earlier versions of this standard did not require *system()* to be thread-safe because it alters
3683 the process-wide disposition of the SIGINT and SIGQUIT signals. It is now required to be
3684 thread-safe to align with the ISO C standard, which (since the introduction of threads in
3685 2011) requires that it avoids data races. However, the function is not required to coordinate
3686 the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set
3687 of overlapping calls, and the above example does not do so. The example also does not
3688 terminate and wait for the child process if the calling thread is cancelled, and so would leak
3689 a process ID in that event.

3690 Ref 7.26.5

3691 On page 2148 line 68796 insert the following new *thrd_**() sections:

3692 **NAME**

3693 *thrd_create* — thread creation

3694 **SYNOPSIS**

```
3695     #include <threads.h>
```

```
3696     int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

3697 **DESCRIPTION**

3698 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3699 Any conflict between the requirements described here and the ISO C standard is
3700 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3701 The *thrd_create()* function shall create a new thread executing *func(arg)*. If the *thrd_create()*
3702 function succeeds, it shall set the object pointed to by *thr* to the identifier of the newly
3703 created thread. (A thread's identifier might be reused for a different thread once the original
3704 thread has exited and either been detached or joined to another thread.) The completion of
3705 the *thrd_create()* function shall synchronize with the beginning of the execution of the new
3706 thread.

3707 [CX]The signal state of the new thread shall be initialized as follows:

- 3708 • The signal mask shall be inherited from the creating thread.
- 3709 • The set of signals pending for the new thread shall be empty.

3710 The thread-local current locale shall not be inherited from the creating thread.

3711 The floating-point environment shall be inherited from the creating thread.[/CX]

3712 [XSI] The alternate stack shall not be inherited from the creating thread.[/XSI]

3713 Returning from *func* shall have the same behavior as invoking *thrd_exit()* with the value
3714 returned from *func*.

3715 If *thrd_create()* fails, no new thread shall be created and the contents of the location
3716 referenced by *thr* are undefined.

3717 [CX]The *thrd_create()* function shall not be affected if the calling thread executes a signal
3718 handler during the call.[/CX]

3719 **RETURN VALUE**

3720 The *thrd_create()* function shall return *thrd_success* on success; or *thrd_nomem* if no
3721 memory could be allocated for the thread requested; or *thrd_error* if the request could not
3722 be honored, [CX]such as if the system-imposed limit on the total number of threads in a
3723 process {*PTHREAD_THREADS_MAX*} would be exceeded.[/CX]

3724 **ERRORS**

3725 See RETURN VALUE.

3726 **EXAMPLES**

3727 None.

3728 **APPLICATION USAGE**

3729 There is no requirement on the implementation that the ID of the created thread be available
3730 before the newly created thread starts executing. The calling thread can obtain the ID of the
3731 created thread through the *thr* argument of the *thrd_create()* function, and the newly created
3732 thread can obtain its ID by a call to *thrd_current()*.

3733 **RATIONALE**
3734 The *thrd_create()* function is not affected by signal handlers for the reasons stated in [xref to
3735 XRAT B.2.3].

3736 **FUTURE DIRECTIONS**
3737 None.

3738 **SEE ALSO**
3739 *pthread_create*, *thrd_current*, *thrd_detach*, *thrd_exit*, *thrd_join*

3740 XBD Section 4.12.2, <**threads.h**>

3741 **CHANGE HISTORY**
3742 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3743 **NAME**
3744 *thrd_current* — get the calling thread ID

3745 **SYNOPSIS**
3746 #include <threads.h>

3747 thrd_t thrd_current(void);

3748 **DESCRIPTION**
3749 [*CX*] The functionality described on this reference page is aligned with the ISO C standard.
3750 Any conflict between the requirements described here and the ISO C standard is
3751 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/*CX*]

3752 The *thrd_current()* function shall identify the thread that called it.

3753 **RETURN VALUE**
3754 The *thrd_current()* function shall return the thread ID of the thread that called it.

3755 The *thrd_current()* function shall always be successful. No return value is reserved to
3756 indicate an error.

3757 **ERRORS**
3758 No errors are defined.

3759 **EXAMPLES**
3760 None.

3761 **APPLICATION USAGE**
3762 None.

3763 **RATIONALE**
3764 None.

3765 **FUTURE DIRECTIONS**
3766 None.

3767 **SEE ALSO**

3768 *pthread_self, thrd_create, thrd_equal*

3769 XBD Section 4.12.2, <**threads.h**>

3770 **CHANGE HISTORY**

3771 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3772 **NAME**

3773 `thrd_detach` — detach a thread

3774 **SYNOPSIS**

3775 `#include <threads.h>`

3776 `int thrd_detach(thrd_t thr);`

3777 **DESCRIPTION**

3778 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3779 Any conflict between the requirements described here and the ISO C standard is
3780 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3781 The `thrd_detach()` function shall change the thread *thr* from joinable to detached, indicating
3782 to the implementation that any resources allocated to the thread can be reclaimed when that
3783 thread terminates. The application shall ensure that the thread identified by *thr* has not been
3784 previously detached or joined with another thread.

3785 [CX]The `thrd_detach()` function shall not be affected if the calling thread executes a signal
3786 handler during the call.[/CX]

3787 **RETURN VALUE**

3788 The `thrd_detach()` function shall return `thrd_success` on success or `thrd_error` if the
3789 request could not be honored.

3790 **ERRORS**

3791 No errors are defined.

3792 **EXAMPLES**

3793 None.

3794 **APPLICATION USAGE**

3795 None.

3796 **RATIONALE**

3797 The `thrd_detach()` function is not affected by signal handlers for the reasons stated in [xref
3798 to XRAT B.2.3].

3799 **FUTURE DIRECTIONS**

3800 None.

3801 **SEE ALSO**

3802 *pthread_detach, thrd_create, thrd_join*

3803 XBD <**threads.h**>

3804 **CHANGE HISTORY**

3805 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3806 **NAME**

3807 `thrd_equal` — compare thread IDs

3808 **SYNOPSIS**

3809 `#include <threads.h>`

3810 `int thrd_equal(thrd_t thr0, thrd_t thr1);`

3811 **DESCRIPTION**

3812 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3813 Any conflict between the requirements described here and the ISO C standard is
3814 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3815 The `thrd_equal()` function shall determine whether the thread identified by `thr0` refers to the
3816 thread identified by `thr1`.

3817 [CX]The `thrd_equal()` function shall not be affected if the calling thread executes a signal
3818 handler during the call.[/CX]

3819 **RETURN VALUE**

3820 The `thrd_equal()` function shall return a non-zero value if `thr0` and `thr1` are equal; otherwise,
3821 zero shall be returned.

3822 If either `thr0` or `thr1` is not a valid thread ID [CX]and is not equal to `PTHREAD_NULL`
3823 (which is defined in `<pthread.h>`)[/CX], the behavior is undefined.

3824 **ERRORS**

3825 No errors are defined.

3826 **EXAMPLES**

3827 None.

3828 **APPLICATION USAGE**

3829 None.

3830 **RATIONALE**

3831 See the RATIONALE section for `pthread_equal()`.

3832 The `thrd_equal()` function is not affected by signal handlers for the reasons stated in [xref to
3833 XRAT B.2.3].

3834 **FUTURE DIRECTIONS**

3835 None.

3836 **SEE ALSO**

3837 `pthread_equal`, `thrd_current`

3838 XBD `<pthread.h>`, `<threads.h>`

3839 **CHANGE HISTORY**

3840 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3841 **NAME**

3842 `thrd_exit` — thread termination

3843 **SYNOPSIS**

3844 `#include <threads.h>`

3845 `_Noreturn void thrd_exit(int res);`

3846 **DESCRIPTION**

3847 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3848 Any conflict between the requirements described here and the ISO C standard is
3849 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3850 For every thread-specific storage key [CX](regardless of whether it has type `tss_t` or
3851 `pthread_key_t`)[/CX] which was created with a non-null destructor and for which the value
3852 is non-null, `thrd_exit()` shall set the value associated with the key to a null pointer value and
3853 then invoke the destructor with its previous value. The order in which destructors are
3854 invoked is unspecified.

3855 If after this process there remain keys with both non-null destructors and values, the
3856 implementation shall repeat this process up to [CX]
3857 `{PTHREAD_DESTRUCTOR_ITERATIONS}`[/CX] times.

3858 Following this, the `thrd_exit()` function shall terminate execution of the calling thread and
3859 shall set its exit status to `res`. [CX]Thread termination shall not release any application
3860 visible process resources, including, but not limited to, mutexes and file descriptors, nor
3861 shall it perform any process-level cleanup actions, including, but not limited to, calling any
3862 `atexit()` routines that might exist.[/CX]

3863 An implicit call to `thrd_exit()` is made when a thread that was created using `thrd_create()`
3864 returns from the start routine that was used to create it (see [xref to `thrd_create()`]).

3865 [CX]The behavior of `thrd_exit()` is undefined if called from a destructor function that was
3866 invoked as a result of either an implicit or explicit call to `thrd_exit()`.[/CX]

3867 The process shall exit with an exit status of zero after the last thread has been terminated.
3868 The behavior shall be as if the implementation called `exit()` with a zero argument at thread
3869 termination time.

3870 **RETURN VALUE**

3871 This function shall not return a value.

3872 **ERRORS**

3873 No errors are defined.

3874 **EXAMPLES**

3875 None.

3876 **APPLICATION USAGE**

3877 Calls to *thrd_exit()* should not be made from threads created using *pthread_create()* or via a
3878 SIGEV_THREAD notification, as their exit status has a different type (**void *** instead of
3879 **int**). If *thrd_exit()* is called from the initial thread and it is not the last thread to terminate,
3880 other threads should not try to obtain its exit status using *pthread_join()*.

3881 **RATIONALE**

3882 The normal mechanism by which a thread that was started using *thrd_create()* terminates is
3883 to return from the function that was specified in the *thrd_create()* call that started it. The
3884 *thrd_exit()* function provides the capability for such a thread to terminate without requiring a
3885 return from the start routine of that thread, thereby providing a function analogous to *exit()*.

3886 Regardless of the method of thread termination, the destructors for any existing thread-
3887 specific data are executed.

3888 **FUTURE DIRECTIONS**

3889 None.

3890 **SEE ALSO**

3891 *exit*, *pthread_create*, *thrd_join*

3892 XBD <**threads.h**>

3893 **CHANGE HISTORY**

3894 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3895 **NAME**

3896 *thrd_join* — wait for thread termination

3897 **SYNOPSIS**

3898 #include <threads.h>

3899 int thrd_join(thrd_t *thr*, int **res*);

3900 **DESCRIPTION**

3901 [**CX**] The functionality described on this reference page is aligned with the ISO C standard.
3902 Any conflict between the requirements described here and the ISO C standard is
3903 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/**CX**]

3904 The *thrd_join()* function shall join the thread identified by *thr* with the current thread by
3905 blocking until the other thread has terminated. If the parameter *res* is not a null pointer,
3906 *thrd_join()* shall store the thread's exit status in the integer pointed to by *res*. The
3907 termination of the other thread shall synchronize with the completion of the *thrd_join()*
3908 function. The application shall ensure that the thread identified by *thr* has not been
3909 previously detached or joined with another thread.

3910 The results of multiple simultaneous calls to *thrd_join()* specifying the same target thread
3911 are undefined.

3912 The behavior is undefined if the value specified by the *thr* argument to *thrd_join()* refers to
3913 the calling thread.

3914 [CX]It is unspecified whether a thread that has exited but remains unjoined counts against
3915 {PTHREAD_THREADS_MAX}.

3916 If *thr* refers to a thread that was created using *pthread_create()* or via a SIGEV_THREAD
3917 notification and the thread terminates, or has already terminated, by returning from its start
3918 routine, the behavior of *thrd_join()* is undefined. If *thr* refers to a thread that terminates, or
3919 has already terminated, by calling *pthread_exit()* or by being cancelled, the behavior of
3920 *thrd_join()* is undefined.

3921 The *thrd_join()* function shall not be affected if the calling thread executes a signal handler
3922 during the call.[/CX]

3923 RETURN VALUE

3924 The *thrd_join()* function shall return `thrd_success` on success or `thrd_error` if the
3925 request could not be honored.

3926 [CX]It is implementation-defined whether *thrd_join()* detects deadlock situations; if it does
3927 detect them, it shall return `thrd_error` when one is detected.[/CX]

3928 ERRORS

3929 See RETURN VALUE.

3930 EXAMPLES

3931 None.

3932 APPLICATION USAGE

3933 None.

3934 RATIONALE

3935 The *thrd_join()* function provides a simple mechanism allowing an application to wait for a
3936 thread to terminate. After the thread terminates, the application may then choose to clean up
3937 resources that were used by the thread. For instance, after *thrd_join()* returns, any
3938 application-provided stack storage could be reclaimed.

3939 The *thrd_join()* or *thrd_detach()* function should eventually be called for every thread that is
3940 created using *thrd_create()* so that storage associated with the thread may be reclaimed.

3941 The *thrd_join()* function cannot be used to obtain the exit status of a thread that was created
3942 using *pthread_create()* or via a SIGEV_THREAD notification and which terminates by
3943 returning from its start routine, or of a thread that terminates by calling *pthread_exit()*,
3944 because such threads have a **void *** exit status, instead of the **int** that *thrd_join()* returns via
3945 its *res* argument.

3946 The *thrd_join()* function cannot be used to obtain the exit status of a thread that terminates
3947 by being cancelled because it has no way to indicate that a thread was cancelled. (The
3948 *pthread_join()* function does this by returning a reserved **void *** exit status; it is not possible
3949 to reserve an **int** value for this purpose without introducing a conflict with the ISO C
3950 standard.) The standard developers considered adding a `thrd_cancelled` enumeration
3951 constant that *thrd_join()* would return in this case. However, this return would be
3952 unexpected in code that is written to conform to the ISO C standard, and it would also not
3953 solve the problem that threads which use only ISO C **<threads.h>** interfaces (such as ones
3954 created by third party libraries written to conform to the ISO C standard) have no way to
3955 handle being cancelled, as the ISO C standard does not provide cancellation cleanup
3956 handlers.

3957 The *thrd_join()* function is not affected by signal handlers for the reasons stated in [xref to
3958 XRAT B.2.3].

3959 FUTURE DIRECTIONS

3960 None.

3961 SEE ALSO

3962 *pthread_create*, *pthread_exit*, *pthread_join*, *thrd_create*, *thrd_exit*

3963 XBD Section 4.12.2, <**threads.h**>

3964 CHANGE HISTORY

3965 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3966 NAME

3967 *thrd_sleep* — suspend execution for an interval

3968 SYNOPSIS

3969 #include <threads.h>

```
3970 int thrd_sleep(const struct timespec *duration,  
3971               struct timespec *remaining);
```

3972 DESCRIPTION

3973 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3974 Any conflict between the requirements described here and the ISO C standard is
3975 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3976 The *thrd_sleep()* function shall suspend execution of the calling thread until either the
3977 interval specified by *duration* has elapsed or a signal is delivered to the calling thread whose
3978 action is to invoke a signal-catching function or to terminate the process. If interrupted by a
3979 signal and the *remaining* argument is not null, the amount of time remaining (the requested
3980 interval minus the time actually slept) shall be stored in the interval it points to. The
3981 *duration* and *remaining* arguments can point to the same object.

3982 The suspension time may be longer than requested because the interval is rounded up to an
3983 integer multiple of the sleep resolution or because of the scheduling of other activity by the
3984 system. But, except for the case of being interrupted by a signal, the suspension time shall
3985 not be less than that specified, as measured by the system clock `TIME_UTC`.

3986 RETURN VALUE

3987 The *thrd_sleep()* function shall return zero if the requested time has elapsed, -1 if it has been
3988 interrupted by a signal, or a negative value (which may also be -1) if it fails for any other
3989 reason. [CX]If it returns a negative value, it shall set *errno* to indicate the error.[/CX]

3990 ERRORS

3991 [CX]The *thrd_sleep()* function shall fail if:

3992 [EINTR]

3993 The *thrd_sleep()* function was interrupted by a signal.

3994 [EINVAL]

3995 The *duration* argument specified a nanosecond value less than zero or greater than or
3996 equal to 1000 million.[/CX]

3997 **EXAMPLES**

3998 None.

3999 **APPLICATION USAGE**

4000 Since the return value may be -1 for errors other than [EINTR], applications should examine
4001 *errno* to distinguish [EINTR] from other errors (and thus determine whether the unslept time
4002 is available in the interval pointed to by *remaining*).

4003 **RATIONALE**

4004 The *thrd_sleep()* function is identical to the *nanosleep()* function except that the return value
4005 may be any negative value when it fails with an error other than [EINTR].

4006 **FUTURE DIRECTIONS**

4007 None.

4008 **SEE ALSO**

4009 *nanosleep*

4010 XBD <**threads.h**>, <**time.h**>

4011 **CHANGE HISTORY**

4012 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4013 **NAME**

4014 thrd_yield — yield the processor

4015 **SYNOPSIS**

4016 #include <threads.h>

4017 void thrd_yield(void);

4018 **DESCRIPTION**

4019 [CX] The functionality described on this reference page is aligned with the ISO C standard.

4020 Any conflict between the requirements described here and the ISO C standard is

4021 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4022 [CX]The *thrd_yield()* function shall force the running thread to relinquish the processor until

4023 it again becomes the head of its thread list.[/CX]

4024 **RETURN VALUE**

4025 This function shall not return a value.

4026 **ERRORS**

4027 No errors are defined.

4028 **EXAMPLES**

4029 None.

4030 **APPLICATION USAGE**

4031 See the APPLICATION USAGE section for *sched_yield()*.

4032 **RATIONALE**

4033 The *thrd_yield()* function is identical to the *sched_yield()* function except that it does not
4034 return a value.

4035 **FUTURE DIRECTIONS**

4036 None.

4037 **SEE ALSO**

4038 *sched_yield*

4039 XBD <**threads.h**>

4040 **CHANGE HISTORY**

4041 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4042 Ref 7.27.2.5

4043 On page 2161 line 69278 insert a new *timespec_get()* section:

4044 **NAME**

4045 *timespec_get* — get time

4046 **SYNOPSIS**

4047 `#include <time.h>`

4048 `int timespec_get(struct timespec *ts, int base);`

4049 **DESCRIPTION**

4050 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4051 Any conflict between the requirements described here and the ISO C standard is
4052 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4053 The *timespec_get()* function shall set the interval pointed to by *ts* to hold the current
4054 calendar time based on the specified time base.

4055 [CX]If *base* is `TIME_UTC`, the members of *ts* shall be set to the same values as would be
4056 set by a call to *clock_gettime(CLOCK_REALTIME, ts)*. If the number of seconds will not
4057 fit in an object of type **time_t**, the function shall return zero.[/CX]

4058 **RETURN VALUE**

4059 If the *timespec_get()* function is successful it shall return the non-zero value *base*; otherwise,
4060 it shall return zero.

4061 **ERRORS**

4062 See DESCRIPTION.

4063 **EXAMPLES**

4064 None.

4065 **APPLICATION USAGE**

4066 None.

4067 **RATIONALE**

4068 None.

4069 **FUTURE DIRECTIONS**

4070 None.

4071 **SEE ALSO**

4072 *clock_getres, time*

4073 XBD <**time.h**>

4074 **CHANGE HISTORY**

4075 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4076 Ref 7.21.4.4 para 4, 7.1.4 para 5

4077 On page 2164 line 69377 section *tmpnam()*, change:

4078 [CX]The *tmpnam()* function need not be thread-safe if called with a NULL parameter.[/CX]

4079 to:

4080 If called with a null pointer argument, the *tmpnam()* function need not be thread-safe;
4081 however, such calls shall avoid data races with calls to *tmpnam()* with a non-null argument
4082 and with calls to all other functions.

4083 Ref 7.30.3.2.1 para 4

4084 On page 2171 line 69568 section *towctrans()*, change:

4085 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped
4086 value of *wc* using the mapping described by *desc*. Otherwise, they shall return *wc*
4087 unchanged.

4088 to:

4089 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped
4090 value of *wc* using the mapping described by *desc*, or the value of *wc* unchanged if *desc* is
4091 zero. [CX]Otherwise, they shall return *wc* unchanged.[/CX]

4092 Ref F.10.6.8 para 2

4093 On page 2177 line 69716 section *trunc()*, add a new paragraph:

4094 [MX]These functions may raise the inexact floating-point exception for finite non-integer
4095 arguments.[/MX]

4096 Ref F.10.6.8 para 1,2

4097 On page 2177 line 69719 section *trunc()*, change:

4098 [MX]The result shall have the same sign as *x*.[/MX]

4099 to:

4100 [MX]The returned value shall be exact, shall be independent of the current rounding
4101 direction mode, and shall have the same sign as *x*.[/MX]

4102 Ref F.10.6.8 para 2

4103 On page 2177 line 69730 section `trunc()`, delete from APPLICATION USAGE:

4104 These functions may raise the inexact floating-point exception if the result differs in value
4105 from the argument.

4106 Ref 7.26.6

4107 On page 2182 line 69835 insert the following new `tss_*` sections:

4108 **NAME**

4109 `tss_create` — thread-specific data key creation

4110 **SYNOPSIS**

4111 `#include <threads.h>`

4112 `int tss_create(tss_t *key, tss_dtor_t dtor);`

4113 **DESCRIPTION**

4114 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4115 Any conflict between the requirements described here and the ISO C standard is
4116 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4117 The `tss_create()` function shall create a thread-specific storage pointer with destructor *dtor*,
4118 which can be null.

4119 A null pointer value shall be associated with the newly created key in all existing threads.
4120 Upon subsequent thread creation, the value associated with all keys shall be initialized to a
4121 null pointer value in the new thread.

4122 Destructors associated with thread-specific storage shall not be invoked at process
4123 termination.

4124 The behavior is undefined if the `tss_create()` function is called from within a destructor.

4125 [CX]The `tss_create()` function shall not be affected if the calling thread executes a signal
4126 handler during the call.[/CX]

4127 **RETURN VALUE**

4128 If the `tss_create()` function is successful, it shall set the thread-specific storage pointed to by
4129 *key* to a value that uniquely identifies the newly created pointer and shall return
4130 `thrd_success`; otherwise, `thrd_error` shall be returned and the thread-specific storage
4131 pointed to by *key* has an indeterminate value.

4132 **ERRORS**

4133 No errors are defined.

4134 **EXAMPLES**

4135 None.

4136 **APPLICATION USAGE**

4137 The *tss_create()* function performs no implicit synchronization. It is the responsibility of the
4138 programmer to ensure that it is called exactly once per key before use of the key.

4139 **RATIONALE**

4140 If the value associated with a key needs to be updated during the lifetime of the thread, it
4141 may be necessary to release the storage associated with the old value before the new value is
4142 bound. Although the *tss_set()* function could do this automatically, this feature is not needed
4143 often enough to justify the added complexity. Instead, the programmer is responsible for
4144 freeing the stale storage:

```
4145     old = tss_get(key);  
4146     new = allocate();  
4147     destructor(old);  
4148     tss_set(key, new);
```

4149 There is no notion of a destructor-safe function. If an application does not call *thrd_exit()* or
4150 *pthread_exit()* from a signal handler, or if it blocks any signal whose handler may call
4151 *thrd_exit()* or *pthread_exit()* while calling async-unsafe functions, all functions can be safely
4152 called from destructors.

4153 The *tss_create()* function is not affected by signal handlers for the reasons stated in [xref to
4154 XRAT B.2.3].

4155 **FUTURE DIRECTIONS**

4156 None.

4157 **SEE ALSO**

4158 *pthread_exit*, *pthread_key_create*, *thrd_exit*, *tss_delete*, *tss_get*

4159 XBD <**threads.h**>

4160 **CHANGE HISTORY**

4161 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4162 **NAME**

4163 *tss_delete* — thread-specific data key deletion

4164 **SYNOPSIS**

```
4165     #include <threads.h>
```

```
4166     void tss_delete(tss_t key);
```

4167 **DESCRIPTION**

4168 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4169 Any conflict between the requirements described here and the ISO C standard is
4170 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4171 The *tss_delete()* function shall release any resources used by the thread-specific storage
4172 identified by *key*. The thread-specific data values associated with *key* need not be null at the
4173 time *tss_delete()* is called. It is the responsibility of the application to free any application
4174 storage or perform any cleanup actions for data structures related to the deleted key or

4175 associated thread-specific data in any threads; this cleanup can be done either before or after
4176 *tss_delete()* is called.

4177 The application shall ensure that the *tss_delete()* function is only called with a value for *key*
4178 that was returned by a call to *tss_create()* before the thread commenced executing
4179 destructors.

4180 If *tss_delete()* is called while another thread is executing destructors, whether this will affect
4181 the number of invocations of the destructor associated with *key* on that thread is unspecified.

4182 The *tss_delete()* function shall be callable from within destructor functions. Calling
4183 *tss_delete()* shall not result in the invocation of any destructors. Any destructor function that
4184 was associated with *key* shall no longer be called upon thread exit.

4185 Any attempt to use *key* following the call to *tss_delete()* results in undefined behavior.

4186 [CX]The *tss_delete()* function shall not be affected if the calling thread executes a signal
4187 handler during the call.[/CX]

4188 **RETURN VALUE**

4189 This function shall not return a value.

4190 **ERRORS**

4191 No errors are defined.

4192 **EXAMPLES**

4193 None.

4194 **APPLICATION USAGE**

4195 None.

4196 **RATIONALE**

4197 A thread-specific data key deletion function has been included in order to allow the
4198 resources associated with an unused thread-specific data key to be freed. Unused thread-
4199 specific data keys can arise, among other scenarios, when a dynamically loaded module that
4200 allocated a key is unloaded.

4201 Conforming applications are responsible for performing any cleanup actions needed for data
4202 structures associated with the key to be deleted, including data referenced by thread-specific
4203 data values. No such cleanup is done by *tss_delete()*. In particular, destructor functions
4204 are not called. See the RATIONALE for *pthread_key_delete()* for the reasons for this
4205 division of responsibility.

4206 The *tss_delete()* function is not affected by signal handlers for the reasons stated in [xref to
4207 XRAT B.2.3].

4208 **FUTURE DIRECTIONS**

4209 None.

4210 **SEE ALSO**

4211 *pthread_key_create*, *tss_create*

4212 XBD <threads.h>

4213 CHANGE HISTORY

4214 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4215 NAME

4216 `tss_get`, `tss_set` — thread-specific data management

4217 SYNOPSIS

4218 `#include <threads.h>`

4219 `void *tss_get(tss_t key);`
4220 `int tss_set(tss_t key, void *val);`

4221 DESCRIPTION

4222 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4223 Any conflict between the requirements described here and the ISO C standard is
4224 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4225 The `tss_get()` function shall return the value for the current thread held in the thread-specific
4226 storage identified by `key`.

4227 The `tss_set()` function shall set the value for the current thread held in the thread-specific
4228 storage identified by `key` to `val`. This action shall not invoke the destructor associated with
4229 the key on the value being replaced.

4230 The application shall ensure that the `tss_get()` and `tss_set()` functions are only called with a
4231 value for `key` that was returned by a call to `tss_create()` before the thread commenced
4232 executing destructors.

4233 The effect of calling `tss_get()` or `tss_set()` after `key` has been deleted with `tss_delete()` is
4234 undefined.

4235 [CX]Both `tss_get()` and `tss_set()` can be called from a thread-specific data destructor
4236 function. A call to `tss_get()` for the thread-specific data key being destroyed shall return a
4237 null pointer, unless the value is changed (after the destructor starts) by a call to `tss_set()`.
4238 Calling `tss_set()` from a thread-specific data destructor function may result either in lost
4239 storage (after at least `PTHREAD_DESTRUCTOR_ITERATIONS` attempts at destruction)
4240 or in an infinite loop.

4241 These functions shall not be affected if the calling thread executes a signal handler during
4242 the call.[/CX]

4243 RETURN VALUE

4244 The `tss_get()` function shall return the value for the current thread. If no thread-specific data
4245 value is associated with `key`, then a null pointer shall be returned.

4246 The `tss_set()` function shall return `thrd_success` on success or `thrd_error` if the request
4247 could not be honored.

4248 ERRORS

4249 No errors are defined.

4250 **EXAMPLES**

4251 None.

4252 **APPLICATION USAGE**

4253 None.

4254 **RATIONALE**

4255 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
4256 B.2.3].

4257 **FUTURE DIRECTIONS**

4258 None.

4259 **SEE ALSO**

4260 *pthread_getspecific*, *tss_create*

4261 XBD <**threads.h**>

4262 **CHANGE HISTORY**

4263 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4264 Ref 7.31.11 para 2

4265 On page 2193 line 70145 section *ungetc()*, change FUTURE DIRECTIONS from:

4266 None.

4267 to:

4268 The ISO C standard states that the use of *ungetc()* on a binary stream where the file position
4269 indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction
4270 between binary and text streams, so this applies to all streams. This feature may be removed
4271 in a future version of this standard.

4272 Ref 7.29.6.3 para 1, 7.1.4 para 5

4273 On page 2242 line 71441 section *wcrtomb()*, change:

4274 [CX]The *wcrtomb()* function need not be thread-safe if called with a NULL *ps* argument.

4275 [/CX]

4276 to:

4277 If called with a null *ps* argument, the *wcrtomb()* function need not be thread-safe; however,
4278 such calls shall avoid data races with calls to *wcrtomb()* with a non-null argument and with
4279 calls to all other functions.

4280 Ref 7.29.6.4 para 1, 7.1.4 para 5

4281 On page 2266 line 72111 section *wcsrtombs()*, change:

4282 [CX]The *wcsnrtombs()* and *wcsrtombs()* functions need not be thread-safe if called with a
4283 NULL *ps* argument.[/CX]

4284 to:

4285 [CX]If called with a null *ps* argument, the *wcsnrtombs()* function need not be thread-safe;
4286 however, such calls shall avoid data races with calls to *wcsnrtombs()* with a non-null
4287 argument and with calls to all other functions.[/CX]

4288 If called with a null *ps* argument, the *wcsrtombs()* function need not be thread-safe; however,
4289 such calls shall avoid data races with calls to *wcsrtombs()* with a non-null argument and with
4290 calls to all other functions.

4291 Ref 7.22.7 para 1, 7.1.4 para 5

4292 On page 2292 line 72879 section *wctomb()*, change:

4293 [CX]The *wctomb()* function need not be thread-safe.[/CX]

4294 to:

4295 The *wctomb()* function need not be thread-safe; however, it shall avoid data races with all
4296 other functions.

4297 **Changes to XCU**

4298 Ref 7.22.2

4299 On page 2333 line 74167 section 1.1.2.2 Mathematical Functions, change:

4300 Section 7.20.2, Pseudo-Random Sequence Generation Functions

4301 to:

4302 Section 7.22.2, Pseudo-Random Sequence Generation Functions

4303 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4304 On page 2542 line 82220 section c99, rename the c99 page to c17.

4305 Ref 7.26

4306 On page 2545 line 82375 section c99 (now c17), change:

4307 ... , **<spawn.h>**, **<sys/socket.h>**, ...

4308 to:

4309 ... , **<spawn.h>**, **<sys/socket.h>**, **<threads.h>**, ...

4310 Ref 7.26

4311 On page 2545 line 82382 section c99 (now c17), change:

4312 This option shall make available all interfaces referenced in **<pthread.h>** and *pthread_kill()*
4313 and *pthread_sigmask()* referenced in **<signal.h>**.

4314 to:

4315 This option shall make available all interfaces referenced in `<pthread.h>` and `<threads.h>`,
4316 and also `pthread_kill()` and `pthread_sigmask()` referenced in `<signal.h>`.

4317 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4318 On page 2552-2553 line 82641-82677 section c99 (now c17), change CHANGE HISTORY to:

4319 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4320 Changes to XRAT

4321 Ref G.1 para 1

4322 On page 3483 line 117680 section A.1.7.1 Codes, add a new tagged paragraph:

4323 MXC This margin code is used to denote functionality related to the IEC 60559 Complex
4324 Floating-Point option. This functionality is mandated by the ISO C standard for IEC
4325 60559 implementations that support `<complex.h>`.

4326 Ref (none)

4327 On page 3489 line 117909 section A.3 Definitions (Byte), change:

4328 alignment with the ISO/IEC 9899: 1999 standard, where the `intN_t` types are now defined.

4329 to:

4330 alignment with the ISO/IEC 9899: 1999 standard, where the `intN_t` types were first defined.

4331 Ref 5.1.2.4, 7.17.3

4332 On page 3515 line 118946 section A.4.12 Memory Synchronization, change:

4333 **A.4.12 Memory Synchronization**

4334 to:

4335 **A.4.12 Memory Ordering and Synchronization**

4336 *A.4.12.1 Memory Ordering*

4337 There is no additional rationale provided for this section.

4338 *A.4.12.2 Memory Synchronization*

4339 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4340 On page 3556 line 120684 section A.12.2 Utility Syntax Guidelines, change:

4341 Thus, they had to devise a new name, `c89` (now superseded by `c99`), rather than ...

4342 to:

4343 Thus, they had to devise a new name, `c89` (subsequently superseded by `c99` and now by

4344 `c17`), rather than ...

4345 Ref K.3.1.1

4346 On page 3567 line 121053 section B.2.2.1 POSIX.1 Symbols, add a new unnumbered subsection:

4347 **The `__STDC_WANT_LIB_EXT1__` Feature Test Macro**

4348 The ISO C standard specifies the feature test macro `__STDC_WANT_LIB_EXT1__` as the
4349 announcement mechanism for the application that it requires functionality from Annex K. It
4350 specifies that the symbols specified in Annex K (if supported) are made visible when
4351 `__STDC_WANT_LIB_EXT1__` is 1 and are not made visible when it is 0, but leaves it
4352 unspecified whether they are made visible when `__STDC_WANT_LIB_EXT1__` is
4353 undefined. POSIX.1 requires that they are not made visible when the macro is undefined
4354 (except for those symbols that are already explicitly allowed to be visible through the
4355 definition of `_POSIX_C_SOURCE` or `_XOPEN_SOURCE`, or both).

4356 POSIX.1 does not include the interfaces specified in Annex K of the ISO C standard, but
4357 allows the symbols to be made visible in headers when requested by the application in order
4358 that applications can use symbols from Annex K and symbols from POSIX.1 in the same
4359 translation unit.

4360 Ref 6.10.3.4

4361 On page 3570 line 121176 section B.2.2.2 The Name Space, change:

4362 as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C
4363 standard

4364 to:

4365 as described for macros that expand to their own name as in Section 6.10.3.4 of the ISO C
4366 standard

4367 Ref 7.5 para 2

4368 On page 3571 line 121228-121243 section B.2.3 Error Numbers, change:

4369 The ISO C standard requires that *errno* be an assignable lvalue. Originally, ...

4370 [...]

4371 ... using the return value for a mixed purpose was judged to be of limited use and

4372 error prone.

4373 to:

4374 The original ISO C standard just required that *errno* be a modifiable lvalue. Since the
4375 introduction of threads in 2011, the ISO C standard has instead required that *errno* be a
4376 macro which expands to a modifiable lvalue that has thread local storage duration.

4377 Ref 7.26

4378 On page 3575 line 121390 section B.2.3 Error Numbers, change:

4379 In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a
4380 special case since it will never occur.

4381 to:

4382 In particular, applications calling blocking interfaces need not handle any possible [EINTR]
4383 return as a special case since it will never occur. In the case of threads functions in
4384 <threads.h>, the requirement is stated in terms of the call not being affected if the calling
4385 thread executes a signal handler during the call, since these functions return errors in a
4386 different way and cannot distinguish an [EINTR] condition from other error conditions.

4387 Ref (none)

4388 On page 3733 line 128128 section C.2.6.4 Arithmetic Expansion, change:

4389 Although the ISO/IEC 9899: 1999 standard now requires support for ...

4390 to:

4391 Although the ISO C standard requires support for ...

4392 Ref 7.17

4393 On page 3789 line 129986 section E.1 Subprofiling Option Groups, change:

4394 by collecting sets of related functions

4395 to:

4396 by collecting sets of related functions and generic functions

4397 Ref 7.22.3.1, 7.27.2.5, 7.22.4

4398 On page 3789, 3792 line 130022-130032, 130112-130114 section E.1 Subprofiling Option Groups,
4399 add new functions (in sorted order) to the existing groups as indicated:

4400 POSIX_C_LANG_SUPPORT

4401 *aligned_alloc(), timespec_get()*

4402 POSIX_MULTI_PROCESS

4403 *at_quick_exit(), quick_exit()*

4404 Ref 7.17

4405 On page 3789 line 129991 section E.1 Subprofiling Option Groups, add:

4406 POSIX_C_LANG_ATOMICS: ISO C Atomic Operations

4407 *atomic_compare_exchange_strong(), atomic_compare_exchange_strong_explicit(),*

4408 *atomic_compare_exchange_weak(), atomic_compare_exchange_weak_explicit(),*

4409 *atomic_exchange(), atomic_exchange_explicit(), atomic_fetch_add(),*

4410 *atomic_fetch_add_explicit(), atomic_fetch_and(), atomic_fetch_and_explicit(),*

4411 *atomic_fetch_or(), atomic_fetch_or_explicit(), atomic_fetch_sub(),*

4412 *atomic_fetch_sub_explicit(), atomic_fetch_xor(), atomic_fetch_xor_explicit(),*

4413 *atomic_flag_clear(), atomic_flag_clear_explicit(), atomic_flag_test_and_set(),*

4414 *atomic_flag_test_and_set_explicit(), atomic_init(), atomic_is_lock_free(),*

4415 *atomic_load(), atomic_load_explicit(), atomic_signal_fence(),*

4416 *atomic_thread_fence(), atomic_store(), atomic_store_explicit(), kill_dependency()*

4417 Ref 7.26

4418 On page 3790 line 1300349 section E.1 Subprofiling Option Groups, add:

4419 POSIX_C_LANG_THREADS: ISO C Threads
4420 *call_once(), cnd_broadcast(), cnd_signal(), cnd_destroy(), cnd_init(),*
4421 *cnd_timedwait(), cnd_wait(), mtx_destroy(), mtx_init(), mtx_lock(), mtx_timedlock(),*
4422 *mtx_trylock(), mtx_unlock(), thrd_create(), thrd_current(), thrd_detach(),*
4423 *thrd_equal(), thrd_exit(), thrd_join(), thrd_sleep(), thrd_yield(), tss_create(),*
4424 *tss_delete(), tss_get(), tss_set()*

4425 POSIX_C_LANG_UCHAR: ISO C Unicode Utilities
4426 *c16rtomb(), c32rtomb(), mbrtoc16(), mbrtoc32()*