

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~~2020-10-29

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 has already ~~shaded-OB-and-so-will-~~
11 automatically been removed by default in Issue 8 bug 1330.

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.

250
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 _IImaginary_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
atomic_int_fast64_t	_Atomic int_fast64_t
atomic_uint_fast64_t	_Atomic uint_fast64_t
atomic_intptr_t	_Atomic intptr_t

<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 [following](#) macros-:

795 `thread_local` ~~which shall~~ [_____](#) `e`Expands to `_Thread_local`.

796 ONCE_FLAG_INIT ~~which shall~~ eExpands to a value that can be used to initialize an
797 object of type **once_flag**; ~~and~~.

798 TSS_DTOR_ITERATIONS ~~which shall~~ eExpands to an integer constant expression
799 representing the maximum number of times that destructors
800 will be called when a thread terminates and shall be suitable
801 for use in **#if** preprocessing directives.-

802 [CX]If {PTHREAD_DESTRUCTOR_ITERATIONS} is defined in **<limits.h>**, the value of
803 TSS_DTOR_ITERATIONS shall be equal to
804 {PTHREAD_DESTRUCTOR_ITERATIONS}; otherwise, the value of
805 TSS_DTOR_ITERATIONS shall be greater than or equal to the value of
806 {_POSIX_THREAD_DESTRUCTOR_ITERATIONS} and shall be less than or equal to the
807 maximum positive value that can be returned by a call to
808 *sysconf*(_SC_THREAD_DESTRUCTOR_ITERATIONS) in any process.[/CX]

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

813 The **<threads.h>** header shall define the enumeration constants **mtx_plain**,
814 **mtx_recursive**, **mtx_timed**, **thrd_busy**, **thrd_error**, **thrd_nomem**, **thrd_success**
815 and **thrd_timedout**.

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

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

845 int tss_set(tss_t, void *);

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

848 **APPLICATION USAGE**

849 The <**threads.h**> header is optional in the ISO C standard but is mandated by POSIX.1-
850 20xx. Note however that subprofiles can choose to make this header optional (see [xref to
851 2.1.5.1 Subprofiling Considerations]), and therefore application portability to subprofile
852 implementations would benefit from checking whether `__STDC_NO_THREADS__` is
853 defined before inclusion of <**threads.h**>.

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

860 **RATIONALE**

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

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

872 The standard developers considered making it implementation-defined whether the types
873 **cond_t**, **mtx_t** and **tss_t** are interchangeable with the corresponding types **pthread_cond_t**,
874 **pthread_mutex_t** and **pthread_key_t** defined in <**pthread.h**> (that is, whether any
875 function that can be called with a valid **cond_t** can also be called with a valid
876 **pthread_cond_t**, and vice versa, and likewise for the other types). However, this would
877 have meant extending `mtx_lock()` to provide a way for it to indicate that the owner of a
878 mutex has terminated (equivalent to [EOWNERDEAD]). It was felt that such an extension
879 would be invention. Although there was no similar concern for **cond_t** and **tss_t**, they were
880 treated the same way as **mtx_t** for consistency. [See also the RATIONALE for `mtx_lock\(\)`
881 concerning the inability of `mtx_t` to contain information about whether or not a mutex
882 supports timeout if it is the same type as `pthread_mutex_t`.](#)

883 **FUTURE DIRECTIONS**

884 None.

885 **SEE ALSO**

886 <**limits.h**>, <**pthread.h**>, <**time.h**>

887 XSH Section 2.9, `call_once()`, `cond_broadcast()`, `cond_destroy()`, `cond_timedwait()`,

888 *mtx_destroy()*, *mtx_lock()*, *sysconf()*, *thrd_create()*, *thrd_current()*, *thrd_detach()*,
889 *thrd_equal()*, *thrd_exit()*, *thrd_join()*, *thrd_sleep()*, *thrd_yield()*, *tss_create()*, *tss_delete()*,
890 *tss_get()*.

891 CHANGE HISTORY

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

893 Ref 7.27.1 para 4

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

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

897	<code>time_t</code>	<code>tv_sec</code>	Seconds.
898	<code>long</code>	<code>tv_nsec</code>	Nanoseconds.

899 and change the members to:

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

902 Ref 7.27.1 para 2

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

904	<code>TIME_UTC</code>	An integer constant greater than 0 that designates the UTC time base 905 in calls to <i>timespec_get()</i> . The value shall be suitable for use in #if 906 preprocessing directives.
-----	-----------------------	--

907 Ref 7.27.2.5

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

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

910 Ref 7.28

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

912 NAME

913 `uchar.h` — Unicode character handling

914 SYNOPSIS

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

916 DESCRIPTION

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

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

921 `mbstate_t` As described in <**wchar.h**>.

922 **size_t** As described in `<stddef.h>`.

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

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

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

927 **size_t** **c16rtomb**(char *restrict, char16_t,
928 mbstate_t *restrict);

929 **size_t** **c32rtomb**(char *restrict, char32_t,
930 mbstate_t *restrict);

931 **size_t** **mbrtoc16**(char16_t *restrict, const char *restrict,
932 size_t, mbstate_t *restrict);

933 **size_t** **mbrtoc32**(char32_t *restrict, const char *restrict,
934 size_t, mbstate_t *restrict);

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

937 **APPLICATION USAGE**

938 None.

939 **RATIONALE**

940 None.

941 **FUTURE DIRECTIONS**

942 None.

943 **SEE ALSO**

944 `<stddef.h>`, `<stdint.h>`, `<wchar.h>`

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

946 **CHANGE HISTORY**

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

948 Ref 7.22.4.5 para 1

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

950 void _exit(int);

951 to:

952 _Noreturn void _exit(int);

953 Ref 7.29.1 para 2

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

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

956 to:

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

958 **Changes to XSH**

959 Ref 7.1.4 paras 5, 6

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

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

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

971 Ref K.3.1.1

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

973 2.2.1.3 *The `__STDC_WANT_LIB_EXT1__` Feature Test Macro*

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

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

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

986 Ref 7.31.8 para 1

987 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>		
----------------------------------	---	--	--

988 Ref 7.31.15 para 1

989 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>		
--------------------------------	--	--	--

990 Ref 7.31.8 para 1
 991 On page 477 line 16410 section 2.2.2, insert a row in the table:

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

992 Ref 7.31.14 para 1
 993 On page 477 line 16417 section 2.2.2, insert a row in the table:

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

994 Ref K.3.4 - K.3.9
 995 On page 477 line 16436 section 2.2.2 The Name Space, add:

996 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
 997 (see [xref to 2.2.1]), implementations may add symbols to the headers shown in the
 998 following table provided the identifiers for those symbols have one of the corresponding
 999 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, wctombs_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

1000 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
 1001 (see [xref to 2.2.1]), if any header in the following table is included, macros with the
 1002 complete names shown may be defined.

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

1003 **Note:** The above two tables only include those symbols from Annex K of the ISO C standard that
 1004 are not already allowed to be visible by entries in earlier tables in this section.

1005 Ref 7.1.3 para 1
 1006 On page 478 line 16438 section 2.2.2, change:

1007 With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that
 1008 begin with an <underscore> and either an uppercase letter or another <underscore> are
 1009 always reserved for any use by the implementation.

1010 to:

1011 With the exception of identifiers beginning with the prefix `_POSIX_` and those identifiers
1012 which are lexically identical to keywords defined by the ISO C standard (for example
1013 `_Bool`), all identifiers that begin with an `<underscore>` and either an uppercase letter or
1014 another `<underscore>` are always reserved for any use by the implementation.

1015 Ref 7.1.3 para 1

1016 On page 478 line 16448 section 2.2.2, change:

1017 that have external linkage are always reserved

1018 to:

1019 that have external linkage and `errno` are always reserved

1020 Ref 7.1.3 para 1

1021 On page 479 line 16453 section 2.2.2, add the following in the appropriate place in the list:

1022	<code>aligned_alloc</code>	<code>c32rtomb</code>
1023	<code>at_quick_exit</code>	<code>call_once</code>
1024	<code>atomic_compare_exchange_strong</code>	<code>cnd_broadcast</code>
1025	<code>atomic_compare_exchange_strong_explicit</code>	<code>cnd_destroy</code>
1026	<code>atomic_compare_exchange_weak</code>	<code>cnd_init</code>
1027	<code>atomic_compare_exchange_weak_explicit</code>	<code>cnd_signal</code>
1028	<code>atomic_exchange</code>	<code>cnd_timedwait</code>
1029	<code>atomic_exchange_explicit</code>	<code>cnd_wait</code>
1030	<code>atomic_fetch_add</code>	<code>kill_dependency</code>
1031	<code>atomic_fetch_add_explicit</code>	<code>mbrtoc16</code>
1032	<code>atomic_fetch_and</code>	<code>mbrtoc32</code>
1033	<code>atomic_fetch_and_explicit</code>	<code>mtx_destroy</code>
1034	<code>atomic_fetch_or</code>	<code>mtx_init</code>
1035	<code>atomic_fetch_or_explicit</code>	<code>mtx_lock</code>
1036	<code>atomic_fetch_sub</code>	<code>mtx_timedlock</code>
1037	<code>atomic_fetch_sub_explicit</code>	<code>mtx_trylock</code>
1038	<code>atomic_fetch_xor</code>	<code>mtx_unlock</code>
1039	<code>atomic_fetch_xor_explicit</code>	<code>quick_exit</code>
1040	<code>atomic_flag_clear</code>	<code>thrd_create</code>
1041	<code>atomic_flag_clear_explicit</code>	<code>thrd_current</code>
1042	<code>atomic_flag_test_and_set</code>	<code>thrd_detach</code>
1043	<code>atomic_flag_test_and_set_explicit</code>	<code>thrd_equal</code>
1044	<code>atomic_init</code>	<code>thrd_exit</code>
1045	<code>atomic_is_lock_free</code>	<code>thrd_join</code>
1046	<code>atomic_load</code>	<code>thrd_sleep</code>
1047	<code>atomic_load_explicit</code>	<code>thrd_yield</code>
1048	<code>atomic_signal_fence</code>	<code>timespec_get</code>
1049	<code>atomic_store</code>	<code>tss_create</code>
1050	<code>atomic_store_explicit</code>	<code>tss_delete</code>
1051	<code>atomic_thread_fence</code>	<code>tss_get</code>
1052	<code>c16rtomb</code>	<code>tss_set</code>

1053 Ref 7.1.2 para 4

1054 On page 480 line 16551 section 2.2.2, change:

1055 Prior to the inclusion of a header, the application shall not define any macros with names
1056 lexically identical to symbols defined by that header.

1057 to:

1058 Prior to the inclusion of a header, or when any macro defined in the header is expanded, the
1059 application shall not define any macros with names lexically identical to symbols defined by
1060 that header.

1061 Ref 7.26.5.1

1062 On page 490 line 16980 section 2.4.2 Realtime Signal Generation and Delivery, change:

1063 The function shall be executed in an environment as if it were the *start_routine* for a newly
1064 created thread with thread attributes specified by *sigev_notify_attributes*.

1065 to:

1066 The function shall be executed in a newly created thread as if it were the *start_routine* for a
1067 call to *pthread_create()* with the thread attributes specified by *sigev_notify_attributes*.

1068 Ref 7.14.1.1 para 5

1069 On page 493 line 17088 section 2.4.3 Signal Actions, change:

1070 with static storage duration

1071 to:

1072 with static or thread storage duration that is not a lock-free atomic object

1073 Ref 7.14.1.1 para 5

1074 On page 493 line 17090 section 2.4.3 Signal Actions, after applying bug 711 change:

1075 other than one of the functions and macros listed in the following table

1076 to:

1077 other than one of the functions and macros specified below as being async-signal-safe

1078 Ref 7.14.1.1 para 5

1079 On page 494 line 17133 section 2.4.3 Signal Actions, add *quick_exit()* to the table of async-signal-
1080 safe functions.

1081 Ref 7.14.1.1 para 5

1082 On page 494 line 17147 section 2.4.3 Signal Actions, change:

1083 Any function or function-like macro not in the above table may be unsafe with respect to
1084 signals.

1085 to:

1086 In addition, the functions in `<stdatomic.h>` other than `atomic_init()` shall be async-signal-
1087 safe when the atomic arguments are lock-free, and the `atomic_is_lock_free()` function shall
1088 be async-signal-safe when called with an atomic argument.

1089 All other functions (including generic functions) and function-like macros may be unsafe
1090 with respect to signals.

1091 Ref 7.21.2 para 7,8

1092 On page 496 line 17228 section 2.5 Standard I/O Streams, add a new paragraph:

1093 Each stream shall have an associated lock that is used to prevent data races when multiple
1094 threads of execution access a stream, and to restrict the interleaving of stream operations
1095 performed by multiple threads. Only one thread can hold this lock at a time. The lock shall
1096 be reentrant: a single thread can hold the lock multiple times at a given time. All functions
1097 that read, write, position, or query the position of a stream, [CX]except those with names
1098 ending `_unlocked`[/CX], shall lock the stream [CX] as if by a call to `flockfile()`[/CX] before
1099 accessing it and release the lock [CX] as if by a call to `funlockfile()`[/CX] when the access is
1100 complete.

1101 Ref (none)

1102 On page 498 line 17312 section 2.5.2 Stream Orientation and Encoding Rules, change:

1103 For conformance to the ISO/IEC 9899: 1999 standard, the definition of a stream includes an
1104 “orientation”.

1105 to:

1106 The definition of a stream includes an “orientation”.

1107 Ref 7.26.5.8

1108 On page 508 line 17720 section 2.8.4 Process Scheduling, change:

1109 When a running thread issues the `sched_yield()` function

1110 to:

1111 When a running thread issues the `sched_yield()` or `thr_yield()` function

1112 Ref 7.17.2.2 para 3, 7.22.2.2 para 3

1113 On page 513 line 17907,17916 section 2.9.1 Thread-Safety, add `atomic_init()` and `srand()` to the list
1114 of functions that need not be thread-safe.

1115 Ref 7.12.8.3, 7.22.4.8

1116 On page 513 line 17907-17927 section 2.9.1 Thread-Safety, delete the following from the list of
1117 functions that need not be thread-safe:

1118 `lgamma()`, `lgammaf()`, `lgammal()`, `system()`

1119 [Note to reviewers: deletion of `mblen\(\)`, `mbtowc\(\)`, and `wctomb\(\)` from this list is the subject of](#)
1120 [Mantis bug 708.](#)

1121 Ref 7.28.1 para 1

1122 On page 513 line 17928 section 2.9.1 Thread-Safety, change:

1123 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument.
1124 The *mbrlen()*, *mbrtowc()*, *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and
1125 *wcsrtombs()* functions need not be thread-safe if passed a NULL *ps* argument.

1126 to:

1127 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a null pointer
1128 argument. The *c16rtomb()*, *c32rtomb()*, *mbrlen()*, *mbrtoc16()*, *mbrtoc32()*, *mbrtowc()*,
1129 *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and *wcsrtombs()* functions need not
1130 be thread-safe if passed a null *ps* argument. The *lgamma()*, *lgammaf()*, and *lgammal()*
1131 functions shall be thread-safe [XSI]except that they need not avoid data races when storing a
1132 value in the *siggam* variable[/XSI].

1133 Ref 7.1.4 para 5

1134 On page 513 line 17934 section 2.9.1 Thread-Safety, change:

1135 Implementations shall provide internal synchronization as necessary in order to satisfy this
1136 requirement.

1137 to:

1138 Some functions that are not required to be thread-safe are nevertheless required to avoid data
1139 races with either all or some other functions, as specified on their individual reference pages.

1140 Implementations shall provide internal synchronization as necessary in order to satisfy
1141 thread-safety requirements.

1142 Ref 7.26.5

1143 On page 513 line 17944 section 2.9.2 Thread IDs, change:

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

1147 to:

1148 The lifetime of a thread ID ends after the thread terminates if it was created using
1149 *pthread_create()* with the *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or
1150 if *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()* has been called for that
1151 thread.

1152 Ref 7.26.5

1153 On page 514 line 17950 section 2.9.2 Thread IDs, change:

1154 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1155 *pthread_detach()* or *pthread_join()*.

1156 to:

1157 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1158 *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()*.

1159 Ref 7.26.4

1160 On page 514 line 17956 section 2.9.3 Thread Mutexes, change:

1161 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

1162 to:

1163 A thread shall become the owner of a mutex, *m*, of type **pthread_mutex_t** when one of the
1164 following occurs:

1165 Ref 7.26.3, 7.26.4

1166 On page 514 line 17972 section 2.9.3 Thread Mutexes, add two new paragraphs and lists:

1167 A thread shall become the owner of a mutex, *m*, of type **mtx_t** when one of the following
1168 occurs:

- 1169 • It calls *mtx_lock()* with *m* as the *mtx* argument and the call returns *thrd_success*.
- 1170 • It calls *mtx_trylock()* with *m* as the *mtx* argument and the call returns
1171 *thrd_success*.
- 1172 • It calls *mtx_timedlock()* with *m* as the *mtx* argument and the call returns
1173 *thrd_success*.
- 1174 • It calls *cond_wait()* with *m* as the *mtx* argument and the call returns *thrd_success*.
- 1175 • It calls *cond_timedwait()* with *m* as the *mtx* argument and the call returns
1176 *thrd_success* or *thrd_timedout*.

1177 The thread shall remain the owner of *m* until one of the following occurs:

- 1178 • It executes *mtx_unlock()* with *m* as the *mtx* argument.
- 1179 • It blocks in a call to *cond_wait()* with *m* as the *mtx* argument.
- 1180 • It blocks in a call to *cond_timedwait()* with *m* as the *mtx* argument.

1181 Ref 7.26.4

1182 On page 514 line 17980 section 2.9.3 Thread Mutexes, change:

1183 Robust mutexes provide a means to enable the implementation to notify other threads in the
1184 event of a process terminating while one of its threads holds a mutex lock.

1185 to:

1186 Robust mutexes provide a means to enable the implementation to notify other threads in the
1187 event of a process terminating while one of its threads holds a lock on a mutex of type
1188 **pthread_mutex_t**.

1189 Ref 7.26.5

1190 On page 517 line 18085 section 2.9.5 Thread Cancellation, change:

1191 The thread cancellation mechanism allows a thread to terminate the execution of any other
1192 thread in the process in a controlled manner.

1193 to:

1194 The thread cancellation mechanism allows a thread to terminate the execution of any thread
1195 in the process, except for threads created using *thrd_create()*, in a controlled manner.

1196 Ref 7.26.3, 7.26.5.6
1197 On page 518 line 18119-18137 section 2.9.5.2 Cancellation Points, add the following to the list of
1198 functions that are required to be cancellation points:

1199 *cnd_timedwait()*, *cnd_wait()*, *thrd_join()*, *thrd_sleep()*

1200 Ref 7.26.5
1201 On page 520 line 18225 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1202 Each thread maintains a list of cancellation cleanup handlers.

1203 to:

1204 Each thread that was not created using *thrd_create()* maintains a list of cancellation cleanup
1205 handlers.

1206 Ref 7.26.6.1
1207 On page 521 line 18240 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1208 as described for *pthread_key_create()*

1209 to:

1210 as described for *pthread_key_create()* and *tss_create()*

1211 Ref 7.26
1212 On page 523 line 18337 section 2.9.9 Synchronization Object Copies and Alternative Mappings,
1213 add a new sentence:

1214 For ISO C functions declared in **<threads.h>**, the above requirements shall apply as if
1215 condition variables of type **cnd_t** and mutexes of type **mtx_t** have a process-shared attribute
1216 that is set to **PTHREAD_PROCESS_PRIVATE**.

1217 Ref 7.26.3
1218 On page 547 line 19279 section 2.12.1 Defined Types, change:

1219 **pthread_cond_t**

1220 to

1221 **pthread_cond_t, cnd_t**

1222 Ref 7.26.6, 7.26.4
1223 On page 547 line 19281 section 2.12.1 Defined Types, change:

1224 **pthread_key_t**
1225 **pthread_mutex_t**

1226 to

1227 **pthread_key_t, tss_t**
1228 **pthread_mutex_t, mtx_t**

1229 Ref 7.26.2.1
1230 On page 547 line 19284 section 2.12.1 Defined Types, change:

1231 **pthread_once_t**

1232 to

1233 **pthread_once_t, once_flag**

1234 Ref 7.26.5
1235 On page 547 line 19287 section 2.12.1 Defined Types, change:

1236 **pthread_t**

1237 to

1238 **pthread_t, thrd_t**

1239 Ref 7.3.9.3
1240 On page 552 line 19370 insert a new CMPLX() section:

1241 **NAME**
1242 CMPLX — make a complex value

1243 **SYNOPSIS**
1244 `#include <complex.h>`

1245 `double complex CMPLX(double x, double y);`
1246 `float complex CMPLXF(float x, float y);`
1247 `long double complex CMPLXL(long double x, long double y);`

1248 **DESCRIPTION**
1249 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1250 Any conflict between the requirements described here and the ISO C standard is
1251 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1252 The CMPLX macros shall expand to an expression of the specified complex type, with the
1253 real part having the (converted) value of x and the imaginary part having the (converted)
1254 value of y . The resulting expression shall be suitable for use as an initializer for an object
1255 with static or thread storage duration, provided both arguments are likewise suitable.

1256 **RETURN VALUE**
1257 The CMPLX macros return the complex value $x + iy$ (where i is the imaginary unit).

1258 These macros shall behave as if the implementation supported imaginary types and the
1259 definitions were:

```
1260     #define CMLX(x, y) ((double complex)((double)(x) + \  
1261         _Imaginary_I * (double)(y)))  
1262     #define CMLXF(x, y) ((float complex)((float)(x) + \  
1263         _Imaginary_I * (float)(y)))  
1264     #define CMLXL(x, y) ((long double complex)((long double)(x) + \  
1265         _Imaginary_I * (long double)(y)))
```

1266 **ERRORS**

1267 No errors are defined.

1268 **EXAMPLES**

1269 None.

1270 **APPLICATION USAGE**

1271 None.

1272 **RATIONALE**

1273 None.

1274 **FUTURE DIRECTIONS**

1275 None.

1276 **SEE ALSO**

1277 XBD <complex.h>

1278 **CHANGE HISTORY**

1279 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1280 Ref 7.22.4.5 para 1

1281 On page 553 line 19384 section `_Exit()`, change:

```
1282     void _Exit(int status);
```

```
1283     #include <unistd.h>
```

```
1284     void _exit(int status);
```

1285 to:

```
1286     _Noreturn void _Exit(int status);
```

```
1287     #include <unistd.h>
```

```
1288     _Noreturn void _exit(int status);
```

1289 Ref 7.22.4.5 para 2

1290 On page 553 line 19396 section `_Exit()`, change:

1291 shall not call functions registered with `atexit()` nor any registered signal handlers

1292 to:

1293 shall not call functions registered with `atexit()` nor `at_quick_exit()`, nor any registered signal

1294 handlers

1295 Ref (none)

1296 On page 557 line 19562 section `_Exit()`, change:

1297 The ISO/IEC 9899: 1999 standard adds the `_Exit()` function

1298 to:

1299 The ISO/IEC 9899: 1999 standard added the `_Exit()` function

1300 Ref 7.22.4.3, 7.22.4.7

1301 On page 557 line 19568 section `_Exit()`, add `at_quick_exit` and `quick_exit` to the SEE ALSO section.

1302 Ref 7.22.4.1 para 1

1303 On page 565 line 19761 section `abort()`, change:

1304 `void abort(void);`

1305 to:

1306 `_Noreturn void abort(void);`

1307 Ref (none)

1308 On page 565 line 19785 section `abort()`, change:

1309 The ISO/IEC 9899: 1999 standard requires the `abort()` function to be async-signal-safe.

1310 to:

1311 The ISO/IEC 9899: 1999 standard required (and the current standard still requires) the

1312 `abort()` function to be async-signal-safe.

1313 Ref 7.22.3.1

1314 On page 597 line 20771 insert the following new `aligned_alloc()` section:

1315 **NAME**

1316 `aligned_alloc` — allocate memory with a specified alignment

1317 **SYNOPSIS**

1318 `#include <stdlib.h>`

1319 `void *aligned_alloc(size_t alignment, size_t size);`

1320 **DESCRIPTION**

1321 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1322 Any conflict between the requirements described here and the ISO C standard is

1323 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1324 The `aligned_alloc()` function shall allocate unused space for an object whose alignment is

1325 specified by `alignment`, whose size in bytes is specified by `size` and whose value is

1326 indeterminate.

1327 The order and contiguity of storage allocated by successive calls to *aligned_alloc()* is
1328 unspecified. Each such allocation shall yield a pointer to an object disjoint from any other
1329 object. The pointer returned shall point to the start (lowest byte address) of the allocated
1330 space. If the value of *alignment* is not a valid alignment supported by the implementation, a
1331 null pointer shall be returned. If the space cannot be allocated, a null pointer shall be
1332 returned. If the size of the space requested is 0, the behavior is implementation-defined:
1333 either a null pointer shall be returned to indicate an error, or the behavior shall be as if the
1334 size were some non-zero value, except that the behavior is undefined if the returned pointer
1335 is used to access an object.

1336 For purposes of determining the existence of a data race, *aligned_alloc()* shall behave as
1337 though it accessed only memory locations accessible through its arguments and not other
1338 static duration storage. The function may, however, visibly modify the storage that it
1339 allocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*,
1340 *[ADV]posix_memalign()*,*[/ADV]* and *realloc()* that allocate or deallocate a particular region
1341 of memory shall occur in a single total order (see *[xref to XBD 4.12.1]*), and each such
1342 deallocation call shall synchronize with the next allocation (if any) in this order.

1343 RETURN VALUE

1344 Upon successful completion with *size* not equal to 0, *aligned_alloc()* shall return a pointer to
1345 the allocated space. If *size* is 0, either:

- 1346 • A null pointer shall be returned *[CX]* and *errno* may be set to an implementation-
1347 defined value,*[/CX]* or
- 1348 • A pointer to the allocated space shall be returned. The application shall ensure that
1349 the pointer is not used to access an object.

1350 Otherwise, it shall return a null pointer *[CX]* and set *errno* to indicate the error $[/CX]$.

1351 ERRORS

1352 The *aligned_alloc()* function shall fail if:

1353 *[CX][EINVAL]* The value of *alignment* is not a valid alignment supported by the
1354 implementation.

1355 *[ENOMEM]* Insufficient storage space is available. $[/CX]$

1356 EXAMPLES

1357 None.

1358 APPLICATION USAGE

1359 None.

1360 RATIONALE

1361 None.

1362 FUTURE DIRECTIONS

1363 None.

1364 SEE ALSO

1365 *calloc, free, getrlimit, malloc, posix_memalign, realloc*

1366 XBD <stdlib.h>

1367 **CHANGE HISTORY**

1368 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1369 Ref 7.27.3, 7.1.4 para 5

1370 On page 600 line 20911 section `asctime()`, change:

1371 [CX]The `asctime()` function need not be thread-safe.[/CX]

1372 to:

1373 The `asctime()` function need not be thread-safe; however, `asctime()` shall avoid data races
1374 with all functions other than itself, `ctime()`, `gmtime()` and `localtime()`.

1375 Ref 7.22.4.3

1376 On page 618 line 21380 insert the following new `at_quick_exit()` section:

1377 **NAME**

1378 `at_quick_exit` — register a function to be called from `quick_exit()`

1379 **SYNOPSIS**

1380 `#include <stdlib.h>`

1381 `int at_quick_exit(void (*func)(void));`

1382 **DESCRIPTION**

1383 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1384 Any conflict between the requirements described here and the ISO C standard is
1385 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1386 The `at_quick_exit()` function shall register the function pointed to by `func`, to be called
1387 without arguments should `quick_exit()` be called. It is unspecified whether a call to the
1388 `at_quick_exit()` function that does not happen before the `quick_exit()` function is called will
1389 succeed.

1390 At least 32 functions can be registered with `at_quick_exit()`.

1391 [CX]After a successful call to any of the `exec` functions, any functions previously registered
1392 by `at_quick_exit()` shall no longer be registered.[/CX]

1393 **RETURN VALUE**

1394 Upon successful completion, `at_quick_exit()` shall return 0; otherwise, it shall return a non-
1395 zero value.

1396 **ERRORS**

1397 No errors are defined.

1398 **EXAMPLES**

1399 None.

1400 **APPLICATION USAGE**

1401 The *at_quick_exit()* function registrations are distinct from the *atexit()* registrations, so
1402 applications might need to call both registration functions with the same argument.

1403 The functions registered by a call to *at_quick_exit()* must return to ensure that all registered
1404 functions are called.

1405 The application should call *sysconf()* to obtain the value of {ATEXIT_MAX}, the number of
1406 functions that can be registered. There is no way for an application to tell how many
1407 functions have already been registered with *at_quick_exit()*.

1408 Since the behavior is undefined if the *quick_exit()* function is called more than once,
1409 portable applications calling *at_quick_exit()* must ensure that the *quick_exit()* function is not
1410 called when the functions registered by the *at_quick_exit()* function are called.

1411 If a function registered by the *at_quick_exit()* function is called and a portable application
1412 needs to stop further *quick_exit()* processing, it must call the *_exit()* function or the *_Exit()*
1413 function or one of the functions which cause abnormal process termination.

1414 **RATIONALE**

1415 None.

1416 **FUTURE DIRECTIONS**

1417 None.

1418 **SEE ALSO**

1419 *atexit*, *exec*, *exit*, *quick_exit*, *sysconf*

1420 XBD <**stdlib.h**>

1421 **CHANGE HISTORY**

1422 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1423 Ref 7.22.4.3

1424 On page 618 line 21381 section *atexit()*, change:

1425 *atexit* — register a function to run at process termination

1426 to:

1427 *atexit* — register a function to be called from *exit()* or after return from *main()*

1428 Ref 7.22.4.2 para 2, 7.22.4.3

1429 On page 618 line 21389 section *atexit()*, change:

1430 The *atexit()* function shall register the function pointed to by *func*, to be called without
1431 arguments at normal program termination. At normal program termination, all functions
1432 registered by the *atexit()* function shall be called, in the reverse order of their registration,
1433 except that a function is called after any previously registered functions that had already
1434 been called at the time it was registered. Normal termination occurs either by a call to *exit()*
1435 or a return from *main()*.

1436 to:

1437 The *atexit()* function shall register the function pointed to by *func*, to be called without
1438 arguments from *exit()*, or after return from the initial call to *main()*, or on the last thread
1439 termination. If the *exit()* function is called, it is unspecified whether a call to the *atexit()*
1440 function that does not happen before *exit()* is called will succeed.

1441 [Note to reviewers: the part about all registered functions being called in reverse order is duplicated](#)
1442 [on the *exit\(\)* page and is not needed here.](#)

1443 Ref 7.22.4.2 para 2

1444 On page 618 line 21405 section *atexit()*, insert a new first APPLICATION USAGE paragraph:

1445 The *atexit()* function registrations are distinct from the *at_quick_exit()* registrations, so
1446 applications might need to call both registration functions with the same argument.

1447 Ref 7.22.4.3

1448 On page 618 line 21410 section *atexit()*, change:

1449 Since the behavior is undefined if the *exit()* function is called more than once, portable
1450 applications calling *atexit()* must ensure that the *exit()* function is not called at normal
1451 process termination when all functions registered by the *atexit()* function are called.

1452 All functions registered by the *atexit()* function are called at normal process termination,
1453 which occurs by a call to the *exit()* function or a return from *main()* or on the last thread
1454 termination, when the behavior is as if the implementation called *exit()* with a zero argument
1455 at thread termination time.

1456 If, at normal process termination, a function registered by the *atexit()* function is called and a
1457 portable application needs to stop further *exit()* processing, it must call the *_exit()* function
1458 or the *_Exit()* function or one of the functions which cause abnormal process termination.

1459 to:

1460 Since the behavior is undefined if the *exit()* function is called more than once, portable
1461 applications calling *atexit()* must ensure that the *exit()* function is not called when the
1462 functions registered by the *atexit()* function are called.

1463 If a function registered by the *atexit()* function is called and a portable application needs to
1464 stop further *exit()* processing, it must call the *_exit()* function or the *_Exit()* function or one
1465 of the functions which cause abnormal process termination.

1466 Ref 7.22.4.3

1467 On page 619 line 21425 section *atexit()*, add *at_quick_exit* to the SEE ALSO section.

1468 Ref 7.16

1469 On page 624 line 21548 insert the following new *atomic_**() sections:

1470 **NAME**

1471 *atomic_compare_exchange_strong*, *atomic_compare_exchange_strong_explicit*,
1472 *atomic_compare_exchange_weak*, *atomic_compare_exchange_weak_explicit* — atomically

1473 compare and exchange the values of two objects

1474 SYNOPSIS

```
1475 #include <stdatomic.h>
1476 _Bool atomic_compare_exchange_strong(volatile A *object,
1477     C *expected, C desired);
1478 _Bool atomic_compare_exchange_strong_explicit(volatile A *object,
1479     C *expected, C desired, memory_order success,
1480     memory_order failure);
1481 _Bool atomic_compare_exchange_weak(volatile A *object,
1482     C *expected, C desired);
1483 _Bool atomic_compare_exchange_weak_explicit(volatile A *object,
1484     C *expected, C desired, memory_order success,
1485     memory_order failure);
```

1486 DESCRIPTION

1487 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1488 Any conflict between the requirements described here and the ISO C standard is
1489 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1490 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1491 `<stdatomic.h>` header nor support these generic functions.

1492 The `atomic_compare_exchange_strong_explicit()` generic function shall atomically compare
1493 the contents of the memory pointed to by *object* for equality with that pointed to by
1494 *expected*, and if true, shall replace the contents of the memory pointed to by *object*
1495 with *desired*, and if false, shall update the contents of the memory pointed to by *expected*
1496 with that pointed to by *object*. This operation shall be an atomic read-modify-write operation
1497 (see [xref to XBD 4.12.1]). If the comparison is true, memory shall be affected according to
1498 the value of *success*, and if the comparison is false, memory shall be affected according to
1499 the value of *failure*. The application shall ensure that *failure* is not
1500 `memory_order_release` nor `memory_order_acq_rel`, and shall ensure that *failure* is
1501 no stronger than *success*.

1502 The `atomic_compare_exchange_strong()` generic function shall be equivalent to
1503 `atomic_compare_exchange_strong_explicit()` called with *success* and *failure* both set to
1504 `memory_order_seq_cst`.

1505 The `atomic_compare_exchange_weak_explicit()` generic function shall be equivalent to
1506 `atomic_compare_exchange_strong_explicit()`, except that the compare-and-exchange
1507 operation may fail spuriously. That is, even when the contents of memory referred to by
1508 *expected* and *object* are equal, it may return zero and store back to *expected* the same
1509 memory contents that were originally there.

1510 The `atomic_compare_exchange_weak()` generic function shall be equivalent to
1511 `atomic_compare_exchange_weak_explicit()` called with *success* and *failure* both set to
1512 `memory_order_seq_cst`.

1513 RETURN VALUE

1514 These generic functions shall return the result of the comparison.

1515 ERRORS

1516 No errors are defined.

1517 **EXAMPLES**

1518 None.

1519 **APPLICATION USAGE**

1520 A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will
1521 be in a loop. For example:

```
1522     exp = atomic_load(&cur);  
1523     do {  
1524         des = function(exp);  
1525     } while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

1526 When a compare-and-exchange is in a loop, the weak version will yield better performance
1527 on some platforms. When a weak compare-and-exchange would require a loop and a strong
1528 one would not, the strong one is preferable.

1529 **RATIONALE**

1530 None.

1531 **FUTURE DIRECTIONS**

1532 None.

1533 **SEE ALSO**

1534 XBD Section 4.12.1, <stdatomic.h>

1535 **CHANGE HISTORY**

1536 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1537 **NAME**

1538 `atomic_exchange`, `atomic_exchange_explicit` — atomically exchange the value of an object

1539 **SYNOPSIS**

```
1540 #include <stdatomic.h>  
1541 C atomic_exchange(volatile A *object, C desired);  
1542 C atomic_exchange_explicit(volatile A *object,  
1543     C desired, memory_order order);
```

1544 **DESCRIPTION**

1545 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1546 Any conflict between the requirements described here and the ISO C standard is
1547 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1548 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1549 `<stdatomic.h>` header nor support these generic functions.

1550 The `atomic_exchange_explicit()` generic function shall atomically replace the value pointed
1551 to by *object* with *desired*. This operation shall be an atomic read-modify-write operation (see
1552 [xref to XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1553 The `atomic_exchange()` generic function shall be equivalent to `atomic_exchange_explicit()`
1554 called with *order* set to `memory_order_seq_cst`.

1555 **RETURN VALUE**

1556 These generic functions shall return the value pointed to by *object* immediately before the
1557 effects.

1558 **ERRORS**

1559 No errors are defined.

1560 **EXAMPLES**

1561 None.

1562 **APPLICATION USAGE**

1563 None.

1564 **RATIONALE**

1565 None.

1566 **FUTURE DIRECTIONS**

1567 None.

1568 **SEE ALSO**

1569 XBD Section 4.12.1, <**stdatomic.h**>

1570 **CHANGE HISTORY**

1571 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1572 **NAME**

1573 atomic_fetch_add, atomic_fetch_add_explicit, atomic_fetch_and,
1574 atomic_fetch_and_explicit, atomic_fetch_or, atomic_fetch_or_explicit, atomic_fetch_sub,
1575 atomic_fetch_sub_explicit, atomic_fetch_xor, atomic_fetch_xor_explicit — atomically
1576 replace the value of an object with the result of a computation

1577 **SYNOPSIS**

```
1578 #include <stdatomic.h>
1579 C   atomic_fetch_add(volatile A *object, M operand);
1580 C   atomic_fetch_add_explicit(volatile A *object, M operand,
1581                               memory_order order);
1582 C   atomic_fetch_and(volatile A *object, M operand);
1583 C   atomic_fetch_and_explicit(volatile A *object, M operand,
1584                               memory_order order);
1585 C   atomic_fetch_or(volatile A *object, M operand);
1586 C   atomic_fetch_or_explicit(volatile A *object, M operand,
1587                               memory_order order);
1588 C   atomic_fetch_sub(volatile A *object, M operand);
1589 C   atomic_fetch_sub_explicit(volatile A *object, M operand,
1590                               memory_order order);
1591 C   atomic_fetch_xor(volatile A *object, M operand);
1592 C   atomic_fetch_xor_explicit(volatile A *object, M operand,
1593                               memory_order order);
```

1594 **DESCRIPTION**

1595 [**CX**] The functionality described on this reference page is aligned with the ISO C standard.
1596 Any conflict between the requirements described here and the ISO C standard is

1597 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1598 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1599 `<stdatomic.h>` header nor support these generic functions.

1600 The `atomic_fetch_add_explicit()` generic function shall atomically replace the value pointed
1601 to by *object* with the result of adding *operand* to this value. This operation shall be an
1602 atomic read-modify-write operation (see [xref to XBD 4.12.1]). Memory shall be affected
1603 according to the value of *order*.

1604 The `atomic_fetch_add()` generic function shall be equivalent to `atomic_fetch_add_explicit()`
1605 called with *order* set to `memory_order_seq_cst`.

1606 The other `atomic_fetch_*`() generic functions shall be equivalent to
1607 `atomic_fetch_add_explicit()` if their name ends with *explicit*, or to `atomic_fetch_add()` if it
1608 does not, respectively, except that they perform the computation indicated in their name,
1609 instead of addition:

1610 *sub* subtraction
1611 *or* bitwise inclusive OR
1612 *xor* bitwise exclusive OR
1613 *and* bitwise AND

1614 For addition and subtraction, the application shall ensure that *A* is an atomic integer type or
1615 an atomic pointer type and is not `atomic_bool`. For the other operations, the application
1616 shall ensure that *A* is an atomic integer type and is not `atomic_bool`.

1617 For signed integer types, the computation shall silently wrap around on overflow; there are
1618 no undefined results. For pointer types, the result can be an undefined address, but the
1619 computations otherwise have no undefined behavior.

1620 RETURN VALUE

1621 These generic functions shall return the value pointed to by *object* immediately before the
1622 effects.

1623 ERRORS

1624 No errors are defined.

1625 EXAMPLES

1626 None.

1627 APPLICATION USAGE

1628 The operation of these generic functions is nearly equivalent to the operation of the
1629 corresponding compound assignment operators `+=`, `-=`, etc. The only differences are that the
1630 compound assignment operators are not guaranteed to operate atomically, and the value
1631 yielded by a compound assignment operator is the updated value of the object, whereas the
1632 value returned by these generic functions is the previous value of the atomic object.

1633 RATIONALE

1634 None.

1635 FUTURE DIRECTIONS

1636 None.

1637 **SEE ALSO**

1638 XBD Section 4.12.1, <stdatomic.h>

1639 **CHANGE HISTORY**

1640 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1641 **NAME**

1642 `atomic_flag_clear`, `atomic_flag_clear_explicit` — clear an atomic flag

1643 **SYNOPSIS**

```
1644 #include <stdatomic.h>
1645 void atomic_flag_clear(volatile atomic_flag *object);
1646 void atomic_flag_clear_explicit(
1647     volatile atomic_flag *object, memory_order order);
```

1648 **DESCRIPTION**

1649 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1650 Any conflict between the requirements described here and the ISO C standard is
1651 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1652 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1653 <stdatomic.h> header nor support these functions.

1654 The `atomic_flag_clear_explicit()` function shall atomically place the atomic flag pointed to
1655 by `object` into the clear state. Memory shall be affected according to the value of `order`,
1656 which the application shall ensure is not `memory_order_acquire` nor
1657 `memory_order_acq_rel`.

1658 The `atomic_flag_clear()` function shall be equivalent to `atomic_flag_clear_explicit()` called
1659 with `order` set to `memory_order_seq_cst`.

1660 **RETURN VALUE**

1661 These functions shall not return a value.

1662 **ERRORS**

1663 No errors are defined.

1664 **EXAMPLES**

1665 None.

1666 **APPLICATION USAGE**

1667 None.

1668 **RATIONALE**

1669 None.

1670 **FUTURE DIRECTIONS**

1671 None.

1672 **SEE ALSO**

1673 XBD Section 4.12.1, <stdatomic.h>

1674 **CHANGE HISTORY**

1675 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1676 **NAME**

1677 `atomic_flag_test_and_set`, `atomic_flag_test_and_set_explicit` — test and set an atomic flag

1678 **SYNOPSIS**

```
1679 #include <stdatomic.h>
1680 _Bool atomic_flag_test_and_set(volatile atomic_flag *object);
1681 _Bool atomic_flag_test_and_set_explicit(
1682     volatile atomic_flag *object, memory_order order);
```

1683 **DESCRIPTION**

1684 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1685 Any conflict between the requirements described here and the ISO C standard is
1686 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1687 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1688 <stdatomic.h> header nor support these functions.

1689 The `atomic_flag_test_and_set_explicit()` function shall atomically place the atomic flag
1690 pointed to by *object* into the set state and return the value corresponding to the immediately
1691 preceding state. This operation shall be an atomic read-modify-write operation (see [xref to
1692 XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1693 The `atomic_flag_test_and_set()` function shall be equivalent to
1694 `atomic_flag_test_and_set_explicit()` called with *order* set to `memory_order_seq_cst`.

1695 **RETURN VALUE**

1696 These functions shall return the value that corresponds to the state of the atomic flag
1697 immediately before the effects. The return value true shall correspond to the set state and the
1698 return value false shall correspond to the clear state.

1699 **ERRORS**

1700 No errors are defined.

1701 **EXAMPLES**

1702 None.

1703 **APPLICATION USAGE**

1704 None.

1705 **RATIONALE**

1706 None.

1707 **FUTURE DIRECTIONS**

1708 None.

1709 **SEE ALSO**

1710 XBD Section 4.12.1, <stdatomic.h>

1711 **CHANGE HISTORY**

1712 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1713 **NAME**

1714 `atomic_init` — initialize an atomic object

1715 **SYNOPSIS**

```
1716 #include <stdatomic.h>
1717 void atomic_init(volatile A *obj, C value);
```

1718 **DESCRIPTION**

1719 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1720 Any conflict between the requirements described here and the ISO C standard is
1721 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1722 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1723 `<stdatomic.h>` header nor support this generic function.

1724 The `atomic_init()` generic function shall initialize the atomic object pointed to by `obj` to the
1725 value `value`, while also initializing any additional state that the implementation might need
1726 to carry for the atomic object.

1727 Although this function initializes an atomic object, it does not avoid data races; concurrent
1728 access to the variable being initialized, even via an atomic operation, constitutes a data race.

1729 **RETURN VALUE**

1730 The `atomic_init()` generic function shall not return a value.

1731 **ERRORS**

1732 No errors are defined.

1733 **EXAMPLES**

```
1734 atomic_int guide;
1735 atomic_init(&guide, 42);
```

1736 **APPLICATION USAGE**

1737 None.

1738 **RATIONALE**

1739 None.

1740 **FUTURE DIRECTIONS**

1741 None.

1742 **SEE ALSO**

1743 XBD `<stdatomic.h>`

1744 **CHANGE HISTORY**

1745 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1746 **NAME**

1747 `atomic_is_lock_free` — indicate whether or not atomic operations are lock-free

1748 **SYNOPSIS**

1749 `#include <stdatomic.h>`
1750 `_Bool atomic_is_lock_free(const volatile A *obj);`

1751 **DESCRIPTION**

1752 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1753 Any conflict between the requirements described here and the ISO C standard is
1754 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1755 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1756 `<stdatomic.h>` header nor support this generic function.

1757 The `atomic_is_lock_free()` generic function shall indicate whether or not atomic operations
1758 on objects of the type pointed to by `obj` are lock-free; `obj` can be a null pointer.

1759 **RETURN VALUE**

1760 The `atomic_is_lock_free()` generic function shall return a non-zero value if and only if
1761 atomic operations on objects of the type pointed to by `obj` are lock-free. During the lifetime
1762 of the calling process, the result of the lock-free query shall be consistent for all pointers of
1763 the same type.

1764 **ERRORS**

1765 No errors are defined.

1766 **EXAMPLES**

1767 None.

1768 **APPLICATION USAGE**

1769 None.

1770 **RATIONALE**

1771 Operations that are lock-free should also be address-free. That is, atomic operations on the
1772 same memory location via two different addresses will communicate atomically. The
1773 implementation should not depend on any per-process state. This restriction enables
1774 communication via memory mapped into a process more than once and memory shared
1775 between two processes.

1776 **FUTURE DIRECTIONS**

1777 None.

1778 **SEE ALSO**

1779 XBD `<stdatomic.h>`

1780 **CHANGE HISTORY**

1781 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1782 **NAME**

1783 `atomic_load`, `atomic_load_explicit` — atomically obtain the value of an object

1784 **SYNOPSIS**

```
1785     #include <stdatomic.h>
1786     C atomic_load(const volatile A *object);
1787     C atomic_load_explicit(const volatile A *object,
1788         memory_order order);
```

1789 **DESCRIPTION**

1790 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1791 Any conflict between the requirements described here and the ISO C standard is
1792 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1793 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1794 `<stdatomic.h>` header nor support these generic functions.

1795 The `atomic_load_explicit()` generic function shall atomically obtain the value pointed to by
1796 `object`. Memory shall be affected according to the value of `order`, which the application shall
1797 ensure is not `memory_order_release` nor `memory_order_acq_rel`.

1798 The `atomic_load()` generic function shall be equivalent to `atomic_load_explicit()` called with
1799 `order` set to `memory_order_seq_cst`.

1800 **RETURN VALUE**

1801 These generic functions shall return the value pointed to by `object`.

1802 **ERRORS**

1803 No errors are defined.

1804 **EXAMPLES**

1805 None.

1806 **APPLICATION USAGE**

1807 None.

1808 **RATIONALE**

1809 None.

1810 **FUTURE DIRECTIONS**

1811 None.

1812 **SEE ALSO**

1813 XBD Section 4.12.1, `<stdatomic.h>`

1814 **CHANGE HISTORY**

1815 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1816 **NAME**

1817 `atomic_signal_fence`, `atomic_thread_fence` — fence operations

1818 **SYNOPSIS**

```
1819     #include <stdatomic.h>
1820     void atomic_signal_fence(memory_order order);
1821     void atomic_thread_fence(memory_order order);
```

1822 **DESCRIPTION**

1823 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1824 Any conflict between the requirements described here and the ISO C standard is
1825 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1826 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1827 `<stdatomic.h>` header nor support these functions.

1828 The `atomic_signal_fence()` and `atomic_thread_fence()` functions provide synchronization
1829 primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A
1830 fence with acquire semantics is called an *acquire fence*; a fence with release semantics is
1831 called a *release fence*.

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

1837 A release fence *A* synchronizes with an atomic operation *B* that performs an acquire
1838 operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is
1839 sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by
1840 any side effect in the hypothetical release sequence *X* would head if it were a release
1841 operation.

1842 An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with
1843 an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced
1844 before *B* and reads the value written by *A* or a value written by any side effect in the release
1845 sequence headed by *A*.

1846 Depending on the value of *order*, the operation performed by `atomic_thread_fence()` shall:

- 1847 • have no effects, if *order* is equal to `memory_order_relaxed`;
- 1848 • be an acquire fence, if *order* is equal to `memory_order_acquire` or
1849 `memory_order_consume`;
- 1850 • be a release fence, if *order* is equal to `memory_order_release`;
- 1851 • be both an acquire fence and a release fence, if *order* is equal to
1852 `memory_order_acq_rel`;
- 1853 • be a sequentially consistent acquire and release fence, if *order* is equal to
1854 `memory_order_seq_cst`.

1855 The `atomic_signal_fence()` function shall be equivalent to `atomic_thread_fence()`, except
1856 that the resulting ordering constraints shall be established only between a thread and a signal
1857 handler executed in the same thread.

1858 **RETURN VALUE**

1859 These functions shall not return a value.

1860 **ERRORS**

1861 No errors are defined.

1862 **EXAMPLES**

1863 None.

1864 **APPLICATION USAGE**

1865 The *atomic_signal_fence()* function can be used to specify the order in which actions
1866 performed by the thread become visible to the signal handler. Implementation reorderings of
1867 loads and stores are inhibited in the same way as with *atomic_thread_fence()*, but the
1868 hardware fence instructions that *atomic_thread_fence()* would have inserted are not
1869 emitted.

1870 **RATIONALE**

1871 None.

1872 **FUTURE DIRECTIONS**

1873 None.

1874 **SEE ALSO**

1875 XBD Section 4.12.1, <**stdatomic.h**>

1876 **CHANGE HISTORY**

1877 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1878 **NAME**

1879 *atomic_store*, *atomic_store_explicit* — atomically store a value in an object

1880 **SYNOPSIS**

```
1881 #include <stdatomic.h>  
1882 void atomic_store(volatile A *object, C desired);  
1883 void atomic_store_explicit(volatile A *object, C desired,  
1884 memory_order order);
```

1885 **DESCRIPTION**

1886 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1887 Any conflict between the requirements described here and the ISO C standard is
1888 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1889 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1890 <**stdatomic.h**> header nor support these generic functions.

1891 The *atomic_store_explicit()* generic function shall atomically replace the value pointed to by
1892 *object* with the value of *desired*. Memory shall be affected according to the value of *order*,
1893 which the application shall ensure is not `memory_order_acquire`,
1894 `memory_order_consume`, nor `memory_order_acq_rel`.

1895 The *atomic_store()* generic function shall be equivalent to *atomic_store_explicit()* called
1896 with *order* set to `memory_order_seq_cst`.

1897 **RETURN VALUE**

1898 These generic functions shall not return a value.

1899 **ERRORS**

1900 No errors are defined.

1901 **EXAMPLES**

1902 None.

1903 **APPLICATION USAGE**

1904 None.

1905 **RATIONALE**

1906 None.

1907 **FUTURE DIRECTIONS**

1908 None.

1909 **SEE ALSO**

1910 XBD Section 4.12.1, <**stdatomic.h**>

1911 **CHANGE HISTORY**

1912 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1913 Ref 7.28.1, 7.1.4 para 5

1914 On page 633 line 21891 insert a new `c16rtomb()` section:

1915 **NAME**

1916 `c16rtomb`, `c32rtomb` — convert a Unicode character code to a character (restartable)

1917 **SYNOPSIS**

1918

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

1919

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

1920

```
          mbstate_t *restrict ps);
```

1921

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

1922

```
          mbstate_t *restrict ps);
```

1923 **DESCRIPTION**

1924 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1925 Any conflict between the requirements described here and the ISO C standard is

1926 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1927 If *s* is a null pointer, the `c16rtomb()` function shall be equivalent to the call:

1928

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

1929 where *buf* is an internal buffer.

1930 If *s* is not a null pointer, the `c16rtomb()` function shall determine the number of bytes needed
1931 to represent the character that corresponds to the wide character given by *c16* (including any
1932 shift sequences), and store the resulting bytes in the array whose first element is pointed to
1933 by *s*. At most `{MB_CUR_MAX}` bytes shall be stored. If *c16* is a null wide character, a null
1934 byte shall be stored, preceded by any shift sequence needed to restore the initial shift state;

1935 the resulting state described shall be the initial conversion state.

1936 If *ps* is a null pointer, the *c16rtomb()* function shall use its own internal **mbstate_t** object,
1937 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
1938 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current
1939 conversion state of the associated character sequence.

1940 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

1941 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

1942 The *c32rtomb()* function shall behave the same way as *c16rtomb()* except that the second
1943 parameter shall be an object of type **char32_t** instead of **char16_t**. References to *c16* in the
1944 above description shall apply as if they were *c32* when they are being read as describing
1945 *c32rtomb()*.

1946 If called with a null *ps* argument, the *c16rtomb()* function need not be thread-safe; however,
1947 such calls shall avoid data races with calls to *c16rtomb()* with a non-null argument and with
1948 calls to all other functions.

1949 If called with a null *ps* argument, the *c32rtomb()* function need not be thread-safe; however,
1950 such calls shall avoid data races with calls to *c32rtomb()* with a non-null argument and with
1951 calls to all other functions.

1952 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
1953 calls *c16rtomb()* or *c32rtomb()* with a null pointer for *ps*.

1954 **RETURN VALUE**

1955 These functions shall return the number of bytes stored in the array object (including any
1956 shift sequences). When *c16* or *c32* is not a valid wide character, an encoding error shall
1957 occur. In this case, the function shall store the value of the macro [EILSEQ] in *errno* and
1958 shall return (**size_t**)-1; the conversion state is unspecified.

1959 **ERRORS**

1960 These function shall fail if:

1961 [EILSEQ] An invalid wide-character code is detected.

1962 These functions may fail if:

1963 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

1964 **EXAMPLES**

1965 None.

1966 **APPLICATION USAGE**

1967 None.

1968 **RATIONALE**

1969 None.

1970 **FUTURE DIRECTIONS**

1971 None.

- 1972 **SEE ALSO**
1973 *mbrtoc16*
- 1974 XBD <**uchar.h**>
- 1975 **CHANGE HISTORY**
1976 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.
- 1977 Ref G.6 para 6, F.10.4.3, F.10.4.2, F.10 para 11
1978 On page 633 line 21905 section *cabs()*, add:
- 1979 [MXC]*cabs(x + iy)*, *cabs(y + ix)*, and *cabs(x - iy)* shall return exactly the same value.
- 1980 If z is $\pm 0 \pm i0$, $+0$ shall be returned.
- 1981 If the real or imaginary part of z is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned, even if the other part is NaN.
- 1982 If the real or imaginary part of z is NaN and the other part is not $\pm\text{Inf}$, NaN shall be returned.
1983 [/MXC]
- 1984 Ref G.6.1.1
1985 On page 634 line 21935 section *cacos()*, add:
- 1986 [MXC]*cacos(conj(z))*, *cacosf(conjf(z))* and *cacosl(conjl(z))* shall return exactly the same
1987 value as *conj(cacos(z))*, *conjf(cacosf(z))* and *conjl(cacosl(z))*, respectively, including for the
1988 special values of z below.
- 1989 If z is $\pm 0 + i0$, $\pi/2 - i0$ shall be returned.
- 1990 If z is $\pm 0 + i\text{NaN}$, $\pi/2 + i\text{NaN}$ shall be returned.
- 1991 If z is $x + i\text{Inf}$ where x is finite, $\pi/2 - i\text{Inf}$ shall be returned.
- 1992 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
1993 floating-point exception may be raised.
- 1994 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $\pi - i\text{Inf}$ shall be returned.
- 1995 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 - i\text{Inf}$ shall be returned.
- 1996 If z is $-\text{Inf} + i\text{Inf}$, $3\pi/4 - i\text{Inf}$ shall be returned.
- 1997 If z is $+\text{Inf} + i\text{Inf}$, $\pi/4 - i\text{Inf}$ shall be returned.
- 1998 If z is $\pm\text{Inf} + i\text{NaN}$, $\text{NaN} \pm i\text{Inf}$ shall be returned; the sign of the imaginary part of the result
1999 is unspecified.
- 2000 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2001 point exception may be raised.
- 2002 If z is $\text{NaN} + i\text{Inf}$, $\text{NaN} - i\text{Inf}$ shall be returned.

2003 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} - i\text{NaN}$ shall be returned.[/MXC]

2004 Ref G.6.2.1

2005 On page 635 line 21966 section `cacosh()`, add:

2006 [MXC]`cacosh(conj(z))`, `cacoshf(conjf(z))` and `cacoshl(conjl(z))` shall return exactly the same
2007 value as `conj(cacosh(z))`, `conjf(cacoshf(z))` and `conjl(cacoshl(z))`, respectively, including for
2008 the special values of z below.

2009 If z is $\pm 0 + i0$, $+0 + i\pi/2$ shall be returned.

2010 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2011 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
2012 unspecified.

2013 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
2014 floating-point exception may be raised.

2015 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2016 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2017 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

2018 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2019 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2020 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2021 point exception may be raised.

2022 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2023 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2024 Ref 7.26.2.1

2025 On page 637 line 21989 insert the following new `call_once()` section:

2026 **NAME**

2027 `call_once` — dynamic package initialization

2028 **SYNOPSIS**

2029 `#include <threads.h>`

2030 `void call_once(once_flag *flag, void (*init_routine)(void));`
2031 `once_flag flag = ONCE_FLAG_INIT;`

2032 **DESCRIPTION**

2033 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2034 Any conflict between the requirements described here and the ISO C standard is
2035 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2036 The *call_once()* function shall use the **once_flag** pointed to by *flag* to ensure that
2037 *init_routine* is called exactly once, the first time the *call_once()* function is called with that
2038 value of *flag*. Completion of an effective call to the *call_once()* function shall synchronize
2039 with all subsequent calls to the *call_once()* function with the same value of *flag*.

2040 [CX]The *call_once()* function is not a cancellation point. However, if *init_routine* is a
2041 cancellation point and is canceled, the effect on *flag* shall be as if *call_once()* was never
2042 called.

2043 If the call to *init_routine* is terminated by a call to *longjmp()* or *siglongjmp()*, the behavior is
2044 undefined.

2045 The behavior of *call_once()* is undefined if *flag* has automatic storage duration or is not
2046 initialized by `ONCE_FLAG_INIT`.

2047 The *call_once()* function shall not be affected if the calling thread executes a signal handler
2048 during the call.[/CX]

2049 **RETURN VALUE**

2050 The *call_once()* function shall not return a value.

2051 **ERRORS**

2052 No errors are defined.

2053 **EXAMPLES**

2054 None.

2055 **APPLICATION USAGE**

2056 If *init_routine* recursively calls *call_once()* with the same *flag*, the recursive call will not call
2057 the specified *init_routine*, and thus the specified *init_routine* will not complete, and thus the
2058 recursive call to *call_once()* will not return. Use of *longjmp()* or *siglongjmp()* within an
2059 *init_routine* to jump to a point outside of *init_routine* prevents *init_routine* from returning.

2060 **RATIONALE**

2061 For dynamic library initialization in a multi-threaded process, if an initialization flag is used
2062 the flag needs to be protected against modification by multiple threads simultaneously
2063 calling into the library. This can be done by using a statically-initialized mutex. However,
2064 the better solution is to use *call_once()* or *pthread_once()* which are designed for exactly
2065 this purpose, for example:

```
2066 #include <threads.h>  
2067 static once_flag random_is_initialized = ONCE_FLAG_INIT;  
2068 extern void initialize_random(void);
```

```
2069 int random_function()  
2070 {  
2071     call_once(&random_is_initialized, initialize_random);  
2072     ...  
2073     /* Operations performed after initialization. */  
2074 }
```

2075 The *call_once()* function is not affected by signal handlers for the reasons stated in [xref to
2076 XRAT B.2.3].

2077 **FUTURE DIRECTIONS**

2078 None.

2079 **SEE ALSO**

2080 *pthread_once*

2081 XBD Section 4.12.2, <**threads.h**>

2082 **CHANGE HISTORY**

2083 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2084 Ref 7.22.3 para 1

2085 On page 637 line 22002 section *calloc()*, change:

2086 a pointer to any type of object

2087 to:

2088 a pointer to any type of object with a fundamental alignment requirement

2089 Ref 7.22.3 para 1

2090 On page 637 line 22007 section *calloc()*, change:

2091 either a null pointer shall be returned, or ...

2092 to:

2093 either a null pointer shall be returned to indicate an error, or ...

2094 Ref 7.22.3 para 2

2095 On page 637 line 22008 section *calloc()*, add a new paragraph:

2096 For purposes of determining the existence of a data race, *calloc()* shall behave as though it
2097 accessed only memory locations accessible through its arguments and not other static
2098 duration storage. The function may, however, visibly modify the storage that it allocates.
2099 Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix_memalign()*, [/ADV] and
2100 *realloc()* that allocate or deallocate a particular region of memory shall occur in a single total
2101 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
2102 next allocation (if any) in this order.

2103 Ref 7.22.3.1

2104 On page 637 line 22029 section *calloc()*, add *aligned_alloc* to the SEE ALSO section.

2105 Ref G.6 para 6, F.10.1.4, F.10 para 11

2106 On page 639 line 22055 section *carg()*, add:

2107 [MXC]If z is $-0 \pm i0$, $\pm\pi$ shall be returned.

- 2108 If z is $+0 \pm i0$, ± 0 shall be returned.
- 2109 If z is $x \pm i0$ where x is negative, $\pm\pi$ shall be returned.
- 2110 If z is $x \pm i0$ where x is positive, ± 0 shall be returned.
- 2111 If z is $\pm 0 + iy$ where y is negative, $-\pi/2$ shall be returned.
- 2112 If z is $\pm 0 + iy$ where y is positive, $\pi/2$ shall be returned.
- 2113 If z is $-\text{Inf} \pm iy$ where y is positive and finite, $\pm\pi$ shall be returned.
- 2114 If z is $+\text{Inf} \pm iy$ where y is positive and finite, ± 0 shall be returned.
- 2115 If z is $x \pm i\text{Inf}$ where x is finite, $\pm\pi/2$ shall be returned.
- 2116 If z is $-\text{Inf} \pm i\text{Inf}$, $\pm 3\pi/4$ shall be returned.
- 2117 If z is $+\text{Inf} \pm i\text{Inf}$, $\pm\pi/4$ shall be returned.
- 2118 If the real or imaginary part of z is NaN, NaN shall be returned.[/MXC]
- 2119 Ref G.6 para 7, G.6.2.2
- 2120 On page 640 line 22086 section `casin()`, add:
- 2121 [MXC]`casin(conj(iz))`, `casinf(conjf(iz))` and `casinl(conjl(iz))` shall return exactly the same
- 2122 value as `conj(casin(iz))`, `conjf(casinf(iz))` and `conjl(casinl(iz))`, respectively, and `casin(-iz)`,
- 2123 `casinf(-iz)` and `casinl(-iz)` shall return exactly the same value as `-casin(iz)`, `-casinf(iz)` and
- 2124 `-casinl(iz)`, respectively, including for the special values of iz below.
- 2125 If iz is $+0 + i0$, $-i (0 + i0)$ shall be returned.
- 2126 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i (+\text{Inf} + i\pi/2)$ shall be returned.
- 2127 If iz is $x + i\text{NaN}$ where x is finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the invalid
- 2128 floating-point exception may be raised.
- 2129 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i (+\text{Inf} + i0)$ shall be returned.
- 2130 If iz is $+\text{Inf} + i\text{Inf}$, $-i (+\text{Inf} + i\pi/4)$ shall be returned.
- 2131 If iz is $+\text{Inf} + i\text{NaN}$, $-i (+\text{Inf} + i\text{NaN})$ shall be returned.
- 2132 If iz is $\text{NaN} + i0$, $-i (\text{NaN} + i0)$ shall be returned.
- 2133 If iz is $\text{NaN} + iy$ where y is non-zero and finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
- 2134 invalid floating-point exception may be raised.
- 2135 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
- 2136 result is unspecified.
- 2137 If iz is $\text{NaN} + i\text{NaN}$, $-i (\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2138 Ref G.6 para 7
2139 On page 640 line 22094 section `casin()`, change RATIONALE from:

2140 None.

2141 to:

2142 The MXC special cases for `casin()` are derived from those for `casinh()` by applying the
2143 formula $\text{casin}(z) = -i \text{casinh}(iz)$.

2144 Ref G.6.2.2
2145 On page 641 line 22118 section `casinh()`, add:

2146 [MXC]`casinh(conj(z))`, `casinhf(conjf(z))` and `casinhl(conjhl(z))` shall return exactly the same
2147 value as `conj(casinh(z))`, `conjf(casinhf(z))` and `conjl(casinhl(z))`, respectively, and `casinh(-z)`,
2148 `casinhf(-z)` and `casinhl(-z)` shall return exactly the same value as $-\text{casinh}(z)$, $-\text{casinhf}(z)$
2149 and $-\text{casinhl}(z)$, respectively, including for the special values of z below.

2150 If z is $+0 + i0$, $0 + i0$ shall be returned.

2151 If z is $x + i\text{Inf}$ where x is positive-signed and finite, $+\text{Inf} + i\pi/2$ shall be returned.

2152 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2153 point exception may be raised.

2154 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2155 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2156 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2157 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2158 If z is $\text{NaN} + iy$ where y is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2159 floating-point exception may be raised.

2160 If z is $\text{NaN} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned; the sign of the real part of the result is
2161 unspecified.

2162 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2163 Ref G.6 para 7, G.6.2.3
2164 On page 643 line 22157 section `catan()`, add:

2165 [MXC]`catan(conj(iz))`, `catanf(conjf(iz))` and `catanl(conjl(iz))` shall return exactly the same
2166 value as `conj(catan(iz))`, `conjf(catanf(iz))` and `conjl(catanl(iz))`, respectively, and `catan(-iz)`,
2167 `catanf(-iz)` and `catanl(-iz)` shall return exactly the same value as $-\text{catan}(iz)$, $-\text{catanf}(iz)$ and
2168 $-\text{catanl}(iz)$, respectively, including for the special values of iz below.

2169 If iz is $+0 + i0$, $-i(+0 + i0)$ shall be returned.

- 2170 If iz is $+0 + iNaN$, $-i (+0 + iNaN)$ shall be returned.
- 2171 If iz is $+1 + i0$, $-i (+Inf + i0)$ shall be returned and the divide-by-zero floating-point
2172 exception shall be raised.
- 2173 If iz is $x + iInf$ where x is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2174 If iz is $x + iNaN$ where x is non-zero and finite, $-i (NaN + iNaN)$ shall be returned and the
2175 invalid floating-point exception may be raised.
- 2176 If iz is $+Inf + iy$ where y is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2177 If iz is $+Inf + iInf$, $-i (+0 + i\pi/2)$ shall be returned.
- 2178 If iz is $+Inf + iNaN$, $-i (+0 + iNaN)$ shall be returned.
- 2179 If iz is $NaN + iy$ where y is finite, $-i (NaN + iNaN)$ shall be returned and the invalid
2180 floating-point exception may be raised.
- 2181 If iz is $NaN + iInf$, $-i (\pm 0 + i\pi/2)$ shall be returned; the sign of the imaginary part of the
2182 result is unspecified.
- 2183 If iz is $NaN + iNaN$, $-i (NaN + iNaN)$ shall be returned.[/MXC]
- 2184 Ref G.6 para 7
2185 On page 643 line 22165 section `catan()`, change RATIONALE from:
- 2186 None.
- 2187 to:
- 2188 The MXC special cases for `catan()` are derived from those for `catanh()` by applying the
2189 formula $catan(z) = -i \operatorname{catanh}(iz)$.
- 2190 Ref G.6.2.3
2191 On page 644 line 22189 section `catanh`, add:
- 2192 [MXC]`catanh(conj(z))`, `catanhf(conj(z))` and `catanhl(conj(z))` shall return exactly the same
2193 value as `conj(catanh(z))`, `conjf(catanhf(z))` and `conjl(catanhl(z))`, respectively, and
2194 `catanh(-z)`, `catanhf(-z)` and `catanhl(-z)` shall return exactly the same value as $-catanh(z)$,
2195 $-catanhf(z)$ and $-catanhl(z)$, respectively, including for the special values of z below.
- 2196 If z is $+0 + i0$, $+0 + i0$ shall be returned.
- 2197 If z is $+0 + iNaN$, $+0 + iNaN$ shall be returned.
- 2198 If z is $+1 + i0$, $+Inf + i0$ shall be returned and the divide-by-zero floating-point exception
2199 shall be raised.
- 2200 If z is $x + iInf$ where x is positive-signed and finite, $+0 + i\pi/2$ shall be returned.
- 2201 If z is $x + iNaN$ where x is non-zero and finite, $NaN + iNaN$ shall be returned and the invalid

- 2202 floating-point exception may be raised.
- 2203 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\pi/2$ shall be returned.
- 2204 If z is $+\text{Inf} + i\text{Inf}$, $+0 + i\pi/2$ shall be returned.
- 2205 If z is $+\text{Inf} + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.
- 2206 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-point exception may be raised.
- 2207
- 2208 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 unspecified.
- 2209
- 2210 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2211 Ref G.6 para 7, G.6.2.4
- 2212 On page 652 line 22426 section `ccos()`, add:
- 2213 [MXC]`ccos(conj(iz))`, `ccosf(conjf(iz))` and `ccosl(conjl(iz))` shall return exactly the same value
- 2214 as `conj(ccos(iz))`, `conjf(ccosf(iz))` and `conjl(ccosl(iz))`, respectively, and `ccos(-iz)`, `ccosf(-iz)`
- 2215 and `ccosl(-iz)` shall return exactly the same value as `ccos(iz)`, `ccosf(iz)` and `ccosl(iz)`,
- 2216 respectively, including for the special values of iz below.
- 2217 If iz is $+0 + i0$, $1 + i0$ shall be returned.
- 2218 If iz is $+0 + i\text{Inf}$, $\text{NaN} \pm i0$ shall be returned and the invalid floating-point exception shall be
- 2219 raised; the sign of the imaginary part of the result is unspecified.
- 2220 If iz is $+0 + i\text{NaN}$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
- 2221 unspecified.
- 2222 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
- 2223 floating-point exception shall be raised.
- 2224 If iz is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the
- 2225 invalid floating-point exception may be raised.
- 2226 If iz is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.
- 2227 If iz is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.
- 2228 If iz is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
- 2229 shall be raised; the sign of the real part of the result is unspecified.
- 2230 If iz is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2231 If iz is $\text{NaN} + i0$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
- 2232 unspecified.
- 2233 If iz is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
- 2234 invalid floating-point exception may be raised.

2235 If iz is NaN + i NaN, NaN + i NaN shall be returned.[/MXC]

2236 Ref G.6 para 7
2237 On page 652 line 22434 section `ccos()`, change RATIONALE from:

2238 None.

2239 to:

2240 The MXC special cases for `ccos()` are derived from those for `ccosh()` by applying the
2241 formula $ccos(z) = ccosh(iz)$.

2242 Ref G.6.2.4
2243 On page 653 line 22455 section `ccosh()`, add:

2244 [MXC]`ccosh(conj(z))`, `ccoshf(conjf(z))` and `ccoshl(conjl(z))` shall return exactly the same
2245 value as `conj(ccosh(z))`, `conjf(ccoshf(z))` and `conjl(ccoshl(z))`, respectively, and `ccosh(-z)`,
2246 `ccoshf(-z)` and `ccoshl(-z)` shall return exactly the same value as `ccosh(z)`, `ccoshf(z)` and
2247 `ccoshl(z)`, respectively, including for the special values of z below.

2248 If z is $+0 + i0$, $1 + i0$ shall be returned.

2249 If z is $+0 + i\text{Inf}$, NaN $\pm i0$ shall be returned and the invalid floating-point exception shall be
2250 raised; the sign of the imaginary part of the result is unspecified.

2251 If z is $+0 + i\text{NaN}$, NaN $\pm i0$ shall be returned; the sign of the imaginary part of the result is
2252 unspecified.

2253 If z is $x + i\text{Inf}$ where x is non-zero and finite, NaN + i NaN shall be returned and the invalid
2254 floating-point exception shall be raised.

2255 If z is $x + i\text{NaN}$ where x is non-zero and finite, NaN + i NaN shall be returned and the invalid
2256 floating-point exception may be raised.

2257 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2258 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2259 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2260 shall be raised; the sign of the real part of the result is unspecified.

2261 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2262 If z is NaN + $i0$, NaN $\pm i0$ shall be returned; the sign of the imaginary part of the result is
2263 unspecified.

2264 If z is NaN + iy where y is any non-zero number, NaN + i NaN shall be returned and the
2265 invalid floating-point exception may be raised.

2266 If z is NaN + i NaN, NaN + i NaN shall be returned.[/MXC]

2267 Ref F.10.6.1 para 4
2268 On page 655 line 22489 section `ceil()`, add a new paragraph:

2269 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2270 arguments.[/MX]

2271 Ref F.10.6.1 para 2
2272 On page 655 line 22491 section `ceil()`, change:

2273 [MX]The result shall have the same sign as x .[/MX]

2274 to:

2275 [MX]The returned value shall be independent of the current rounding direction mode and
2276 shall have the same sign as x .[/MX]

2277 Ref F.10.6.1 para 4
2278 On page 655 line 22504 section `ceil()`, delete from APPLICATION USAGE:

2279 These functions may raise the inexact floating-point exception if the result differs in value
2280 from the argument.

2281 Ref G.6.3.1
2282 On page 657 line 22539 section `cexp()`, add:

2283 [MXC]`cexp(conj(z))`, `cexpf(conjf(z))` and `cexpl(conjl(z))` shall return exactly the same value
2284 as `conj(cexp(z))`, `conjf(cexpf(z))` and `conjl(cexpl(z))`, respectively, including for the special
2285 values of z below.

2286 If z is $\pm 0 + i0$, $1 + i0$ shall be returned.

2287 If z is $x + i\text{Inf}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-point
2288 exception shall be raised.

2289 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2290 point exception may be raised.

2291 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2292 If z is $-\text{Inf} + iy$ where y is finite, $+0 (\cos(y) + i\sin(y))$ shall be returned.

2293 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2294 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
2295 result are unspecified.

2296 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2297 shall be raised; the sign of the real part of the result is unspecified.

2298 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
2299 result are unspecified.

2300 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
2301 unspecified.

2302 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2303 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2304 invalid floating-point exception may be raised.

2305 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2306 Ref 7.26.5.7
2307 On page 679 line 23268 section `clock_getres()`, change:

2308 including the `nanosleep()` function

2309 to:

2310 including the `nanosleep()` and `thrd_sleep()` functions

2311 Ref G.6.3.2
2312 On page 687 line 23495 section `clog()`, add:

2313 [MXC]`clog(conj(z))`, `clogf(conjf(z))` and `clogl(conjl(z))` shall return exactly the same value as
2314 `conj(clog(z))`, `conjf(clogf(z))` and `conjl(clogl(z))`, respectively, including for the special
2315 values of z below.

2316 If z is $-0 + i0$, $-\text{Inf} + i\pi$ shall be returned and the divide-by-zero floating-point exception
2317 shall be raised.

2318 If z is $+0 + i0$, $-\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2319 shall be raised.

2320 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2321 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2322 point exception may be raised.

2323 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2324 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2325 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

2326 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2327 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2328 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2329 point exception may be raised.

2330 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2331 If z is NaN + i NaN, NaN + i NaN shall be returned.[/MXC]

2332 Ref 7.26.3

2333 On page 698 line 23854 insert the following new `cond_*`() sections:

2334 [Note to reviewers: changes to `cond_broadcast` and `cond_signal` may be needed depending on the](#)
2335 [outcome of Mantis bug 609.](#)

2336 **NAME**

2337 `cond_broadcast`, `cond_signal` — broadcast or signal a condition

2338 **SYNOPSIS**

2339 `#include <threads.h>`

2340 `int cond_broadcast(cond_t *cond);`

2341 `int cond_signal(cond_t *cond);`

2342 **DESCRIPTION**

2343 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2344 Any conflict between the requirements described here and the ISO C standard is
2345 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2346 The `cond_broadcast()` function shall unblock all of the threads that are blocked on the
2347 condition variable pointed to by `cond` at the time of the call.

2348 The `cond_signal()` function shall unblock one of the threads that are blocked on the condition
2349 variable pointed to by `cond` at the time of the call (if any threads are blocked on `cond`).

2350 If no threads are blocked on the condition variable pointed to by `cond` at the time of the call,
2351 these functions shall have no effect and shall return `thrd_success`.

2352 [CX]If more than one thread is blocked on a condition variable, the scheduling policy shall
2353 determine the order in which threads are unblocked. When each thread unblocked as a result
2354 of a `cond_broadcast()` or `cond_signal()` returns from its call to `cond_wait()` or `cond_timedwait()`,
2355 the thread shall own the mutex with which it called `cond_wait()` or `cond_timedwait()`. The
2356 thread(s) that are unblocked shall contend for the mutex according to the scheduling policy
2357 (if applicable), and as if each had called `mtx_lock()`.

2358 The `cond_broadcast()` and `cond_signal()` functions can be called by a thread whether or not it
2359 currently owns the mutex that threads calling `cond_wait()` or `cond_timedwait()` have associated
2360 with the condition variable during their waits; however, if predictable scheduling behavior is
2361 required, then that mutex shall be locked by the thread calling `cond_broadcast()` or
2362 `cond_signal()`.

2363 These functions shall not be affected if the calling thread executes a signal handler during
2364 the call.[/CX]

2365 The behavior is undefined if the value specified by the `cond` argument to `cond_broadcast()` or
2366 `cond_signal()` does not refer to an initialized condition variable.

2367 **RETURN VALUE**

2368 These functions shall return `thrd_success` on success, or `thrd_error` if the request
2369 could not be honored.

2370 **ERRORS**

2371 No errors are defined.

2372 **EXAMPLES**

2373 None.

2374 **APPLICATION USAGE**

2375 See the APPLICATION USAGE section for *pthread_cond_broadcast()*, substituting
2376 *cond_broadcast()* for *pthread_cond_broadcast()* and *cond_signal()* for *pthread_cond_signal()*.

2377 **RATIONALE**

2378 As for *pthread_cond_broadcast()* and *pthread_cond_signal()*, spurious wakeups may occur
2379 with *cond_broadcast()* and *cond_signal()*, necessitating that applications code a predicate-
2380 testing-loop around the condition wait. (See the RATIONALE section for
2381 *pthread_cond_broadcast()*.)

2382 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2383 B.2.3].

2384 **FUTURE DIRECTIONS**

2385 None.

2386 **SEE ALSO**

2387 *cond_destroy*, *cond_timedwait*, *pthread_cond_broadcast*

2388 XBD Section 4.12.2, <**threads.h**>

2389 **CHANGE HISTORY**

2390 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2391 **NAME**

2392 *cond_destroy*, *cond_init* — destroy and initialize condition variables

2393 **SYNOPSIS**

2394 `#include <threads.h>`

2395 `void cond_destroy(cond_t *cond);`

2396 `int cond_init(cond_t *cond);`

2397 **DESCRIPTION**

2398 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2399 Any conflict between the requirements described here and the ISO C standard is
2400 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2401 The *cond_destroy()* function shall release all resources used by the condition variable pointed
2402 to by *cond*. It shall be safe to destroy an initialized condition variable upon which no threads
2403 are currently blocked. Attempting to destroy a condition variable upon which other threads
2404 are currently blocked results in undefined behavior. A destroyed condition variable object
2405 can be reinitialized using *cond_init()*; the results of otherwise referencing the object after it
2406 has been destroyed are undefined. The behavior is undefined if the value specified by the
2407 *cond* argument to *cond_destroy()* does not refer to an initialized condition variable.

2408 The *cond_init()* function shall initialize a condition variable. If it succeeds it shall set the
2409 variable pointed to by *cond* to a value that uniquely identifies the newly initialized condition
2410 variable. Attempting to initialize an already initialized condition variable results in
2411 undefined behavior. A thread that calls *cond_wait()* on a newly initialized condition variable
2412 shall block.

2413 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
2414 further requirements.

2415 These functions shall not be affected if the calling thread executes a signal handler during
2416 the call.[/CX]

2417 **RETURN VALUE**

2418 The *cond_destroy()* function shall not return a value.

2419 The *cond_init()* function shall return *thrd_success* on success, or *thrd_nomem* if no
2420 memory could be allocated for the newly created condition, or *thrd_error* if the request
2421 could not be honored.

2422 **ERRORS**

2423 See RETURN VALUE.

2424 **EXAMPLES**

2425 None.

2426 **APPLICATION USAGE**

2427 None.

2428 **RATIONALE**

2429 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2430 B.2.3].

2431 **FUTURE DIRECTIONS**

2432 None.

2433 **SEE ALSO**

2434 *cond_broadcast*, *cond_timedwait*

2435 XBD <**threads.h**>

2436 **CHANGE HISTORY**

2437 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2438 **NAME**

2439 *cond_timedwait*, *cond_wait* — wait on a condition

2440 **SYNOPSIS**

```
2441 #include <threads.h>
2442 int cond_timedwait(cond_t * restrict cond, mtx_t * restrict mtx,
2443                  const struct timespec * restrict ts);
2444 int cond_wait(cond_t *cond, mtx_t *mtx);
```

2445 **DESCRIPTION**

2446 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2447 Any conflict between the requirements described here and the ISO C standard is
2448 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2449 The *cnd_timedwait()* function shall atomically unlock the mutex pointed to by *mtx* and block
2450 until the condition variable pointed to by *cond* is signaled by a call to *cnd_signal()* or to
2451 *cnd_broadcast()*, or until after the TIME_UTC-based calendar time pointed to by *ts*, or until
2452 it is unblocked due to an unspecified reason.

2453 The *cnd_wait()* function shall atomically unlock the mutex pointed to by *mtx* and block until
2454 the condition variable pointed to by *cond* is signaled by a call to *cnd_signal()* or to
2455 *cnd_broadcast()*, or until it is unblocked due to an unspecified reason.

2456 [CX]Atomically here means "atomically with respect to access by another thread to the
2457 mutex and then the condition variable". That is, if another thread is able to acquire the mutex
2458 after the about-to-block thread has released it, then a subsequent call to *cnd_broadcast()* or
2459 *cnd_signal()* in that thread shall behave as if it were issued after the about-to-block thread
2460 has blocked.[/CX]

2461 When the calling thread becomes unblocked, these functions shall lock the mutex pointed to
2462 by *mtx* before they return. The application shall ensure that the mutex pointed to by *mtx* is
2463 locked by the calling thread before it calls these functions.

2464 When using condition variables there is always a Boolean predicate involving shared
2465 variables associated with each condition wait that is true if the thread should proceed.
2466 Spurious wakeups from the *cnd_timedwait()* and *cnd_wait()* functions may occur. Since the
2467 return from *cnd_timedwait()* or *cnd_wait()* does not imply anything about the value of this
2468 predicate, the predicate should be re-evaluated upon such return.

2469 When a thread waits on a condition variable, having specified a particular mutex to either
2470 the *cnd_timedwait()* or the *cnd_wait()* operation, a dynamic binding is formed between that
2471 mutex and condition variable that remains in effect as long as at least one thread is blocked
2472 on the condition variable. During this time, the effect of an attempt by any thread to wait on
2473 that condition variable using a different mutex is undefined. Once all waiting threads have
2474 been unblocked (as by the *cnd_broadcast()* operation), the next wait operation on
2475 that condition variable shall form a new dynamic binding with the mutex specified by that
2476 wait operation. Even though the dynamic binding between condition variable and mutex
2477 might be removed or replaced between the time a thread is unblocked from a wait on the
2478 condition variable and the time that it returns to the caller or begins cancellation cleanup, the
2479 unblocked thread shall always re-acquire the mutex specified in the condition wait operation
2480 call from which it is returning.

2481 [CX]A condition wait (whether timed or not) is a cancellation point. When the cancelability
2482 type of a thread is set to PTHREAD_CANCEL_DEFERRED, a side-effect of acting upon a
2483 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired
2484 before calling the first cancellation cleanup handler. The effect is as if the thread were
2485 unblocked, allowed to execute up to the point of returning from the call to *cnd_timedwait()*
2486 or *cnd_wait()*, but at that point notices the cancellation request and instead of returning to
2487 the caller of *cnd_timedwait()* or *cnd_wait()*, starts the thread cancellation activities, which
2488 includes calling cancellation cleanup handlers.

2489 A thread that has been unblocked because it has been canceled while blocked in a call to
2490 *cond_timedwait()* or *cond_wait()* shall not consume any condition signal that may be directed
2491 concurrently at the condition variable if there are other threads blocked on the condition
2492 variable.[/CX]

2493 When *cond_timedwait()* times out, it shall nonetheless release and re-acquire the mutex
2494 referenced by *mutex*, and may consume a condition signal directed concurrently at the
2495 condition variable.

2496 [CX]These functions shall not be affected if the calling thread executes a signal handler
2497 during the call, except that if a signal is delivered to a thread waiting for a condition
2498 variable, upon return from the signal handler either the thread shall resume waiting for the
2499 condition variable as if it was not interrupted, or it shall return *thrd_success* due to
2500 spurious wakeup.[/CX]

2501 The behavior is undefined if the value specified by the *cond* or *mtx* argument to these
2502 functions does not refer to an initialized condition variable or an initialized mutex object,
2503 respectively.

2504 **RETURN VALUE**

2505 The *cond_timedwait()* function shall return *thrd_success* upon success, or
2506 *thrd_timedout* if the time specified in the call was reached without acquiring the
2507 requested resource, or *thrd_error* if the request could not be honored.

2508 The *cond_wait()* function shall return *thrd_success* upon success or *thrd_error* if the
2509 request could not be honored.

2510 **ERRORS**

2511 See RETURN VALUE.

2512 **EXAMPLES**

2513 None.

2514 **APPLICATION USAGE**

2515 None.

2516 **RATIONALE**

2517 These functions are not affected by signal handlers (except as stated in the DESCRIPTION)
2518 for the reasons stated in [xref to XRAT B.2.3].

2519 **FUTURE DIRECTIONS**

2520 None.

2521 **SEE ALSO**

2522 *cond_broadcast*, *cond_destroy*, *timespec_get*

2523 XBD Section 4.12.2, <**threads.h**>

2524 **CHANGE HISTORY**

2525 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2526 Ref F.10.8.1 para 2
2527 On page 705 line 24155 section `copysign()`, add a new paragraph:

2528 [MX]The returned value shall be exact and shall be independent of the current rounding
2529 direction mode.[/MX]

2530 Ref G.6.4.1 para 1
2531 On page 711 line 24308 section `cpow()`, add a new paragraph:

2532 [MXC]These functions shall raise floating-point exceptions if appropriate for the calculation
2533 of the parts of the result, and may also raise spurious floating-point exceptions.[/MXC]

2534 Ref G.6.4.1 footnote 386
2535 On page 711 line 24318 section `cpow()`, change RATIONALE from:

2536 None.

2537 to:

2538 Permitting spurious floating-point exceptions allows `cpow(z, c)` to be implemented as `cexp(c`
2539 `clog(z))` without precluding implementations that treat special cases more carefully.

2540 Ref G.6 para 7, G.6.2.5
2541 On page 718 line 24545 section `csin()`, add:

2542 [MXC]`csin(conj(iz))`, `csinf(conjf(iz))` and `csinl(conjl(iz))` shall return exactly the same value
2543 as `conj(csin(iz))`, `conjf(csinf(iz))` and `conjl(csinl(iz))`, respectively, and `csin(-iz)`, `csinf(-iz)`
2544 and `csinl(-iz)` shall return exactly the same value as `-csin(iz)`, `-csinf(iz)` and `-csinl(iz)`,
2545 respectively, including for the special values of `iz` below.

2546 If `iz` is `+0 + i0`, `-i (+0 + i0)` shall be returned.

2547 If `iz` is `+0 + iInf`, `-i (±0 + iNaN)` shall be returned and the invalid floating-point exception
2548 shall be raised; the sign of the imaginary part of the result is unspecified.

2549 If `iz` is `+0 + iNaN`, `-i (±0 + iNaN)` shall be returned; the sign of the imaginary part of the
2550 result is unspecified.

2551 If `iz` is `x + iInf` where `x` is positive and finite, `-i (NaN + iNaN)` shall be returned and the
2552 invalid floating-point exception shall be raised.

2553 If `iz` is `x + iNaN` where `x` is non-zero and finite, `-i (NaN + iNaN)` shall be returned and the
2554 invalid floating-point exception may be raised.

2555 If `iz` is `+Inf + i0`, `-i (+Inf + i0)` shall be returned.

2556 If `iz` is `+Inf + iy` where `y` is positive and finite, `-iInf (cos(y) + isin(y))` shall be returned.

2557 If `iz` is `+Inf + iInf`, `-i (±Inf + iNaN)` shall be returned and the invalid floating-point exception
2558 shall be raised; the sign of the imaginary part of the result is unspecified.

2559 If iz is $+\text{Inf} + i\text{NaN}$, $-i (\pm\text{Inf} + i\text{NaN})$ shall be returned; the sign of the imaginary part of the
2560 result is unspecified.

2561 If iz is $\text{NaN} + i0$, $-i (\text{NaN} + i0)$ shall be returned.

2562 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
2563 invalid floating-point exception may be raised.

2564 If iz is $\text{NaN} + i\text{NaN}$, $-i (\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2565 Ref G.6 para 7
2566 On page 718 line 24553 section `csin()`, change RATIONALE from:

2567 None.

2568 to:

2569 The MXC special cases for `csin()` are derived from those for `csinh()` by applying the formula
2570 $csin(z) = -i csinh(iz)$.

2571 Ref G.6.2.5
2572 On page 719 line 24574 section `csinh()`, add:

2573 [MXC]`csinh(conj(z))`, `csinhf(conjf(z))` and `csinhl(conjl(z))` shall return exactly the same
2574 value as `conj(csinh(z))`, `conjf(csinhf(z))` and `conjl(csinhl(z))`, respectively, and `csinh(-z)`,
2575 `csinhf(-z)` and `csinhl(-z)` shall return exactly the same value as $-csinh(z)$, $-csinhf(z)$ and
2576 $-csinhl(z)$, respectively, including for the special values of z below.

2577 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2578 If z is $+0 + i\text{Inf}$, $\pm 0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2579 raised; the sign of the real part of the result is unspecified.

2580 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
2581 unspecified.

2582 If z is $x + i\text{Inf}$ where x is positive and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2583 floating-point exception shall be raised.

2584 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
2585 floating-point exception may be raised.

2586 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2587 If z is $+\text{Inf} + iy$ where y is positive and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2588 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2589 shall be raised; the sign of the real part of the result is unspecified.

2590 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
2591 unspecified.

2592 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2593 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2594 invalid floating-point exception may be raised.

2595 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2596 Ref G.6.4.2
2597 On page 721 line 24612 section `csqrt()`, add:

2598 [MXC]`csqrt(conj(z))`, `csqrtf(conjf(z))` and `csqrtl(conjl(z))` shall return exactly the same value
2599 as `conj(csqrt(z))`, `conjf(csqrtf(z))` and `conjl(csqrtl(z))`, respectively, including for the special
2600 values of z below.

2601 If z is $\pm 0 + i0$, $+0 + i0$ shall be returned.

2602 If the imaginary part of z is Inf , $+\text{Inf} + i\text{Inf}$, shall be returned.

2603 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2604 point exception may be raised.

2605 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\text{Inf}$ shall be returned.

2606 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2607 If z is $-\text{Inf} + i\text{NaN}$, $\text{NaN} \pm i\text{Inf}$ shall be returned; the sign of the imaginary part of the result
2608 is unspecified.

2609 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2610 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2611 point exception may be raised.

2612 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2613 Ref G.6 para 7, G.6.2.6
2614 On page 722 line 24641 section `ctan()`, add:

2615 [MXC]`ctan(conj(iz))`, `ctanf(conjf(iz))` and `ctanl(conjl(iz))` shall return exactly the same value
2616 as `conj(ctan(iz))`, `conjf(ctanf(iz))` and `conjl(ctanl(iz))`, respectively, and `ctan(-iz)`, `ctanf(-iz)`
2617 and `ctanl(-iz)` shall return exactly the same value as `-ctan(iz)`, `-ctanf(iz)` and `-ctanl(iz)`,
2618 respectively, including for the special values of iz below.

2619 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.

2620 If iz is $0 + i\text{Inf}$, $-i (0 + i\text{NaN})$ shall be returned and the invalid floating-point exception shall
2621 be raised.

2622 If iz is $x + i\text{Inf}$ where x is non-zero and finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
2623 invalid floating-point exception shall be raised.

2624 If iz is $0 + i\text{NaN}$, $-i (0 + i\text{NaN})$ shall be returned.

- 2625 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
 2626 invalid floating-point exception may be raised.
- 2627 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i(1 + i0 \sin(2y))$ shall be returned.
- 2628 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
 2629 unspecified.
- 2630 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
 2631 unspecified.
- 2632 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.
- 2633 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
 2634 invalid floating-point exception may be raised.
- 2635 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2636 Ref G.6 para 7
 2637 On page 722 line 24649 section `ctan()`, change RATIONALE from:
- 2638 None.
- 2639 to:
- 2640 The MXC special cases for `ctan()` are derived from those for `ctanh()` by applying the
 2641 formula $ctan(z) = -i ctanh(iz)$.
- 2642 Ref G.6.2.6
 2643 On page 723 line 24670 section `ctanh()`, add:
- 2644 [MXC]`ctanh(conj(z))`, `ctanhf(conjf(z))` and `ctanhl(conjl(z))` shall return exactly the same
 2645 value as `conj(ctanh(z))`, `conjf(ctanhf(z))` and `conjl(ctanhl(z))`, respectively, and `ctanh(-z)`,
 2646 `ctanhf(-z)` and `ctanhl(-z)` shall return exactly the same value as $-ctanh(z)$, $-ctanhf(z)$ and
 2647 $-ctanhl(z)$, respectively, including for the special values of z below.
- 2648 If z is $+0 + i0$, $+0 + i0$ shall be returned.
- 2649 If z is $0 + i\text{Inf}$, $0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
 2650 raised.
- 2651 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
 2652 floating-point exception shall be raised.
- 2653 If z is $0 + i\text{NaN}$, $0 + i\text{NaN}$ shall be returned.
- 2654 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
 2655 floating-point exception may be raised.
- 2656 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $1 + i0 \sin(2y)$ shall be returned.

2657 If z is $+Inf + iInf$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2658 unspecified.

2659 If z is $+Inf + iNaN$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2660 unspecified.

2661 If z is $NaN + i0$, $NaN + i0$ shall be returned.

2662 If z is $NaN + iy$ where y is any non-zero number, $NaN + iNaN$ shall be returned and the
2663 invalid floating-point exception may be raised.

2664 If z is $NaN + iNaN$, $NaN + iNaN$ shall be returned.[/MXC]

2665 Ref 7.27.3, 7.1.4 para 5
2666 On page 727 line 24774 section `ctime()`, change:

2667 [CX]The `ctime()` function need not be thread-safe.[/CX]

2668 to:
2669 The `ctime()` function need not be thread-safe; however, `ctime()` shall avoid data races with all
2670 functions other than itself, `asctime()`, `gmtime()` and `localtime()`.

2671 Ref 7.5 para 2
2672 On page 781 line 26447 section `errno`, change:

2673 The lvalue `errno` is used by many functions to return error values.

2674 to:
2675 The lvalue to which the macro `errno` expands is used by many functions to return error
2676 values.

2677 Ref 7.5 para 3
2678 On page 781 line 26449 section `errno`, change:

2679 The value of `errno` shall be defined only after a call to a function for which it is explicitly
2680 stated to be set and until it is changed by the next function call or if the application assigns it
2681 a value.

2682 to:
2683 The value of `errno` in the initial thread shall be zero at program startup (the initial value of
2684 `errno` in other threads is an indeterminate value) and shall otherwise be defined only after a
2685 call to a function for which it is explicitly stated to be set and until it is changed by the next
2686 function call or if the application assigns it a value.

2687 Ref 7.5 para 2
2688 On page 781 line 26456 section `errno`, delete:

2689 It is unspecified whether `errno` is a macro or an identifier declared with external linkage.

2690 Ref 7.22.4.4 para 2

2691 On page 796 line 27057 section `exit()`, add a new (unshaded) paragraph:

2692 The `exit()` function shall cause normal process termination to occur. No functions registered
2693 by the `at_quick_exit()` function shall be called. If a process calls the `exit()` function more
2694 than once, or calls the `quick_exit()` function in addition to the `exit()` function, the behavior is
2695 undefined.

2696 Ref 7.22.4.4 para 2
2697 On page 796 line 27068 section `exit()`, delete:

2698 If `exit()` is called more than once, the behavior is undefined.

2699 Ref 7.22.4.3, 7.22.4.7
2700 On page 796 line 27086 section `exit()`, add `at_quick_exit` and `quick_exit` to the SEE ALSO section.

2701 Ref F.10.4.2 para 2
2702 On page 804 line 27323 section `fabs()`, add a new paragraph:

2703 [MX]The returned value shall be exact and shall be independent of the current rounding
2704 direction mode.[/MX]

2705 Ref 7.21.2 para 7,8
2706 On page 874 line 29483 section `flockfile()`, change:

2707 These functions shall provide for explicit application-level locking of stdio (**FILE ***)
2708 objects.

2709 to:

2710 These functions shall provide for explicit application-level locking of the locks associated
2711 with standard I/O streams (see [xref to 2.5]).

2712 Ref 7.21.2 para 7,8
2713 On page 874 line 29499 section `flockfile()`, delete:

2714 All functions that reference (**FILE ***) objects, except those with names ending in `_unlocked`,
2715 shall behave as if they use `flockfile()` and `funlockfile()` internally to obtain ownership of these
2716 (**FILE ***) objects.

2717 Ref F.10.6.2 para 3
2718 On page 876 line 29560 section `floor()`, add a new paragraph:

2719 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2720 arguments.[/MX]

2721 Ref F.10.6.2 para 2
2722 On page 876 line 29562 section `floor()`, change:

2723 [MX]The result shall have the same sign as `x`.[/MX]

2724 to:

2725 [MX]The returned value shall be independent of the current rounding direction mode and
2726 shall have the same sign as x.[/MX]

2727 Ref F.10.6.2 para 3
2728 On page 876 line 29576 section floor(), delete from APPLICATION USAGE:

2729 These functions may raise the inexact floating-point exception if the result differs in value
2730 from the argument.

2731 Ref F.10.9.2 para 2
2732 On page 880 line 29695 section fmax(), add a new paragraph:

2733 [MX]The returned value shall be exact and shall be independent of the current rounding
2734 direction mode.[/MX]

2735 Ref F.10.9.3 para 2
2736 On page 884 line 29844 section fmin(), add a new paragraph:

2737 [MX]The returned value shall be exact and shall be independent of the current rounding
2738 direction mode.[/MX]

2739 Ref F.10.7.1 para 2
2740 On page 885 line 29892 section fmod(), change:

2741 [MXX]If the correct value would cause underflow, and is representable, a range error may
2742 occur and the correct value shall be returned.[/MXX]

2743 to:

2744 [MX]When subnormal results are supported, the returned value shall be exact and shall be
2745 independent of the current rounding direction mode.[/MX]

2746 [Ref 7.21.5.3 para 5](#)
2747 [On page 892 line 30117 section fopen\(\), change:](#)

2748 [\[CX\]The functionality described on this reference page is aligned with the ISO C standard.](#)
2749 [Any conflict between the requirements described here and the ISO C standard is](#)
2750 [unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.\[/CX\]](#)

2751 [to:](#)

2752 [\[CX\]Except for the “exclusive access” requirement \(see below\), the functionality described](#)
2753 [on this reference page is aligned with the ISO C standard. Any other conflict between the](#)
2754 [requirements described here and the ISO C standard is unintentional. This volume of](#)
2755 [POSIX.1-202x defers to the ISO C standard for all fopen\(\) functionality except in relation to](#)
2756 [“exclusive access”.\[/CX\]](#)

2757 [Ref 7.21.5.3 para 3](#)
2758 [On page 892 line 30125 section fopen\(\), add:](#)

2759 ~~wx or wbx~~ [Create file for writing.](#)

2760 Ref 7.21.5.3 para 3
2761 On page 892 line 30128 section `fopen()`, add:

2762 ~~`w+x` or `w+bx` or `wb+x`~~ Create file for update.

2763 Ref 7.21.5.3 para 5
2764 On page 892 line 30132 section `fopen()`, ~~add a new paragraph and list after applying bug 411,~~
2765 change:

2766 'x' If specified with a prefix beginning with 'w' [CX] or 'a' [/CX], then the function shall
2767 fail if the file already exists, [CX] as if by the `O_EXCL` flag to `open()`. If specified
2768 with a prefix beginning with 'r', this modifier shall have no effect. [/CX]

2769 to:

2770 'x' If specified with a prefix beginning with 'w' [CX] or 'a' [/CX], then the function -
2771 Opening a file with exclusive mode (`x` as the last character in the `mode`-
2772 argument) shall fail if the file already exists or cannot be created. ~~Otherwise, the file;~~
2773 if the file does not exist and can be created, it shall be created with [CX] an
2774 implementation-defined form of [/CX] exclusive (also known as non-shared) access-
2775 to the extent that, [CX] if supported by the underlying file system supports exclusive
2776 access, provided the resulting file permissions are the same as they would be without
2777 the 'x' modifier. If specified with a prefix beginning with 'r', this modifier shall have
2778 no effect. [/CX]

2779 Note: The ISO C standard requires exclusive access “to the extent that the underlying file
2780 system supports exclusive access”, but does not define what it means by this. Taken
2781 at face value—that systems must do whatever they are capable of, at the file system
2782 level, in order to exclude access by others—this would require POSIX.1 systems to
2783 set the file permissions in a way that prevents access by other users and groups.
2784 Consequently, this volume of POSIX.1-202x does not defer to the ISO C standard as
2785 regards the “exclusive access” requirement.

2786 Note to reviewers: This “exclusive access” requirement is the subject of discussions in WG14-
2787 which hopefully will result in a clarification may be clarified in C2x, in which case the above text
2788 will may be changed to match the proposed C2x text.

2789 Ref 7.21.5.3 para 3
2790 On page 892 line 30144 section `fopen()`, change:

2791 If `mode` is `w`, `wb`, `a`, `ab`, `w+`, `wb+`, `w+b`, `a+`, `ab+`, or `a+b`, and ...

2792 to:

2793 If the first character in `mode` is `w` or `a`, and ...

2794 Ref 7.21.5.3 para 3,5
2795 On page 892 line 30148 section `fopen()`, change:

2796 If `mode` is `w`, `wb`, `a`, `ab`, `w+`, `wb+`, `w+b`, `a+`, `ab+`, or `a+b`, and the file did not previously
2797 exist, the `fopen()` function shall create a file as if it called the `creat()` function with a value
2798 appropriate for the `path` argument interpreted from `pathname` and a value of `S_IRUSR` |
2799 `S_IWUSR` | `S_IRGRP` | `S_IWGRP` | `S_IROTH` | `S_IWOTH` for the `mode` argument.

2800 to:

2801 If the first character in *mode* is *w* or *a*, and the file did not previously exist, the *fopen()*
2802 function shall create a file as if it called the *open()* function with a value appropriate for the
2803 *path* argument interpreted from *pathname*, a value for the *oflag* argument as specified below,
2804 and a value of S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH for
2805 the third argument.

2806 Ref 7.21.5.3 para 5
2807 On page 893 line 30158 section *fopen()*, change:

2808 The file descriptor ...

2809 to:

2810 If the ~~last~~ first character in *mode* is *r*, or the suffix of *mode* ~~is~~ does not include *x*, the file
2811 descriptor ...

2812 Ref (none; see bug 411)

2813 On page 893 line 30160 section *fopen()*, change the first column heading from:

2814 ***fopen()* Mode**

2815 to:

2816 ***fopen()* Mode Without Suffix**

2817 and add the following text after the table:

2818 with the addition of the O_CLOEXEC flag if the suffix of *mode* includes *e*.

2819 Ref 7.21.5.3 para 5

2820 On page 893 line 30166 section *fopen()*, add the following new paragraphs:

2821 [CX]If the ~~last~~ first character in *mode* is *w* or *a*, the suffix of *mode* ~~is~~ includes *x*, and the
2822 underlying file system does not support exclusive access, then the file descriptor associated
2823 with the opened stream shall be allocated and opened as if by a call to *open()* with the
2824 following flags:

<i>fopen()</i> Mode Without Suffix	<i>open()</i> Flags
<u>[CX]<i>a</i> or <i>ab</i></u>	<u>O_WRONLY O_CREAT O_EXCL O_APPEND</u>
<u><i>a+</i> or <i>a+b</i> or <i>ab+</i></u>	<u>O_RDWR O_CREAT O_EXCL O_APPEND [CX]</u>
<u><i>w*</i> or <i>wb*</i></u>	<u>O_WRONLY O_CREAT O_EXCL O_TRUNC</u>
<u><i>w+*</i> or <i>w+b*</i> or <i>wb+*</i></u>	<u>O_RDWR O_CREAT O_EXCL O_TRUNC</u>

2825 with the addition of the O_CLOEXEC flag if the suffix of *mode* includes *e*.

2826 If the ~~last~~ first character in *mode* is *w* or *a*, the suffix of *mode* ~~is~~ includes *x*, and the
2827 underlying file system supports exclusive access, then the file descriptor associated with the

2828 opened stream shall be allocated and opened as if by a call to *open()* with the above flags or
2829 with the above flags ORed with an implementation-defined file creation flag if necessary to
2830 enable exclusive access (see above).[/CX]

2831 | [Note to reviewers: The above change may need to be updated depending on the outcome of whether](#)
2832 | [WG14 discussions about clarify the “exclusive access” requirement.](#)

2833 | [Ref 7.21.5.3 para 5](#)

2834 | [On page 893 line 30175 section *fopen\(\)*, add \(within the CX shading\):](#)

2835 | ~~[EEXIST]—The last character in *mode* is *x* and the named file exists.~~

2836 | Ref 7.21.5.3 para 5

2837 | On page 895 line 30236 section *fopen()*, change APPLICATION USAGE from:

2838 | None.

2839 | to:

2840 | If an application needs to create a file in a way that fails if the file already exists, and either
2841 | requires that it does not have exclusive access to the file or does not need exclusive access, it
2842 | should use *open()* with the O_CREAT and O_EXCL flags instead of using *fopen()* with an *x*
2843 | in the *mode*. A stream can then be created, if needed, by calling *fdopen()* on the file
2844 | descriptor returned by *open()*.

2845 | [Note to reviewers: The above change may need to be updated depending on the outcome of whether](#)
2846 | [WG14 discussions about clarify the “exclusive access” requirement.](#)

2847 | Ref 7.21.5.3 para 5

2848 | On page 895 line 30238 section *fopen()*, [after applying bug 411](#), change ~~RATIONALE~~ from:

2849 | ~~None. The *x* mode suffix character was added by C1x only for files opened with a mode~~
2850 | ~~[string beginning with *w*.](#)~~

2851 | to:

2852 | [The *x* mode suffix character is specified by the ISO C standard only for files opened with a](#)
2853 | [mode string beginning with *w*.](#)

2854 | [and then add two new paragraphs after the one that starts with the above text:](#)

2855 | When the last character in *mode* is *x*, the ISO C standard requires that the file is created with
2856 | exclusive access to the extent that the underlying system supports exclusive access.
2857 | Although POSIX.1 does not specify any method of enabling exclusive access, it allows for
2858 | the existence of an implementation-defined file creation flag that enables it. Note that it must
2859 | be a file creation flag, not a file access mode flag (that is, one that is included in
2860 | O_ACCMODE) or a file status flag, so that it does not affect the value returned by *fcntl()*
2861 | with F_GETFL. On implementations that have such a flag, if support for it is file system
2862 | dependent and exclusive access is requested when using *fopen()* to create a file on a file
2863 | system that does not support it, the flag must not be used if it would cause *fopen()* to fail.

2864 | Some implementations support mandatory file locking as a means of enabling exclusive

2865 access to a file. Locks are set in the normal way, but instead of only preventing others from
2866 setting conflicting locks they prevent others from accessing the contents of the locked part
2867 of the file in a way that conflicts with the lock. However, unless the implementation has a
2868 way of setting a whole-file write lock on file creation, this does not satisfy the requirement
2869 in the ISO C standard that the file is “created with exclusive access to the extent that the
2870 underlying system supports exclusive access”. (Having *fopen()* create the file and set a lock
2871 on the file as two separate operations is not the same, and it would introduce a race
2872 condition whereby another process could open the file and write to it (or set a lock) in
2873 between the two operations.) However, on all implementations that support mandatory file
2874 locking, its use is discouraged; therefore, it is recommended that implementations which
2875 support mandatory file locking do **not** add a means of creating a file with a whole-file
2876 exclusive lock set, so that *fopen()* is not required to enable mandatory file locking in order to
2877 conform to the ISO C standard. Note also that, since mandatory file locking is enabled via a
2878 file permissions change, the requirement that the 'x' modifier does not alter the permissions
2879 means that this standard does not allow mandatory file locking to be enabled. An
2880 implementation that has a means of creating a file with a whole-file exclusive lock set would
2881 need to provide a way to change the behavior of *fopen()* depending on whether the calling
2882 process is executing in a POSIX.1 conforming environment or an ISO C conforming
2883 environment.

2884 [Note to reviewers: The above change may need to be updated depending on ~~the outcome of whether~~](#)
2885 [WG14 discussions about clarify](#) the “exclusive access” requirement.

2886 Ref 7.22.3.3 para 2

2887 On page 933 line 31673 section *free()*, change:

2888 Otherwise, if the argument does not match a pointer earlier returned by a function in
2889 POSIX.1-2017 that allocates memory as if by *malloc()*, or if the space has been deallocated
2890 by a call to *free()* or *realloc()*, the behavior is undefined.

2891 to:

2892 Otherwise, if the argument does not match a pointer earlier returned by *aligned_alloc()*,
2893 *calloc()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] *realloc()*, or a function in POSIX.1-
2894 20xx that allocates memory as if by *malloc()*, or if the space has been deallocated by a call
2895 to *free()* or *realloc()*, the behavior is undefined.

2896 Ref 7.22.3 para 2

2897 On page 933 line 31677 section *free()*, add a new paragraph:

2898 For purposes of determining the existence of a data race, *free()* shall behave as though it
2899 accessed only memory locations accessible through its argument and not other static
2900 duration storage. The function may, however, visibly modify the storage that it deallocates.
2901 Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] and
2902 *realloc()* that allocate or deallocate a particular region of memory shall occur in a single total
2903 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
2904 next allocation (if any) in this order.

2905 Ref 7.22.3.1

2906 On page 933 line 31691 section *free()*, add *aligned_alloc* to the SEE ALSO section.

2907 [Ref 7.21.5.3 para 5](#)

2908 On page 942 line 31988 section freopen(), change:

2909 [CX]The functionality described on this reference page is aligned with the ISO C standard.
2910 Any conflict between the requirements described here and the ISO C standard is
2911 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2912 to:

2913 [CX]Except for the “exclusive access” requirement (see [xref to fopen()]), the functionality
2914 described on this reference page is aligned with the ISO C standard. Any other conflict
2915 between the requirements described here and the ISO C standard is unintentional. This
2916 volume of POSIX.1-202x defers to the ISO C standard for all *freopen()* functionality except
2917 in relation to “exclusive access”.[/CX]

2918 Ref 7.21.5.3 para 3,5; 7.21.5.4 para 2

2919 On page 942 line 32010 section freopen(), replace the following text:

2920 shall be allocated and opened as if by a call to *open()* with the following flags:

2921 and the table that follows it, and the paragraph added by bug 411 after the table, with:

2922 shall be allocated and opened as if by a call to *open()* with the flags specified for *fopen()*
2923 with the same *mode* argument.

2924 Ref (none)

2925 On page 944 line 32094 section freopen(), change:

2926 It is possible that these side-effects are an unintended consequence of the way the feature is
2927 specified in the ISO/IEC 9899: 1999 standard, but unless or until the ISO C standard is
2928 changed, ...

2929 to:

2930 It is possible that these side-effects are an unintended consequence of the way the feature
2931 was specified in the ISO/IEC 9899: 1999 standard (and still is in the current standard), but
2932 unless or until the ISO C standard is changed, ...

2933 Note to reviewers: if the APPLICATION USAGE and RATIONALE additions for fopen() are
2934 retained, changes should be added here to make the equivalent sections for freopen() refer to those
2935 for fopen().

2936 Ref (none)

2937 On page 944 line 32102 section freopen(), after applying bug 411 change:

2938 The *x* mode suffix character was added by C1x only for files opened with a *mode* string
2939 beginning with *w*.

2940 to:

2941 The *x* mode suffix character is specified by the ISO C standard only for files opened with a
2942 mode string beginning with *w*.

2943 | Ref 7.12.6.4 para 3
2944 On page 947 line 32161 section `frexp()`, change:

2945 The integer exponent shall be stored in the **int** object pointed to by *exp*.

2946 to:

2947 The integer exponent shall be stored in the **int** object pointed to by *exp*; if the integer
2948 exponent is outside the range of **int**, the results are unspecified.

2949 Ref F.10.3.4 para 3
2950 On page 947 line 32164 section `frexp()`, add a new paragraph:

2951 [MX]When the radix of the argument is a power of 2, the returned value shall be exact and
2952 shall be independent of the current rounding direction mode.[/MX]

2953 Ref 7.21.6.2 para 4
2954 On page 950 line 32239 section `fscanf()`, change:

2955 If a directive fails, as detailed below, the function shall return.

2956 to:

2957 When all directives have been executed, or if a directive fails (as detailed below), the
2958 function shall return.

2959 Ref 7.21.6.2 para 5
2960 On page 950 line 32242 section `fscanf()`, after applying bug 1163 change:

2961 A directive composed of one or more white-space bytes shall be executed by reading input
2962 until no more valid input can be read, or up to the first non-white-space byte , which remains
2963 unread.

2964 to:

2965 A directive composed of one or more white-space bytes shall be executed by reading input
2966 up to the first non-white-space byte, which shall remain unread, or until no more bytes can
2967 be read. The directive shall never fail.

2968 Ref (none)
2969 On page 955 line 32471 section `fscanf()`, change:

2970 This function is aligned with the ISO/IEC 9899: 1999 standard, and in doing so a few
2971 “obvious” things were not included. Specifically, the set of characters allowed in a scanset is
2972 limited to single-byte characters. In other similar places, multi-byte characters have been
2973 permitted, but for alignment with the ISO/IEC 9899: 1999 standard, it has not been done
2974 here.

2975 to:

2976 The set of characters allowed in a scanset is limited to single-byte characters. In other
2977 similar places, multi-byte characters have been permitted, but for alignment with the ISO C

2978 standard, it has not been done here.

2979 Ref 7.29.2.2 para 4
2980 On page 1004 line 34144 section `fwscanf()`, change:

2981 If a directive fails, as detailed below, the function shall return.

2982 to:

2983 When all directives have been executed, or if a directive fails (as detailed below), the
2984 function shall return.

2985 Ref 7.29.2.2 para 5
2986 On page 1004 line 34147 section `fwscanf()`, change:

2987 A directive composed of one or more white-space wide characters is executed by reading
2988 input until no more valid input can be read, or up to the first wide character which is not a
2989 white-space wide character, which remains unread.

2990 to:

2991 A directive composed of one or more white-space wide characters shall be executed by
2992 reading input up to the first wide character that is not a white-space wide character, which
2993 shall remain unread, or until no more wide characters can be read. The directive shall never
2994 fail.

2995 Ref 7.27.3, 7.1.4 para 5
2996 On page 1113 line 37680 section `gmtime()`, change:

2997 [CX]The `gmtime()` function need not be thread-safe.[/CX]

2998 to:
2999 The `gmtime()` function need not be thread-safe; however, `gmtime()` shall avoid data races
3000 with all functions other than itself, `asctime()`, `ctime()` and `localtime()`.

3001 Ref F.10.3.5 para 1
3002 On page 1133 line 38281 section `ilogb()`, add a new paragraph:

3003 [MX]When the correct result is representable in the range of the return type, the returned
3004 value shall be exact and shall be independent of the current rounding direction mode.[/MX]

3005 Ref F.10.3.5 para 3
3006 On page 1133 line 38282,38285,38288 section `ilogb()`, change:

3007 [XSI]On XSI-conformant systems, a domain error shall occur[/XSI]

3008 to:

3009 [XSI|MX]On XSI-conformant systems and on systems that support the IEC 60559 Floating-
3010 Point option, a domain error shall occur[/XSI|MX]

3011 Ref 7.12.6.5 para 2

3012 On page 1133 line 38291 section `ilogb()`, change:

3013 If the correct value is greater than `{INT_MAX}`, `[MX]`a domain error shall occur and`[/MX]`
3014 an unspecified value shall be returned. `[XSI]`On XSI-conformant systems, a domain error
3015 shall occur and `{INT_MAX}` shall be returned.`[/XSI]`

3016 If the correct value is less than `{INT_MIN}`, `[MX]`a domain error shall occur and`[/MX]` an
3017 unspecified value shall be returned. `[XSI]`On XSI-conformant systems, a domain error shall
3018 occur and `{INT_MIN}` shall be returned.`[/XSI]`

3019 to:

3020 If the correct value is greater than `{INT_MAX}` or less than `{INT_MIN}`, an unspecified
3021 value shall be returned. `[XSI]`On XSI-conformant systems, a domain error shall occur and
3022 `{INT_MAX}` or `{INT_MIN}`, respectively, shall be returned;`[/XSI]` `[MX]`if the IEC 60559
3023 Floating-Point option is supported, a domain error shall occur;`[/MX]` otherwise, a domain
3024 error or range error may occur.

3025 Ref F.10.3.5 para 3

3026 On page 1133 line 38300 section `ilogb()`, change:

3027 `[XSI]`The `x` argument is zero, NaN, or `±Inf.``[/XSI]`

3028 to:

3029 `[XSI|MX]`The `x` argument is zero, NaN, or `±Inf.``[/XSI|MX]`

3030 Ref F.10.11 para 1

3031 On page 1174 line 39604 section `isgreater()`,
3032 and page 1175 line 39642 section `isgreaterequal()`,
3033 and page 1177 line 39708 section `isless()`,
3034 and page 1178 line 39746 section `islessequal()`,
3035 and page 1179 line 39784 section `islessgreater()`, add a new paragraph:

3036 `[MX]`Relational operators and their corresponding comparison macros shall produce
3037 equivalent result values, even if argument values are represented in wider formats. Thus,
3038 comparison macro arguments represented in formats wider than their semantic types shall
3039 not be converted to the semantic types, unless the wide evaluation method converts operands
3040 of relational operators to their semantic types. The standard wide evaluation methods
3041 characterized by `FLT_EVAL_METHOD` equal to 1 or 2 (see `[xref to <float.h>]`) do not
3042 convert operands of relational operators to their semantic types.`[/MX]`

3043 (The editors may wish to merge the pages for the above interfaces to reduce duplication – they have
3044 duplicate APPLICATION USAGE as well.)

3045 Ref 7.30.2.2.1 para 4

3046 On page 1202 line 40411 section `iswctype()`, remove the CX shading from:

3047 If `charclass` is `(wctype_t)0`, these functions shall return 0.

3048 Ref 7.17.3.1

3049 On page 1229 line 41126 insert a new `kill_dependency()` section:

3050 **NAME**

3051 `kill_dependency` — terminate a dependency chain

3052 **SYNOPSIS**

3053 `#include <stdatomic.h>`
3054 `type kill_dependency(type y);`

3055 **DESCRIPTION**

3056 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3057 Any conflict between the requirements described here and the ISO C standard is
3058 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3059 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
3060 `<stdatomic.h>` header nor support this macro.

3061 The `kill_dependency()` macro shall terminate a dependency chain (see [xref to XBD 4.12.1
3062 Memory Ordering]). The argument shall not carry a dependency to the return value.

3063 **RETURN VALUE**

3064 The `kill_dependency()` macro shall return the value of `y`.

3065 **ERRORS**

3066 No errors are defined.

3067 **EXAMPLES**

3068 None.

3069 **APPLICATION USAGE**

3070 None.

3071 **RATIONALE**

3072 None.

3073 **FUTURE DIRECTIONS**

3074 None.

3075 **SEE ALSO**

3076 XBD Section 4.12.1, `<stdatomic.h>`

3077 **CHANGE HISTORY**

3078 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3079 Ref 7.12.8.3, 7.1.4 para 5

3080 On page 1241 line 41433 section `lgamma()`, change:

3081 [CX]These functions need not be thread-safe.[/CX]

3082 to:

3083 [XSI]If concurrent calls are made to these functions, the value of `signgam` is indeterminate.[/

3084 XSI]

3085 Ref 7.12.8.3, 7.1.4 para 5

3086 On page 1242 line 41464 section `lgamma()`, add a new paragraph to APPLICATION USAGE:

3087 If the value of *signgam* will be obtained after a call to *lgamma()*, *lgammaf()*, or *lgammal()*,
3088 in order to ensure that the value will not be altered by another call in a different thread,
3089 applications should either restrict calls to these functions to be from a single thread or use a
3090 lock such as a mutex or spin lock to protect a critical section starting before the function call
3091 and ending after the value of *signgam* has been obtained.

3092 Ref 7.12.8.3, 7.1.4 para 5

3093 On page 1242 line 41466 section `lgamma()`, change RATIONALE from:

3094 None.

3095 to:

3096 Earlier versions of this standard did not require *lgamma()*, *lgammaf()*, and *lgammal()* to be
3097 thread-safe because *signgam* was a global variable. They are now required to be thread-safe
3098 to align with the ISO C standard (which, since the introduction of threads in 2011, requires
3099 that they avoid data races), with the exception that they need not avoid data races when
3100 storing a value in the *signgam* variable. Since *signgam* is not specified by the ISO C
3101 standard, this exception is not a conflict with that standard.

3102 Ref 7.11.2.1, 7.1.4 para 5

3103 On page 1262 line 42124 section `localeconv()`, change:

3104 [CX]The *localeconv()* function need not be thread-safe.[/CX]

3105 to:

3106 The *localeconv()* function need not be thread-safe; however, *localeconv()* shall avoid data
3107 races with all other functions.

3108 Ref 7.27.3, 7.1.4 para 5

3109 On page 1265 line 42217 section `localtime()`, change:

3110 [CX]The *localtime()* function need not be thread-safe.[/CX]

3111 to:

3112 The *localtime()* function need not be thread-safe; however, *localtime()* shall avoid data races
3113 with all functions other than itself, *asctime()*, *ctime()* and *gmtime()*.

3114 Ref F.10.3.11 para 2

3115 On page 1280 line 42723 section `logb()`, add a new paragraph:

3116 [MX]The returned value shall be exact and shall be independent of the current rounding
3117 direction mode.[/MX]

3118 Ref 7.13.2.1 para 1

3119 On page 1283 line 42780 section `longjmp()`, change:

3120 `void longjmp(jmp_buf env, int val);`

3121 to:

3122 `_Noreturn void longjmp(jmp_buf env, int val);`

3123 Ref 7.13.2.1 para 2

3124 On page 1283 line 42804 section `longjmp()`, remove the CX shading from:

3125 The effect of a call to `longjmp()` where initialization of the **jmp_buf** structure was not
3126 performed in the calling thread is undefined.

3127 Ref 7.13.2.1 para 4

3128 On page 1283 line 42807 section `longjmp()`, change:

3129 After `longjmp()` is completed, program execution continues ...

3130 to:

3131 After `longjmp()` is completed, thread execution shall continue ...

3132 Ref 7.22.3 para 1

3133 On page 1295 line 43144 section `malloc()`, change:

3134 a pointer to any type of object

3135 to:

3136 a pointer to any type of object with a fundamental alignment requirement

3137 Ref 7.22.3 para 1

3138 On page 1295 line 43148 section `malloc()`, change:

3139 either a null pointer shall be returned, or ...

3140 to:

3141 either a null pointer shall be returned to indicate an error, or ...

3142 Ref 7.22.3 para 2

3143 On page 1295 line 43150 section `malloc()`, add a new paragraph:

3144 For purposes of determining the existence of a data race, `malloc()` shall behave as though it
3145 accessed only memory locations accessible through its argument and not other static
3146 duration storage. The function may, however, visibly modify the storage that it allocates.
3147 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV] and
3148 `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total
3149 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
3150 next allocation (if any) in this order.

3151 Ref 7.22.3.1

3152 On page 1295 line 43171 section `malloc()`, add *aligned_alloc* to the SEE ALSO section.

3153 Ref 7.22.7.1 para 2

3154 On page 1297 line 43194 section `mblen()`, change:

3155 `mbtowc((wchar_t *)0, s, n);`

3156 to:

3157 `mbtowc((wchar_t *)0, (const char *)0, 0);`

3158 `mbtowc((wchar_t *)0, s, n);`

3159 Ref 7.22.7 para 1

3160 On page 1297 line 43198 section `mblen()`, change:

3161 this function shall be placed into its initial state by a call for which

3162 to:

3163 this function shall be placed into its initial state at program startup and can be returned to
3164 that state by a call for which

3165 Ref 7.22.7 para 1, 7.1.4 para 5

3166 On page 1297 line 43206 section `mblen()`, change:

3167 ~~[CX]The `mblen()` function need not be thread-safe.[/CX]~~

3168 to:

3169 The `mblen()` function need not be thread-safe; however, it shall avoid data races with all
3170 other functions.

3171 Ref 7.29.6.3 para 1, 7.1.4 para 5

3172 On page 1299 line 43254 section `mbrlen()`, change:

3173 ~~[CX]The `mbrlen()` function need not be thread-safe if called with a NULL *ps*
3174 argument.[/CX]~~

3175 to:

3176 If called with a null *ps* argument, the `mbrlen()` function need not be thread-safe; however,
3177 such calls shall avoid data races with calls to `mbrlen()` with a non-null argument and with
3178 calls to all other functions.

3179 Ref 7.28.1, 7.1.4 para 5

3180 On page 1301 line 43296 insert a new `mbrtoc16()` section:

3181 **NAME**

3182 `mbrtoc16`, `mbrtoc32` — convert a character to a Unicode character code (restartable)

3183 **SYNOPSIS**

3184 `#include <uchar.h>`

```
3185     size_t mbrtoc16(char16_t *restrict pc16, const char *restrict s,  
3186                   size_t n, mbstate_t *restrict ps);  
3187     size_t mbrtoc32(char32_t *restrict pc32, const char *restrict s,  
3188                   size_t n, mbstate_t *restrict ps);
```

3189 DESCRIPTION

3190 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3191 Any conflict between the requirements described here and the ISO C standard is
3192 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3193 If *s* is a null pointer, the *mbrtoc16()* function shall be equivalent to the call:

```
3194 mbrtoc16(NULL, "", 1, ps)
```

3195 In this case, the values of the parameters *pc16* and *n* are ignored.

3196 If *s* is not a null pointer, the *mbrtoc16()* function shall inspect at most *n* bytes beginning with
3197 the byte pointed to by *s* to determine the number of bytes needed to complete the next
3198 character (including any shift sequences). If the function determines that the next character
3199 is complete and valid, it shall determine the values of the corresponding wide characters and
3200 then, if *pc16* is not a null pointer, shall store the value of the first (or only) such character in
3201 the object pointed to by *pc16*. Subsequent calls shall store successive wide characters
3202 without consuming any additional input until all the characters have been stored. If the
3203 corresponding wide character is the null wide character, the resulting state described shall be
3204 the initial conversion state.

3205 If *ps* is a null pointer, the *mbrtoc16()* function shall use its own internal **mbstate_t** object,
3206 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
3207 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current
3208 conversion state of the associated character sequence.

3209 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

3210 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

3211 The *mbrtoc32()* function shall behave the same way as *mbrtoc16()* except that the first
3212 parameter shall point to an object of type **char32_t** instead of **char16_t**. References to *pc16*
3213 in the above description shall apply as if they were *pc32* when they are being read as
3214 describing *mbrtoc32()*.

3215 If called with a null *ps* argument, the *mbrtoc16()* function need not be thread-safe; however,
3216 such calls shall avoid data races with calls to *mbrtoc16()* with a non-null argument and with
3217 calls to all other functions.

3218 If called with a null *ps* argument, the *mbrtoc32()* function need not be thread-safe; however,
3219 such calls shall avoid data races with calls to *mbrtoc32()* with a non-null argument and with
3220 calls to all other functions.

3221 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
3222 calls *mbrtoc16()* or *mbrtoc32()* with a null pointer for *ps*.

3223 RETURN VALUE

3224 These functions shall return the first of the following that applies:

3225 0 If the next *n* or fewer bytes complete the character that corresponds to the null

3226 wide character (which is the value stored).

3227 between 1 and n inclusive

3228 If the next n or fewer bytes complete a valid character (which is the value

3229 stored); the value returned shall be the number of bytes that complete the

3230 character.

3231 **(size_t)-3** If the next character resulting from a previous call has been stored, in which

3232 case no bytes from the input shall be consumed by the call.

3233 **(size_t)-2** If the next n bytes contribute to an incomplete but potentially valid character,

3234 and all n bytes have been processed (no value is stored). When n has at least

3235 the value of the {MB_CUR_MAX} macro, this case can only occur if s

3236 points at a sequence of redundant shift sequences (for implementations with

3237 state-dependent encodings).

3238 **(size_t)-1** If an encoding error occurs, in which case the next n or fewer bytes do not

3239 contribute to a complete and valid character (no value is stored). In this case,

3240 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

3241 **ERRORS**

3242 These function shall fail if:

3243 [EILSEQ] An invalid character sequence is detected. [CX]In the POSIX locale

3244 an [EILSEQ] error cannot occur since all byte values are valid

3245 characters.[/CX]

3246 These functions may fail if:

3247 [CX][EINVAL] ps points to an object that contains an invalid conversion state.[/CX]

3248 **EXAMPLES**

3249 None.

3250 **APPLICATION USAGE**

3251 None.

3252 **RATIONALE**

3253 None.

3254 **FUTURE DIRECTIONS**

3255 None.

3256 **SEE ALSO**

3257 *c16rtomb*

3258 XBD <uchar.h>

3259 **CHANGE HISTORY**

3260 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3261 Ref 7.29.6.3 para 1, 7.1.4 para 5

3262 On page 1301 line 43322 section `mbrtowc()`, change:

3263 [CX]The `mbrtowc()` function need not be thread-safe if called with a NULL *ps*
3264 argument.[/CX]

3265 to:

3266 If called with a null *ps* argument, the `mbrtowc()` function need not be thread-safe; however,
3267 such calls shall avoid data races with calls to `mbrtowc()` with a non-null argument and with
3268 calls to all other functions.

3269 Ref 7.29.6.4 para 1, 7.1.4 para 5

3270 On page 1304 line 43451 section `mbsrtowcs()`, change:

3271 [CX]The `mbsnrtowcs()` and `mbsrtowcs()` functions need not be thread-safe if called with a
3272 NULL *ps* argument.[/CX]

3273 to:

3274 [CX]If called with a null *ps* argument, the `mbsnrtowcs()` function need not be thread-safe;
3275 however, such calls shall avoid data races with calls to `mbsnrtowcs()` with a non-null
3276 argument and with calls to all other functions.[/CX]

3277 If called with a null *ps* argument, the `mbsrtowcs()` function need not be thread-safe;
3278 however, such calls shall avoid data races with calls to `mbsrtowcs()` with a non-null
3279 argument and with calls to all other functions.

3280 Ref 7.22.7 para 1

3281 On page 1308 line 43557 section `mbtowc()`, change:

3282 this function is placed into its initial state by a call for which

3283 to:

3284 this function shall be placed into its initial state at program startup and can be returned to
3285 that state by a call for which

3286 Ref 7.22.7 para 1, 7.1.4 para 5

3287 On page 1308 line 43567 section `mbtowc()`, change:

3288 [CX]The `mbtowc()` function need not be thread-safe.[/CX]

3289 to:

3290 The `mbtowc()` function need not be thread-safe; however, it shall avoid data races with all
3291 other functions.

3292 Ref 7.24.5.1 para 2

3293 On page 1311 line 43642 section `memchr()`, change:

3294 Implementations shall behave as if they read the memory byte by byte from the beginning of

3295 the bytes pointed to by *s* and stop at the first occurrence of *c* (if it is found in the initial *n*
3296 bytes).

3297 to:

3298 The implementation shall behave as if it reads the bytes sequentially and stops as soon as a
3299 matching byte is found.

3300 Ref F.10.3.12 para 2

3301 On page 1346 line 44854 section `modf()`, add a new paragraph:

3302 [MX]The returned value shall be exact and shall be independent of the current rounding
3303 direction mode.[/MX]

3304 Ref 7.26.4

3305 On page 1384 line 46032 insert the following new `mtx_*`() sections:

3306 NAME

3307 `mtx_destroy`, `mtx_init` — destroy and initialize a mutex

3308 SYNOPSIS

3309 `#include <threads.h>`

3310 `void mtx_destroy(mtx_t *mtx);`
3311 `int mtx_init(mtx_t *mtx, int type);`

3312 DESCRIPTION

3313 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3314 Any conflict between the requirements described here and the ISO C standard is
3315 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3316 The `mtx_destroy()` function shall release any resources used by the mutex pointed to by *mtx*.
3317 A destroyed mutex object can be reinitialized using `mtx_init()`; the results of otherwise
3318 referencing the object after it has been destroyed are undefined. It shall be safe to destroy an
3319 initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that
3320 another thread is attempting to lock, or a mutex that is being used in a `cond_timedwait()` or
3321 `cond_wait()` call by another thread, results in undefined behavior. The behavior is undefined if
3322 the value specified by the *mtx* argument to `mtx_destroy()` does not refer to an initialized
3323 mutex.

3324 The `mtx_init()` function shall initialize a mutex object with properties indicated by *type*,
3325 whose valid values include:

3326 `mtx_plain` for a simple non-recursive mutex,

3327 `mtx_timed` for a non-recursive mutex that supports timeout,

3328 `mtx_plain | mtx_recursive` for a simple recursive mutex, or

3329 `mtx_timed | mtx_recursive` for a recursive mutex that supports timeout.

3330 If the `mtx_init()` function succeeds, it shall set the mutex pointed to by *mtx* to a value that


```
3368     int mtx_trylock(mtx_t *mtx);
3369     int mtx_unlock(mtx_t *mtx);
```

3370 DESCRIPTION

3371 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3372 Any conflict between the requirements described here and the ISO C standard is
3373 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3374 The *mtx_lock()* function shall block until it locks the mutex pointed to by *mtx*. If the mutex
3375 is non-recursive, the application shall ensure that it is not already locked by the calling
3376 thread.

3377 The *mtx_timedlock()* function shall block until it locks the mutex pointed to by *mtx* or until
3378 after the *TIME_UTC* -based calendar time pointed to by *ts*. The application shall ensure that
3379 the specified mutex supports timeout. [CX]Under no circumstance shall the function fail
3380 with a timeout if the mutex can be locked immediately. The validity of the *ts* parameter need
3381 not be checked if the mutex can be locked immediately.[/CX]

3382 The *mtx_trylock()* function shall endeavor to lock the mutex pointed to by *mtx*. If the mutex
3383 is already locked (by any thread, including the current thread), the function shall return
3384 without blocking. If the mutex is recursive and the mutex is currently owned by the calling
3385 thread, the mutex lock count (see below) shall be incremented by one and the *mtx_trylock()*
3386 function shall immediately return success.

3387 [CX]These functions shall not be affected if the calling thread executes a signal handler
3388 during the call; if a signal is delivered to a thread waiting for a mutex, upon return from the
3389 signal handler the thread shall resume waiting for the mutex as if it was not
3390 interrupted.[/CX]

3391 If a call to *mtx_lock()*, *mtx_timedlock()* or *mtx_trylock()* locks the mutex, prior calls to
3392 *mtx_unlock()* on the same mutex shall synchronize with this lock operation.

3393 The *mtx_unlock()* function shall unlock the mutex pointed to by *mtx* . The application shall
3394 ensure that the mutex pointed to by *mtx* is locked by the calling thread. [CX]If there are
3395 threads blocked on the mutex object referenced by *mtx* when *mtx_unlock()* is called,
3396 resulting in the mutex becoming available, the scheduling policy shall determine which
3397 thread shall acquire the mutex.[/CX]

3398 A recursive mutex shall maintain the concept of a lock count. When a thread successfully
3399 acquires a mutex for the first time, the lock count shall be set to one. Every time a thread
3400 relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks
3401 the mutex, the lock count shall be decremented by one. When the lock count reaches zero,
3402 the mutex shall become available for other threads to acquire.

3403 For purposes of determining the existence of a data race, mutex lock and unlock operations
3404 on mutexes of type **mtx_t** behave as atomic operations. All lock and unlock operations on a
3405 particular mutex occur in some particular total order.

3406 If *mtx* does not refer to an initialized mutex object, the behavior of these functions is
3407 undefined.

3408 RETURN VALUE

3409 The *mtx_lock()* and *mtx_unlock()* functions shall return *thrd_success* on success, or
3410 *thrd_error* if the request could not be honored.

3411 The *mtx_timedlock()* function shall return *thrd_success* on success, or *thrd_timedout*
3412 if the time specified was reached without acquiring the requested resource, or *thrd_error*
3413 if the request could not be honored.

3414 The *mtx_trylock()* function shall return *thrd_success* on success, or *thrd_busy* if the
3415 resource requested is already in use, or *thrd_error* if the request could not be honored.
3416 The *mtx_trylock()* function can spuriously fail to lock an unused resource, in which case it
3417 shall return *thrd_busy*.

3418 **ERRORS**

3419 See RETURN VALUE.

3420 **EXAMPLES**

3421 None.

3422 **APPLICATION USAGE**

3423 None.

3424 **RATIONALE**

3425 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3426 B.2.3].

3427 Since <pthread.h> has no equivalent of the *mtx_timed* mutex property, if the <threads.h>
3428 interfaces are implemented as a thin wrapper around <pthread.h> interfaces (meaning
3429 *mtx_t* and *pthread_mutex_t* are the same type), all mutexes support timeout and
3430 *mtx_timedlock()* will not fail for a mutex that was not initialized with *mtx_timed*.
3431 Alternatively, implementations can use a less thin wrapper where *mtx_t* contains additional
3432 properties that are not held in *pthread_mutex_t* in order to be able to return a failure
3433 indication from *mtx_timedlock()* calls where the mutex was not initialized with
3434 *mtx_timed*.

3435 **FUTURE DIRECTIONS**

3436 None.

3437 **SEE ALSO**

3438 *mtx_destroy*, *timespec_get*

3439 XBD Section 4.12.2, <threads.h>

3440 **CHANGE HISTORY**

3441 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3442 Ref F.10.8.2 para 2

3443 On page 1388 line 46143 section *nan()*, add a new paragraph:

3444 [MX]The returned value shall be exact and shall be independent of the current rounding
3445 direction mode.[/MX]

3446 Ref F.10.8.3 para 2, F.10.8.4 para 2
3447 On page 1395 line 46388 section `nextafter()`, add a new paragraph:

3448 [MX]Even though underflow or overflow can occur, the returned value shall be independent
3449 of the current rounding direction mode.[/MX]

3450 Ref 7.22.3 para 2
3451 On page 1448 line 48069 section `posix_memalign()`, add a new (unshaded) paragraph:

3452 For purposes of determining the existence of a data race, `posix_memalign()` shall behave as
3453 though it accessed only memory locations accessible through its arguments and not other
3454 static duration storage. The function may, however, visibly modify the storage that it
3455 allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `posix_memalign()`, and `realloc()`
3456 that allocate or deallocate a particular region of memory shall occur in a single total order
3457 (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the next
3458 allocation (if any) in this order.

3459 Ref 7.22.3.1
3460 On page 1449 line 48107 section `posix_memalign()`, add `aligned_alloc` to the SEE ALSO section.

3461 Ref F.10.4.4 para 1
3462 On page 1548 line 50724 section `pow()`, change:

3463 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 , a pole error shall
3464 occur and `pow()`, `powf()`, and `powl()` shall return `±HUGE_VAL`, `±HUGE_VALF`, and
3465 `±HUGE_VALL`, respectively if y is an odd integer, or `HUGE_VAL`, `HUGE_VALF`, and
3466 `HUGE_VALL`, respectively if y is not an odd integer.

3467 to:

3468 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 :

- 3469 • if y is an odd integer, a pole error shall occur and `pow()`, `powf()`, and `powl()` shall
3470 return `±HUGE_VAL`, `±HUGE_VALF`, and `±HUGE_VALL`, respectively;
- 3471 • if y is finite and is not an odd integer, a pole error shall occur and `pow()`, `powf()`, and
3472 `powl()` shall return `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively;
- 3473 • if y is `-Inf`, a pole error may occur and `pow()`, `powf()`, and `powl()` shall return
3474 `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

3475 Ref 7.26
3476 On page 1603 line 52244 section `pthread_cancel()`, add a new paragraph:

3477 If *thread* refers to a thread that was created using `thrd_create()`, the behavior is undefined.

3478 Ref 7.26.5.6
3479 On page 1603 line 52277 section `pthread_cancel()`, add a new RATIONALE paragraph:

3480 Use of `pthread_cancel()` to cancel a thread that was created using `thrd_create()` is undefined
3481 because `thrd_join()` has no way to indicate a thread was cancelled. The standard developers
3482 considered adding a `thrd_cancelled` enumeration constant that `thrd_join()` would return in

3483 this case. However, this return would be unexpected in code that is written to conform to the
3484 ISO C standard, and it would also not solve the problem that threads which use only ISO C
3485 <**threads.h**> interfaces (such as ones created by third party libraries written to conform to
3486 the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not
3487 provide cancellation cleanup handlers.

3488 Ref 7.26.5.5
3489 On page 1639 line 53422 section `pthread_exit()`, change:

3490 `void pthread_exit(void *value_ptr);`

3491 to:

3492 `_Noreturn void pthread_exit(void *value_ptr);`

3493 Ref 7.26.6
3494 On page 1639 line 53427 section `pthread_exit()`, change:

3495 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3496 specific data, appropriate destructor functions shall be called in an unspecified order.

3497 to:

3498 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3499 specific data (whether associated with key type `tss_t` or `pthread_key_t`), appropriate
3500 destructor functions shall be called in an unspecified order.

3501 Ref 7.26.5.5
3502 On page 1639 line 53432 section `pthread_exit()`, change:

3503 An implicit call to `pthread_exit()` is made when a thread other than the thread in which
3504 `main()` was first invoked returns from the start routine that was used to create it.

3505 to:

3506 An implicit call to `pthread_exit()` is made when a thread that was not created using
3507 `thrd_create()`, and is not the thread in which `main()` was first invoked, returns from the start
3508 routine that was used to create it.

3509 Ref 7.26.5.5
3510 On page 1639 line 53451 section `pthread_exit()`, change APPLICATION USAGE from:

3511 None.

3512 to:

3513 Calls to `pthread_exit()` should not be made from threads created using `thrd_create()`, as their
3514 exit status has a different type (**int** instead of **void ***). If `pthread_exit()` is called from the
3515 initial thread and it is not the last thread to terminate, other threads should not try to obtain
3516 its exit status using `thrd_join()`.

3517 Ref 7.26.5.5

3518 On page 1639 line 53453 section `pthread_exit()`, change:

3519 The normal mechanism by which a thread terminates is to return from the routine that was
3520 specified in the `pthread_create()` call that started it.

3521 to:

3522 The normal mechanism by which a thread that was started using `pthread_create()` terminates
3523 is to return from the routine that was specified in the `pthread_create()` call that started it.

3524 Ref 7.26.5.5, 7.26.6

3525 On page 1640 line 53470 section `pthread_exit()`, add `pthread_key_create`, `thrd_create`, `thrd_exit` and
3526 `tss_create` to the SEE ALSO section.

3527 Ref 7.26.5.5

3528 On page 1649 line 53748 section `pthread_join()`, add a new paragraph:

3529 If *thread* refers to a thread that was created using `thrd_create()` and the thread terminates, or
3530 has already terminated, by returning from its start routine, the behavior of `pthread_join()` is
3531 undefined. If *thread* refers to a thread that terminates, or has already terminated, by calling
3532 `thrd_exit()`, the behavior of `pthread_join()` is undefined.

3533 Ref 7.26.5.5

3534 On page 1651 line 53819 section `pthread_join()`, add a new RATIONALE paragraph:

3535 The `pthread_join()` function cannot be used to obtain the exit status of a thread that was
3536 created using `thrd_create()` and which terminates by returning from its start routine, or of a
3537 thread that terminates by calling `thrd_exit()`, because such threads have an **int** exit status,
3538 instead of the **void *** that `pthread_join()` returns via its `value_ptr` argument.

3539 Ref 7.22.4.7

3540 On page 1765 line 57040 insert the following new `quick_exit()` section:

3541 **NAME**

3542 `quick_exit` — terminate a process

3543 **SYNOPSIS**

3544 `#include <stdlib.h>`

3545 `_Noreturn void quick_exit(int status);`

3546 **DESCRIPTION**

3547 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3548 Any conflict between the requirements described here and the ISO C standard is
3549 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3550 The `quick_exit()` function shall cause normal process termination to occur. It shall not call
3551 functions registered with `atexit()` nor any registered signal handlers. If a process calls the
3552 `quick_exit()` function more than once, or calls the `exit()` function in addition to the
3553 `quick_exit()` function, the behavior is undefined. If a signal is raised while the `quick_exit()`
3554 function is executing, the behavior is undefined.

3555 The *quick_exit()* function shall first call all functions registered by *at_quick_exit()*, in the
3556 reverse order of their registration, except that a function is called after any previously
3557 registered functions that had already been called at the time it was registered. If, during the
3558 call to any such function, a call to the *longjmp()* [CX] or *siglongjmp()*[/CX] function is made
3559 that would terminate the call to the registered function, the behavior is undefined.

3560 If a function registered by a call to *at_quick_exit()* fails to return, the remaining registered
3561 functions shall not be called and the rest of the *quick_exit()* processing shall not be
3562 completed.

3563 Finally, the *quick_exit()* function shall terminate the process as if by a call to *_Exit(status)*.

3564 **RETURN VALUE**

3565 The *quick_exit()* function does not return.

3566 **ERRORS**

3567 No errors are defined.

3568 **EXAMPLES**

3569 None.

3570 **APPLICATION USAGE**

3571 None.

3572 **RATIONALE**

3573 None.

3574 **FUTURE DIRECTIONS**

3575 None.

3576 **SEE ALSO**

3577 *_Exit*, *at_quick_exit*, *atexit*, *exit*

3578 XBD <stdlib.h>

3579 **CHANGE HISTORY**

3580 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3581 Ref 7.22.2.1 para 3, 7.1.4 para 5

3582 On page 1767 line 57095 section *rand()*, change:

3583 [CX]The *rand()* function need not be thread-safe.[/CX]

3584 to:

3585 The *rand()* function need not be thread-safe; however, *rand()* shall avoid data races with all
3586 functions other than non-thread-safe pseudo-random sequence generation functions.

3587 Ref 7.22.2.2 para 3, 7.1.4 para 5

3588 On page 1767 line 57105 section *rand()*, add a new paragraph:

3589 The *srand()* function need not be thread-safe; however, *srand()* shall avoid data races with
3590 all functions other than non-thread-safe pseudo-random sequence generation functions.

3591 Ref 7.22.3 para 1,2; 7.22.3.5 para 2,3,4; 7.31.12 para 2
3592 On page 1788 line 57862-57892 section *realloc()*, replace the DESCRIPTION and RETURN
3593 VALUE sections with:

3594 **DESCRIPTION**

3595 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3596 Any conflict between the requirements described here and the ISO C standard is
3597 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3598 The *realloc()* function shall deallocate the old object pointed to by *ptr* and return a pointer to
3599 a new object that has the size specified by *size*. The contents of the new object shall be the
3600 same as that of the old object prior to deallocation, up to the lesser of the new and old sizes.
3601 Any bytes in the new object beyond the size of the old object have indeterminate values.

3602 If *ptr* is a null pointer, *realloc()* shall be equivalent to *malloc()* function for the specified
3603 size. Otherwise, if *ptr* does not match a pointer returned earlier by *aligned_alloc()*, *calloc()*,
3604 *malloc()*, [ADV]*posix_memalign()*,[/ADV] *realloc()*, or a function in POSIX.1-20xx that
3605 allocates memory as if by *malloc()*, or if the space has been deallocated by a call to *free()* or
3606 *realloc()*, the behavior is undefined.

3607 If *size* is non-zero and memory for the new object is not allocated, the old object shall not be
3608 deallocated. [OB]If *size* is zero and memory for the new object is not allocated, it is
3609 implementation-defined whether the old object is deallocated; if the old object is not
3610 deallocated, its value shall be unchanged.[/OB]

3611 The order and contiguity of storage allocated by successive calls to *realloc()* is unspecified.
3612 The pointer returned if the allocation succeeds shall be suitably aligned so that it may be
3613 assigned to a pointer to any type of object with a fundamental alignment requirement and
3614 then used to access such an object in the space allocated (until the space is explicitly freed or
3615 reallocated). Each such allocation shall yield a pointer to an object disjoint from any other
3616 object. The pointer returned shall point to the start (lowest byte address) of the allocated
3617 space. If the space cannot be allocated, a null pointer shall be returned. [OB]If the size of the
3618 space requested is 0, the behavior is implementation-defined: either a null pointer shall be
3619 returned to indicate an error, or the behavior shall be as if the size were some non-zero
3620 value, except that the behavior is undefined if the returned pointer is used to access an
3621 object.[/OB]

3622 For purposes of determining the existence of a data race, *realloc()* shall behave as though it
3623 accessed only memory locations accessible through its arguments and not other static
3624 duration storage. The function may, however, visibly modify the storage that it allocates or
3625 deallocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*,
3626 [ADV]*posix_memalign()*,[/ADV] and *realloc()* that allocate or deallocate a particular region
3627 of memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
3628 deallocation call shall synchronize with the next allocation (if any) in this order.

3629 **RETURN VALUE**

3630 The *realloc()* function shall return a pointer to the new object (which can have the same
3631 value as a pointer to the old object), or a null pointer if the new object has not been
3632 allocated.

3633 [OB]If size is zero, either:

- 3634 • A null pointer shall be returned [CX]and, if *ptr* is not a null pointer, *errno* shall be set
- 3635 to an implementation-defined value.[/CX]
- 3636 • A pointer to the allocated space shall be returned, and the memory object pointed to
- 3637 by *ptr* shall be freed. The application shall ensure that the pointer is not used to
- 3638 access an object.[/OB]

3639 If there is not enough available memory, *realloc()* shall return a null pointer [CX]and set

3640 *errno* to [ENOMEM][/CX].

3641 Ref 7.22.3.5 para 3,4

3642 On page 1789 line 57899 section *realloc()*, change:

3643 The description of *realloc()* has been modified from previous versions of this standard to

3644 align with the ISO/IEC 9899: 1999 standard. Previous versions explicitly permitted a call to

3645 *realloc(p, 0)* to free the space pointed to by *p* and return a null pointer. While this behavior

3646 could be interpreted as permitted by this version of the standard, the C language committee

3647 have indicated that this interpretation is incorrect. Applications should assume that if

3648 *realloc()* returns a null pointer, the space pointed to by *p* has not been freed. Since this could

3649 lead to double-frees, implementations should also set *errno* if a null pointer actually

3650 indicates a failure, and applications should only free the space if *errno* was changed.

3651 to:

3652 The ISO C standard makes it implementation-defined whether a call to *realloc(p, 0)* frees the

3653 space pointed to by *p* if it returns a null pointer because memory for the new object was not

3654 allocated. POSIX.1 instead requires that implementations set *errno* if a null pointer is

3655 returned and the space has not been freed, and POSIX applications should only free the

3656 space if *errno* was changed.

3657 Ref 7.31.12 para 2

3658 On page 1789 line 57909-57912 section *realloc()*, change FUTURE DIRECTIONS to:

3659 The ISO C standard states that invoking *realloc()* with a *size* argument equal to zero is an

3660 obsolescent feature. This feature may be removed in a future version of this standard.

3661 Ref 7.22.3.1

3662 On page 1789 line 57914 section *realloc()*, add *aligned_alloc* to the SEE ALSO section.

3663 Ref F.10.7.2 para 2

3664 On page 1809 line 58638 section *remainder()*, add a new paragraph:

3665 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3666 Ref F.10.7.3 para 2

3667 On page 1814 line 58758 section *remquo()*, add a new paragraph:

3668 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3669 Ref F.10.6.6 para 3

3670 On page 1828 line 59258 section `round()`, add a new paragraph:

3671 [MX]These functions may raise the inexact floating-point exception for finite non-integer
3672 arguments.[/MX]

3673 Ref F.10.6.6 para 3
3674 On page 1828 line 59272 section `round()`, delete from APPLICATION USAGE:

3675 These functions may raise the inexact floating-point exception if the result differs in value
3676 from the argument.

3677 Ref F.10.3.13 para 2
3678 On page 1829 line 59306 section `scalbn()`, add a new paragraph:

3679 [MX]If the calculation does not overflow or underflow, the returned value shall be exact and
3680 shall be independent of the current rounding direction mode.[/MX]

3681 Ref 7.11.1.1 para 5
3682 On page 1903 line 61520 section `setlocale()`, ~~remove the CX shading from~~change:

3683 [CX]The `setlocale()` function need not be thread-safe.[/CX]

3684 to:

3685 The `setlocale()` function need not be thread-safe; however, it shall avoid data races with all
3686 function calls that do not affect and are not affected by the global locale.

3687 Ref 7.13.2.1 para 1
3688 On page 1970 line 63497 section `siglongjmp()`, change:

3689 `void siglongjmp(sigjmp_buf env, int val);`

3690 to:

3691 `_Noreturn void siglongjmp(sigjmp_buf env, int val);`

3692 Ref 7.13.2.1 para 4
3693 On page 1970 line 63504 section `siglongjmp()`, change:

3694 After `siglongjmp()` is completed, program execution shall continue ...

3695 to:

3696 After `siglongjmp()` is completed, thread execution shall continue ...

3697 Ref 7.14.1.1 para 5
3698 On page 1971 line 63564 section `signal()`, change:

3699 with static storage duration

3700 to:

3701 with static or thread storage duration that is not a lock-free atomic object

3702 [Ref 7.14.1.1 para 7](#)

3703 [On page 1972 line 63573 section `signal\(\)`, add a new paragraph:](#)

3704 [\[CX\]The `signal\(\)` function is required to be thread-safe. \(See \[\\[xref to 2.9.1 Thread-Safety\\]\]\(#\).\)](#)
3705 [\[/CX\]](#)

3706 [Ref 7.14.1.1 para 7](#)

3707 [On page 1972 line 63591 section `signal\(\)`, change RATIONALE from:](#)

3708 [None.](#)

3709 [to:](#)

3710 [The ISO C standard says that the use of `signal\(\)` in a multi-threaded program results in](#)
3711 [undefined behavior. However, POSIX.1 has required `signal\(\)` to be thread-safe since before](#)
3712 [threads were added to the ISO C standard.](#)

3713 Ref F.10.4.5 para 1

3714 On page 2009 line 64624 section `sqrt()`, add:

3715 [\[MX\]The returned value shall be dependent on the current rounding direction mode.\[/MX\]](#)

3716 Ref 7.24.6.2 para 3, 7.1.4 para 5

3717 On page 2035 line 65231 section `strerror()`, change:

3718 [\[CX\]The `strerror\(\)` function need not be thread-safe.\[/CX\]](#)

3719 to:

3720 The `strerror()` function need not be thread-safe; however, `strerror()` shall avoid data races
3721 with all other functions.

3722 Ref 7.22.1.3 para 10

3723 On page 2073 line 66514 section `strtod()`, change:

3724 If the correct value is outside the range of representable values

3725 to:

3726 If the correct value would cause an overflow and default rounding is in effect

3727 Ref 7.24.5.8 para 6, 7.1.4 para 5

3728 On page 2078 line 66674 section `strtok()`, change:

3729 [\[CX\]The `strtok\(\)` function need not be thread-safe.\[/CX\]](#)

3730 to:

3731 The `strtok()` function need not be thread-safe; however, `strtok()` shall avoid data races with
3732 all other functions.

3733 Ref 7.22.4.8, 7.1.4 para 5

3734 On page 2107 line 67579 section `system()`, change:

3735 The `system()` function need not be thread-safe.

3736 to:

3737 [~~CX~~]If concurrent calls to `system()` are made from multiple threads, it is unspecified
3738 whether:

- 3739 • each call saves and restores the dispositions of the SIGINT and SIGQUIT signals
3740 independently, or
- 3741 • in a set of concurrent calls the dispositions in effect after the last call returns are
3742 those that were in effect on entry to the first call.

3743 If a thread is cancelled while it is in a call to `system()`, it is unspecified whether the child
3744 process is terminated and waited for, or is left running.[~~CX~~]

3745 Ref 7.22.4.8, 7.1.4 para 5

3746 On page 2108 line 67627 section `system()`, change:

3747 Using the `system()` function in more than one thread in a process or when the SIGCHLD
3748 signal is being manipulated by more than one thread in a process may produce unexpected
3749 results.

3750 to:

3751 Although `system()` is required to be thread-safe, it is recommended that concurrent calls
3752 from multiple threads are avoided, since `system()` is not required to coordinate the saving
3753 and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of
3754 overlapping calls, and therefore the signals might end up being set to ignored after the last
3755 call returns. Applications should also avoid cancelling a thread while it is in a call to
3756 `system()` as the child process may be left running in that event. In addition, if another thread
3757 alters the disposition of the SIGCHLD signal, a call to `signal()` may produce unexpected
3758 results.

3759 Ref 7.22.4.8, 7.1.4 para 5

3760 On page 2109 line 67675 section `system()`, delete:

3761 `#include <signal.h>`

3762 Ref 7.22.4.8, 7.1.4 para 5

3763 On page 2109 line 67692,67696,67712 section `system()`, change `sigprocmask` to
3764 `pthread_sigmask`.

3765 Ref 7.22.4.8, 7.1.4 para 5

3766 On page 2110 line 67718 section `system()`, change:

3767 Note also that the above example implementation is not thread-safe. Implementations can
3768 provide a thread-safe `system()` function, but doing so involves complications such as how to
3769 restore the signal dispositions for SIGINT and SIGQUIT correctly if there are overlapping
3770 calls, and how to deal with cancellation. The example above would not restore the signal
3771 dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-

3772 safe implementation since canceling a non-thread-safe function results in undefined
3773 behavior (see Section 2.9.5.2, on page 518). To avoid leaking a process ID, a thread-safe
3774 implementation would need to terminate the child process when acting on a cancellation.

3775 to:

3776 Earlier versions of this standard did not require *system()* to be thread-safe because it alters
3777 the process-wide disposition of the SIGINT and SIGQUIT signals. It is now required to be
3778 thread-safe to align with the ISO C standard, which (since the introduction of threads in
3779 2011) requires that it avoids data races. However, the function is not required to coordinate
3780 the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set
3781 of overlapping calls, and the above example does not do so. The example also does not
3782 terminate and wait for the child process if the calling thread is cancelled, and so would leak
3783 a process ID in that event.

3784 Ref 7.26.5

3785 On page 2148 line 68796 insert the following new *thrd_**() sections:

3786 **NAME**

3787 *thrd_create* — thread creation

3788 **SYNOPSIS**

3789 `#include <threads.h>`

3790 `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`

3791 **DESCRIPTION**

3792 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3793 Any conflict between the requirements described here and the ISO C standard is
3794 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3795 The *thrd_create()* function shall create a new thread executing *func(arg)*. If the *thrd_create()*
3796 function succeeds, it shall set the object pointed to by *thr* to the identifier of the newly
3797 created thread. (A thread's identifier might be reused for a different thread once the original
3798 thread has exited and either been detached or joined to another thread.) The completion of
3799 the *thrd_create()* function shall synchronize with the beginning of the execution of the new
3800 thread.

3801 [CX]The signal state of the new thread shall be initialized as follows:

- 3802 • The signal mask shall be inherited from the creating thread.
- 3803 • The set of signals pending for the new thread shall be empty.

3804 The thread-local current locale shall not be inherited from the creating thread.

3805 The floating-point environment shall be inherited from the creating thread.[/CX]

3806 [XSI] The alternate stack shall not be inherited from the creating thread.[/XSI]

3807 Returning from *func* shall have the same behavior as invoking *thrd_exit()* with the value
3808 returned from *func*.

3809 If *thrd_create()* fails, no new thread shall be created and the contents of the location
3810 referenced by *thr* are undefined.

3811 [CX]The *thrd_create()* function shall not be affected if the calling thread executes a signal
3812 handler during the call.[/CX]

3813 RETURN VALUE

3814 The *thrd_create()* function shall return *thrd_success* on success; or *thrd_nomem* if no
3815 memory could be allocated for the thread requested; or *thrd_error* if the request could not
3816 be honored, [CX]such as if the system-imposed limit on the total number of threads in a
3817 process {*PTHREAD_THREADS_MAX*} would be exceeded.[/CX]

3818 ERRORS

3819 See RETURN VALUE.

3820 EXAMPLES

3821 None.

3822 APPLICATION USAGE

3823 There is no requirement on the implementation that the ID of the created thread be available
3824 before the newly created thread starts executing. The calling thread can obtain the ID of the
3825 created thread through the *thr* argument of the *thrd_create()* function, and the newly created
3826 thread can obtain its ID by a call to *thrd_current()*.

3827 RATIONALE

3828 The *thrd_create()* function is not affected by signal handlers for the reasons stated in [xref to
3829 XRAT B.2.3].

3830 FUTURE DIRECTIONS

3831 None.

3832 SEE ALSO

3833 *pthread_create*, *thrd_current*, *thrd_detach*, *thrd_exit*, *thrd_join*

3834 XBD Section 4.12.2, <**threads.h**>

3835 CHANGE HISTORY

3836 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3837 NAME

3838 *thrd_current* — get the calling thread ID

3839 SYNOPSIS

3840 `#include <threads.h>`

3841 `thrd_t thrd_current(void);`

3842 DESCRIPTION

3843 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3844 Any conflict between the requirements described here and the ISO C standard is
3845 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3846 The *thrd_current()* function shall identify the thread that called it.

3847 **RETURN VALUE**

3848 The *thrd_current()* function shall return the thread ID of the thread that called it.

3849 The *thrd_current()* function shall always be successful. No return value is reserved to
3850 indicate an error.

3851 **ERRORS**

3852 No errors are defined.

3853 **EXAMPLES**

3854 None.

3855 **APPLICATION USAGE**

3856 None.

3857 **RATIONALE**

3858 None.

3859 **FUTURE DIRECTIONS**

3860 None.

3861 **SEE ALSO**

3862 *pthread_self*, *thrd_create*, *thrd_equal*

3863 XBD Section 4.12.2, <**threads.h**>

3864 **CHANGE HISTORY**

3865 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3866 **NAME**

3867 *thrd_detach* — detach a thread

3868 **SYNOPSIS**

3869 `#include <threads.h>`

3870 `int thrd_detach(thrd_t thr);`

3871 **DESCRIPTION**

3872 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3873 Any conflict between the requirements described here and the ISO C standard is
3874 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3875 The *thrd_detach()* function shall change the thread *thr* from joinable to detached, indicating
3876 to the implementation that any resources allocated to the thread can be reclaimed when that
3877 thread terminates. The application shall ensure that the thread identified by *thr* has not been
3878 previously detached or joined with another thread.

3879 [CX]The *thrd_detach()* function shall not be affected if the calling thread executes a signal
3880 handler during the call.[/CX]

3881 **RETURN VALUE**

3882 The *thrd_detach()* function shall return *thrd_success* on success or *thrd_error* if the
3883 request could not be honored.

3884 **ERRORS**

3885 No errors are defined.

3886 **EXAMPLES**

3887 None.

3888 **APPLICATION USAGE**

3889 None.

3890 **RATIONALE**

3891 The *thrd_detach()* function is not affected by signal handlers for the reasons stated in [xref
3892 to XRAT B.2.3].

3893 **FUTURE DIRECTIONS**

3894 None.

3895 **SEE ALSO**

3896 *pthread_detach*, *thrd_create*, *thrd_join*

3897 XBD <**threads.h**>

3898 **CHANGE HISTORY**

3899 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3900 **NAME**

3901 *thrd_equal* — compare thread IDs

3902 **SYNOPSIS**

3903 `#include <threads.h>`

3904 `int thrd_equal(thrd_t thr0, thrd_t thr1);`

3905 **DESCRIPTION**

3906 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3907 Any conflict between the requirements described here and the ISO C standard is
3908 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3909 The *thrd_equal()* function shall determine whether the thread identified by *thr0* refers to the
3910 thread identified by *thr1*.

3911 [CX]The *thrd_equal()* function shall not be affected if the calling thread executes a signal
3912 handler during the call.[/CX]

3913 **RETURN VALUE**

3914 The *thrd_equal()* function shall return a non-zero value if *thr0* and *thr1* are equal; otherwise,
3915 zero shall be returned.

3916 If either *thr0* or *thr1* is not a valid thread ID [CX]and is not equal to PTHREAD_NULL
3917 (which is defined in <pthread.h>[/CX], the behavior is undefined.

3918 **ERRORS**

3919 No errors are defined.

3920 **EXAMPLES**

3921 None.

3922 **APPLICATION USAGE**

3923 None.

3924 **RATIONALE**

3925 See the RATIONALE section for *pthread_equal()*.

3926 The *thrd_equal()* function is not affected by signal handlers for the reasons stated in [xref to
3927 XRAT B.2.3].

3928 **FUTURE DIRECTIONS**

3929 None.

3930 **SEE ALSO**

3931 *pthread_equal*, *thrd_current*

3932 XBD <pthread.h>, <threads.h>

3933 **CHANGE HISTORY**

3934 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3935 **NAME**

3936 *thrd_exit* — thread termination

3937 **SYNOPSIS**

3938 `#include <threads.h>`

3939 `_Noreturn void thrd_exit(int res);`

3940 **DESCRIPTION**

3941 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3942 Any conflict between the requirements described here and the ISO C standard is
3943 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3944 For every thread-specific storage key [CX](regardless of whether it has type **tss_t** or
3945 **pthread_key_t**[/CX] which was created with a non-null destructor and for which the value
3946 is non-null, *thrd_exit()* shall set the value associated with the key to a null pointer value and
3947 then invoke the destructor with its previous value. The order in which destructors are
3948 invoked is unspecified.

3949 If after this process there remain keys with both non-null destructors and values, the
3950 implementation shall repeat this process up to [CX]
3951 {PTHREAD_DESTRUCTOR_ITERATIONS}[/CX] times.

3952 Following this, the *thrd_exit()* function shall terminate execution of the calling thread and
3953 shall set its exit status to *res*. [CX]Thread termination shall not release any application
3954 visible process resources, including, but not limited to, mutexes and file descriptors, nor
3955 shall it perform any process-level cleanup actions, including, but not limited to, calling any
3956 *atexit()* routines that might exist.[/CX]

3957 An implicit call to *thrd_exit()* is made when a thread that was created using *thrd_create()*
3958 returns from the start routine that was used to create it (see [xref to *thrd_create()*]).

3959 [CX]The behavior of *thrd_exit()* is undefined if called from a destructor function that was
3960 invoked as a result of either an implicit or explicit call to *thrd_exit()*.[/CX]

3961 The process shall exit with an exit status of zero after the last thread has been terminated.
3962 The behavior shall be as if the implementation called *exit()* with a zero argument at thread
3963 termination time.

3964 **RETURN VALUE**

3965 This function shall not return a value.

3966 **ERRORS**

3967 No errors are defined.

3968 **EXAMPLES**

3969 None.

3970 **APPLICATION USAGE**

3971 Calls to *thrd_exit()* should not be made from threads created using *pthread_create()* or via a
3972 SIGEV_THREAD notification, as their exit status has a different type (**void *** instead of
3973 **int**). If *thrd_exit()* is called from the initial thread and it is not the last thread to terminate,
3974 other threads should not try to obtain its exit status using *pthread_join()*.

3975 **RATIONALE**

3976 The normal mechanism by which a thread that was started using *thrd_create()* terminates is
3977 to return from the function that was specified in the *thrd_create()* call that started it. The
3978 *thrd_exit()* function provides the capability for such a thread to terminate without requiring a
3979 return from the start routine of that thread, thereby providing a function analogous to *exit()*.

3980 Regardless of the method of thread termination, the destructors for any existing thread-
3981 specific data are executed.

3982 **FUTURE DIRECTIONS**

3983 None.

3984 **SEE ALSO**

3985 *exit*, *pthread_create*, *thrd_join*

3986 XBD <**threads.h**>

3987 **CHANGE HISTORY**

3988 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3989 **NAME**

3990 `thrd_join` — wait for thread termination

3991 **SYNOPSIS**

3992 `#include <threads.h>`

3993 `int thrd_join(thrd_t thr, int *res);`

3994 **DESCRIPTION**

3995 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3996 Any conflict between the requirements described here and the ISO C standard is
3997 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3998 The `thrd_join()` function shall join the thread identified by *thr* with the current thread by
3999 blocking until the other thread has terminated. If the parameter *res* is not a null pointer,
4000 `thrd_join()` shall store the thread's exit status in the integer pointed to by *res*. The
4001 termination of the other thread shall synchronize with the completion of the `thrd_join()`
4002 function. The application shall ensure that the thread identified by *thr* has not been
4003 previously detached or joined with another thread.

4004 The results of multiple simultaneous calls to `thrd_join()` specifying the same target thread
4005 are undefined.

4006 The behavior is undefined if the value specified by the *thr* argument to `thrd_join()` refers to
4007 the calling thread.

4008 [CX]It is unspecified whether a thread that has exited but remains unjoined counts against
4009 {PTHREAD_THREADS_MAX}.

4010 If *thr* refers to a thread that was created using `pthread_create()` or via a SIGEV_THREAD
4011 notification and the thread terminates, or has already terminated, by returning from its start
4012 routine, the behavior of `thrd_join()` is undefined. If *thr* refers to a thread that terminates, or
4013 has already terminated, by calling `pthread_exit()` or by being cancelled, the behavior of
4014 `thrd_join()` is undefined.

4015 The `thrd_join()` function shall not be affected if the calling thread executes a signal handler
4016 during the call.[/CX]

4017 **RETURN VALUE**

4018 The `thrd_join()` function shall return `thrd_success` on success or `thrd_error` if the
4019 request could not be honored.

4020 [CX]It is implementation-defined whether `thrd_join()` detects deadlock situations; if it does
4021 detect them, it shall return `thrd_error` when one is detected.[/CX]

4022 **ERRORS**

4023 See RETURN VALUE.

4024 **EXAMPLES**

4025 None.

4026 **APPLICATION USAGE**

4027 None.

4028 **RATIONALE**

4029 The *thrd_join()* function provides a simple mechanism allowing an application to wait for a
4030 thread to terminate. After the thread terminates, the application may then choose to clean up
4031 resources that were used by the thread. For instance, after *thrd_join()* returns, any
4032 application-provided stack storage could be reclaimed.

4033 The *thrd_join()* or *thrd_detach()* function should eventually be called for every thread that is
4034 created using *thrd_create()* so that storage associated with the thread may be reclaimed.

4035 The *thrd_join()* function cannot be used to obtain the exit status of a thread that was created
4036 using *pthread_create()* or via a SIGEV_THREAD notification and which terminates by
4037 returning from its start routine, or of a thread that terminates by calling *pthread_exit()*,
4038 because such threads have a **void *** exit status, instead of the **int** that *thrd_join()* returns via
4039 its *res* argument.

4040 The *thrd_join()* function cannot be used to obtain the exit status of a thread that terminates
4041 by being cancelled because it has no way to indicate that a thread was cancelled. (The
4042 *pthread_join()* function does this by returning a reserved **void *** exit status; it is not possible
4043 to reserve an **int** value for this purpose without introducing a conflict with the ISO C
4044 standard.) The standard developers considered adding a *thrd_cancelled* enumeration
4045 constant that *thrd_join()* would return in this case. However, this return would be
4046 unexpected in code that is written to conform to the ISO C standard, and it would also not
4047 solve the problem that threads which use only ISO C **<threads.h>** interfaces (such as ones
4048 created by third party libraries written to conform to the ISO C standard) have no way to
4049 handle being cancelled, as the ISO C standard does not provide cancellation cleanup
4050 handlers.

4051 The *thrd_join()* function is not affected by signal handlers for the reasons stated in [xref to
4052 XRAT B.2.3].

4053 **FUTURE DIRECTIONS**

4054 None.

4055 **SEE ALSO**

4056 *pthread_create*, *pthread_exit*, *pthread_join*, *thrd_create*, *thrd_exit*

4057 XBD Section 4.12.2, **<threads.h>**

4058 **CHANGE HISTORY**

4059 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4060 **NAME**

4061 *thrd_sleep* — suspend execution for an interval

4062 **SYNOPSIS**

4063 `#include <threads.h>`

4064 `int thrd_sleep(const struct timespec *duration,`
4065 `struct timespec *remaining);`

4066 **DESCRIPTION**

4067 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4068 Any conflict between the requirements described here and the ISO C standard is
4069 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4070 The *thrd_sleep()* function shall suspend execution of the calling thread until either the
4071 interval specified by *duration* has elapsed or a signal is delivered to the calling thread whose
4072 action is to invoke a signal-catching function or to terminate the process. If interrupted by a
4073 signal and the *remaining* argument is not null, the amount of time remaining (the requested
4074 interval minus the time actually slept) shall be stored in the interval it points to. The
4075 *duration* and *remaining* arguments can point to the same object.

4076 The suspension time may be longer than requested because the interval is rounded up to an
4077 integer multiple of the sleep resolution or because of the scheduling of other activity by the
4078 system. But, except for the case of being interrupted by a signal, the suspension time shall
4079 not be less than that specified, as measured by the system clock TIME_UTC.

4080 RETURN VALUE

4081 The *thrd_sleep()* function shall return zero if the requested time has elapsed, -1 if it has
4082 been interrupted by a signal, or a negative value (which may also be -1) if it fails for any
4083 other reason. [CX]If it returns a negative value, it shall set *errno* to indicate the error.[/CX]

4084 ERRORS

4085 [CX]The *thrd_sleep()* function shall fail if:

4086 [EINTR]

4087 The *thrd_sleep()* function was interrupted by a signal.

4088 [EINVAL]

4089 The *duration* argument specified a nanosecond value less than zero or greater than or
4090 equal to 1000 million.[/CX]

4091 EXAMPLES

4092 None.

4093 APPLICATION USAGE

4094 Since the return value may be -1 for errors other than [EINTR], applications should examine
4095 *errno* to distinguish [EINTR] from other errors (and thus determine whether the unslept time
4096 is available in the interval pointed to by *remaining*).

4097 RATIONALE

4098 The *thrd_sleep()* function is identical to the *nanosleep()* function except that the return value
4099 may be any negative value when it fails with an error other than [EINTR].

4100 FUTURE DIRECTIONS

4101 None.

4102 SEE ALSO

4103 *nanosleep*

4104 XBD <threads.h>, <time.h>

4105 CHANGE HISTORY

4106 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4107 **NAME**
4108 `thrd_yield` — yield the processor

4109 **SYNOPSIS**
4110 `#include <threads.h>`
4111 `void thrd_yield(void);`

4112 **DESCRIPTION**
4113 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4114 Any conflict between the requirements described here and the ISO C standard is
4115 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4116 [CX]The `thrd_yield()` function shall force the running thread to relinquish the processor until
4117 it again becomes the head of its thread list.[/CX]

4118 **RETURN VALUE**
4119 This function shall not return a value.

4120 **ERRORS**
4121 No errors are defined.

4122 **EXAMPLES**
4123 None.

4124 **APPLICATION USAGE**
4125 See the APPLICATION USAGE section for `sched_yield()`.

4126 **RATIONALE**
4127 The `thrd_yield()` function is identical to the `sched_yield()` function except that it does not
4128 return a value.

4129 **FUTURE DIRECTIONS**
4130 None.

4131 **SEE ALSO**
4132 `sched_yield`
4133 XBD <**threads.h**>

4134 **CHANGE HISTORY**
4135 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4136 Ref 7.27.2.5
4137 On page 2161 line 69278 insert a new `timespec_get()` section:

4138 **NAME**
4139 `timespec_get` — get time

4140 **SYNOPSIS**
4141 `#include <time.h>`

4142 `int timespec_get(struct timespec *ts, int base);`

4143 **DESCRIPTION**

4144 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4145 Any conflict between the requirements described here and the ISO C standard is
4146 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4147 The *timespec_get()* function shall set the interval pointed to by *ts* to hold the current
4148 calendar time based on the specified time base.

4149 [CX]If *base* is `TIME_UTC`, the members of *ts* shall be set to the same values as would be
4150 set by a call to *clock_gettime(CLOCK_REALTIME, ts)*. If the number of seconds will not
4151 fit in an object of type **time_t**, the function shall return zero.[/CX]

4152 **RETURN VALUE**

4153 If the *timespec_get()* function is successful it shall return the non-zero value *base*; otherwise,
4154 it shall return zero.

4155 **ERRORS**

4156 See DESCRIPTION.

4157 **EXAMPLES**

4158 None.

4159 **APPLICATION USAGE**

4160 None.

4161 **RATIONALE**

4162 None.

4163 **FUTURE DIRECTIONS**

4164 None.

4165 **SEE ALSO**

4166 *clock_getres*, *time*

4167 XBD <**time.h**>

4168 **CHANGE HISTORY**

4169 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4170 Ref 7.21.4.4 para 4, 7.1.4 para 5

4171 On page 2164 line 69377 section *tmpnam()*, change:

4172 [CX]The *tmpnam()* function need not be thread-safe if called with a NULL parameter.[/CX]

4173 to:

4174 If called with a null pointer argument, the *tmpnam()* function need not be thread-safe;
4175 however, such calls shall avoid data races with calls to *tmpnam()* with a non-null argument
4176 and with calls to all other functions.

4177 Ref 7.30.3.2.1 para 4
4178 On page 2171 line 69568 section `towctrans()`, change:

4179 If successful, the `towctrans()` [CX]and `towctrans_l()`[/CX] functions shall return the mapped
4180 value of `wc` using the mapping described by `desc`. Otherwise, they shall return `wc`
4181 unchanged.

4182 to:

4183 If successful, the `towctrans()` [CX]and `towctrans_l()`[/CX] functions shall return the mapped
4184 value of `wc` using the mapping described by `desc`, or the value of `wc` unchanged if `desc` is
4185 zero. [CX]Otherwise, they shall return `wc` unchanged.[/CX]

4186 Ref F.10.6.8 para 2
4187 On page 2177 line 69716 section `trunc()`, add a new paragraph:

4188 [MX]These functions may raise the inexact floating-point exception for finite non-integer
4189 arguments.[/MX]

4190 Ref F.10.6.8 para 1,2
4191 On page 2177 line 69719 section `trunc()`, change:

4192 [MX]The result shall have the same sign as `x`.[/MX]

4193 to:

4194 [MX]The returned value shall be exact, shall be independent of the current rounding
4195 direction mode, and shall have the same sign as `x`.[/MX]

4196 Ref F.10.6.8 para 2
4197 On page 2177 line 69730 section `trunc()`, delete from APPLICATION USAGE:

4198 These functions may raise the inexact floating-point exception if the result differs in value
4199 from the argument.

4200 Ref 7.26.6
4201 On page 2182 line 69835 insert the following new `tss_*`() sections:

4202 **NAME**
4203 `tss_create` — thread-specific data key creation

4204 **SYNOPSIS**
4205 `#include <threads.h>`

4206 `int tss_create(tss_t *key, tss_dtor_t dtor);`

4207 **DESCRIPTION**
4208 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4209 Any conflict between the requirements described here and the ISO C standard is
4210 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4211 The `tss_create()` function shall create a thread-specific storage pointer with destructor `dtor`,
4212 which can be null.

4213 A null pointer value shall be associated with the newly created key in all existing threads.
4214 Upon subsequent thread creation, the value associated with all keys shall be initialized to a
4215 null pointer value in the new thread.

4216 Destructors associated with thread-specific storage shall not be invoked at process
4217 termination.

4218 The behavior is undefined if the *tss_create()* function is called from within a destructor.

4219 [CX]The *tss_create()* function shall not be affected if the calling thread executes a signal
4220 handler during the call.[/CX]

4221 **RETURN VALUE**

4222 If the *tss_create()* function is successful, it shall set the thread-specific storage pointed to by
4223 *key* to a value that uniquely identifies the newly created pointer and shall return
4224 *thrd_success*; otherwise, *thrd_error* shall be returned and the thread-specific storage
4225 pointed to by *key* has an indeterminate value.

4226 **ERRORS**

4227 No errors are defined.

4228 **EXAMPLES**

4229 None.

4230 **APPLICATION USAGE**

4231 The *tss_create()* function performs no implicit synchronization. It is the responsibility of the
4232 programmer to ensure that it is called exactly once per key before use of the key.

4233 **RATIONALE**

4234 If the value associated with a key needs to be updated during the lifetime of the thread, it
4235 may be necessary to release the storage associated with the old value before the new value is
4236 bound. Although the *tss_set()* function could do this automatically, this feature is not needed
4237 often enough to justify the added complexity. Instead, the programmer is responsible for
4238 freeing the stale storage:

```
4239 old = tss_get(key);  
4240 new = allocate();  
4241 destructor(old);  
4242 tss_set(key, new);
```

4243 There is no notion of a destructor-safe function. If an application does not call *thrd_exit()* or
4244 *pthread_exit()* from a signal handler, or if it blocks any signal whose handler may call
4245 *thrd_exit()* or *pthread_exit()* while calling async-unsafe functions, all functions can be safely
4246 called from destructors.

4247 The *tss_create()* function is not affected by signal handlers for the reasons stated in [xref to
4248 XRAT B.2.3].

4249 **FUTURE DIRECTIONS**

4250 None.

4251 **SEE ALSO**

4252 *pthread_exit, pthread_key_create, thrd_exit, tss_delete, tss_get*

4253 XBD <threads.h>

4254 CHANGE HISTORY

4255 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4256 NAME

4257 *tss_delete* — thread-specific data key deletion

4258 SYNOPSIS

4259 `#include <threads.h>`

4260 `void tss_delete(tss_t key);`

4261 DESCRIPTION

4262 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4263 Any conflict between the requirements described here and the ISO C standard is
4264 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4265 The *tss_delete()* function shall release any resources used by the thread-specific storage
4266 identified by *key*. The thread-specific data values associated with *key* need not be null at the
4267 time *tss_delete()* is called. It is the responsibility of the application to free any application
4268 storage or perform any cleanup actions for data structures related to the deleted key or
4269 associated thread-specific data in any threads; this cleanup can be done either before or after
4270 *tss_delete()* is called.

4271 The application shall ensure that the *tss_delete()* function is only called with a value for *key*
4272 that was returned by a call to *tss_create()* before the thread commenced executing
4273 destructors.

4274 If *tss_delete()* is called while another thread is executing destructors, whether this will affect
4275 the number of invocations of the destructor associated with *key* on that thread is unspecified.

4276 The *tss_delete()* function shall be callable from within destructor functions. Calling
4277 *tss_delete()* shall not result in the invocation of any destructors. Any destructor function that
4278 was associated with *key* shall no longer be called upon thread exit.

4279 Any attempt to use *key* following the call to *tss_delete()* results in undefined behavior.

4280 [CX]The *tss_delete()* function shall not be affected if the calling thread executes a signal
4281 handler during the call.[/CX]

4282 RETURN VALUE

4283 This function shall not return a value.

4284 ERRORS

4285 No errors are defined.

4286 EXAMPLES

4287 None.

4288 **APPLICATION USAGE**

4289 None.

4290 **RATIONALE**

4291 A thread-specific data key deletion function has been included in order to allow the
4292 resources associated with an unused thread-specific data key to be freed. Unused thread-
4293 specific data keys can arise, among other scenarios, when a dynamically loaded module that
4294 allocated a key is unloaded.

4295 Conforming applications are responsible for performing any cleanup actions needed for data
4296 structures associated with the key to be deleted, including data referenced by thread-specific
4297 data values. No such cleanup is done by *tss_delete()*. In particular, destructor functions
4298 are not called. See the RATIONALE for *pthread_key_delete()* for the reasons for this
4299 division of responsibility.

4300 The *tss_delete()* function is not affected by signal handlers for the reasons stated in [xref to
4301 XRAT B.2.3].

4302 **FUTURE DIRECTIONS**

4303 None.

4304 **SEE ALSO**

4305 *pthread_key_create*, *tss_create*

4306 XBD <**threads.h**>

4307 **CHANGE HISTORY**

4308 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4309 **NAME**

4310 *tss_get*, *tss_set* — thread-specific data management

4311 **SYNOPSIS**

4312 `#include <threads.h>`

4313 `void *tss_get(tss_t key);`
4314 `int tss_set(tss_t key, void *val);`

4315 **DESCRIPTION**

4316 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4317 Any conflict between the requirements described here and the ISO C standard is
4318 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4319 The *tss_get()* function shall return the value for the current thread held in the thread-specific
4320 storage identified by *key*.

4321 The *tss_set()* function shall set the value for the current thread held in the thread-specific
4322 storage identified by *key* to *val*. This action shall not invoke the destructor associated with
4323 the key on the value being replaced.

4324 The application shall ensure that the *tss_get()* and *tss_set()* functions are only called with a
4325 value for *key* that was returned by a call to *tss_create()* before the thread commenced

4326 executing destructors.

4327 The effect of calling *tss_get()* or *tss_set()* after *key* has been deleted with *tss_delete()* is
4328 undefined.

4329 [CX]Both *tss_get()* and *tss_set()* can be called from a thread-specific data destructor
4330 function. A call to *tss_get()* for the thread-specific data key being destroyed shall return a
4331 null pointer, unless the value is changed (after the destructor starts) by a call to *tss_set()*.
4332 Calling *tss_set()* from a thread-specific data destructor function may result either in lost
4333 storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction)
4334 or in an infinite loop.

4335 These functions shall not be affected if the calling thread executes a signal handler during
4336 the call.[/CX]

4337 **RETURN VALUE**

4338 The *tss_get()* function shall return the value for the current thread. If no thread-specific data
4339 value is associated with *key*, then a null pointer shall be returned.

4340 The *tss_set()* function shall return *thrd_success* on success or *thrd_error* if the request
4341 could not be honored.

4342 **ERRORS**

4343 No errors are defined.

4344 **EXAMPLES**

4345 None.

4346 **APPLICATION USAGE**

4347 None.

4348 **RATIONALE**

4349 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
4350 B.2.3].

4351 **FUTURE DIRECTIONS**

4352 None.

4353 **SEE ALSO**

4354 *pthread_getspecific*, *tss_create*

4355 XBD <**threads.h**>

4356 **CHANGE HISTORY**

4357 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4358 Ref 7.31.11 para 2

4359 On page 2193 line 70145 section *ungetc()*, change FUTURE DIRECTIONS from:

4360 None.

4361 to:

4362 The ISO C standard states that the use of *ungetc()* on a binary stream where the file position
4363 indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction
4364 between binary and text streams, so this applies to all streams. This feature may be removed
4365 in a future version of this standard.

4366 Ref 7.29.6.3 para 1, 7.1.4 para 5
4367 On page 2242 line 71441 section *wcrtomb()*, change:

4368 [CX]The *wcrtomb()* function need not be thread-safe if called with a NULL *ps*
4369 argument.[/CX]

4370 to:

4371 If called with a null *ps* argument, the *wcrtomb()* function need not be thread-safe; however,
4372 such calls shall avoid data races with calls to *wcrtomb()* with a non-null argument and with
4373 calls to all other functions.

4374 Ref 7.29.6.4 para 1, 7.1.4 para 5
4375 On page 2266 line 72111 section *wcsrtombs()*, change:

4376 [CX]The *wcsnrtombs()* and *wcsrtombs()* functions need not be thread-safe if called with a
4377 NULL *ps* argument.[/CX]

4378 to:

4379 [CX]If called with a null *ps* argument, the *wcsnrtombs()* function need not be thread-safe;
4380 however, such calls shall avoid data races with calls to *wcsnrtombs()* with a non-null
4381 argument and with calls to all other functions.[/CX]

4382 If called with a null *ps* argument, the *wcsrtombs()* function need not be thread-safe;
4383 however, such calls shall avoid data races with calls to *wcsrtombs()* with a non-null
4384 argument and with calls to all other functions.

4385 Ref 7.22.7 para 1, 7.1.4 para 5
4386 On page 2292 line 72879 section *wctomb()*, change:

4387 [CX]The *wctomb()* function need not be thread-safe.[/CX]

4388 to:

4389 The *wctomb()* function need not be thread-safe; however, it shall avoid data races with all
4390 other functions.

4391 **Changes to XCU**

4392 Ref 7.22.2
4393 On page 2333 line 74167 section 1.1.2.2 Mathematical Functions, change:

4394 Section 7.20.2, Pseudo-Random Sequence Generation Functions

4395 to:

4396 Section 7.22.2, Pseudo-Random Sequence Generation Functions

4397 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4398 On page 2542 line 82220 section c99, rename the c99 page to c17.

4399 Ref 7.26

4400 On page 2545 line 82375 section c99 (now c17), change:

4401 ... , `<spawn.h>`, `<sys/socket.h>`, ...

4402 to:

4403 ... , `<spawn.h>`, `<sys/socket.h>`, `<threads.h>`, ...

4404 Ref 7.26

4405 On page 2545 line 82382 section c99 (now c17), change:

4406 This option shall make available all interfaces referenced in `<pthread.h>` and `pthread_kill()`
4407 and `pthread_sigmask()` referenced in `<signal.h>`.

4408 to:

4409 This option shall make available all interfaces referenced in `<pthread.h>` and `<threads.h>`,
4410 and also `pthread_kill()` and `pthread_sigmask()` referenced in `<signal.h>`.

4411 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4412 On page 2552-2553 line 82641-82677 section c99 (now c17), change CHANGE HISTORY to:

4413 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4414 Changes to XRAT

4415 Ref G.1 para 1

4416 On page 3483 line 117680 section A.1.7.1 Codes, add a new tagged paragraph:

4417 MXC This margin code is used to denote functionality related to the IEC 60559 Complex
4418 Floating-Point option. ~~This functionality is mandated by the ISO C standard for IEC-~~
4419 ~~60559 implementations that support `<complex.h>`.~~

4420 Ref (none)

4421 On page 3489 line 117909 section A.3 Definitions (Byte), change:

4422 alignment with the ISO/IEC 9899: 1999 standard, where the `intN_t` types are now defined.

4423 to:

4424 alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types were first defined.

4425 Ref 5.1.2.4, 7.17.3

4426 On page 3515 line 118946 section A.4.12 Memory Synchronization, change:

4427 **A.4.12 Memory Synchronization**

4428 to:

4429 **A.4.12 Memory Ordering and Synchronization**

4430 *A.4.12.1 Memory Ordering*

4431 There is no additional rationale provided for this section.

4432 *A.4.12.2 Memory Synchronization*

4433 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4434 On page 3556 line 120684 section A.12.2 Utility Syntax Guidelines, change:

4435 Thus, they had to devise a new name, *c89* (now superseded by *c99*), rather than ...

4436 to:

4437 Thus, they had to devise a new name, *c89* (subsequently superseded by *c99* and now by
4438 *c17*), rather than ...

4439 Ref K.3.1.1

4440 On page 3567 line 121053 section B.2.2.1 POSIX.1 Symbols, add a new unnumbered subsection:

4441 **The `__STDC_WANT_LIB_EXT1__` Feature Test Macro**

4442 The ISO C standard specifies the feature test macro `__STDC_WANT_LIB_EXT1__` as the
4443 announcement mechanism for the application that it requires functionality from Annex K. It
4444 specifies that the symbols specified in Annex K (if supported) are made visible when
4445 `__STDC_WANT_LIB_EXT1__` is 1 and are not made visible when it is 0, but leaves it
4446 unspecified whether they are made visible when `__STDC_WANT_LIB_EXT1__` is
4447 undefined. POSIX.1 requires that they are not made visible when the macro is undefined
4448 (except for those symbols that are already explicitly allowed to be visible through the
4449 definition of `_POSIX_C_SOURCE` or `_XOPEN_SOURCE`, or both).

4450 POSIX.1 does not include the interfaces specified in Annex K of the ISO C standard, but
4451 allows the symbols to be made visible in headers when requested by the application in order
4452 that applications can use symbols from Annex K and symbols from POSIX.1 in the same
4453 translation unit.

4454 Ref 6.10.3.4

4455 On page 3570 line 121176 section B.2.2.2 The Name Space, change:

4456 as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C
4457 standard

4458 to:

4459 as described for macros that expand to their own name as in Section 6.10.3.4 of the ISO C
4460 standard

4461 Ref 7.5 para 2
4462 On page 3571 line 121228-121243 section B.2.3 Error Numbers, change:

4463 The ISO C standard requires that *errno* be an assignable lvalue. Originally, ...
4464 [...]
4465 ... using the return value for a mixed purpose was judged to be of limited use and
4466 error prone.

4467 to:
4468 The original ISO C standard just required that *errno* be an modifiable lvalue. Since the
4469 introduction of threads in 2011, the ISO C standard has instead required that *errno* be a
4470 macro which expands to a modifiable lvalue that has thread local storage duration.

4471 Ref 7.26
4472 On page 3575 line 121390 section B.2.3 Error Numbers, change:

4473 In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a
4474 special case since it will never occur.

4475 to:
4476 In particular, applications calling blocking interfaces need not handle any possible [EINTR]
4477 return as a special case since it will never occur. In the case of threads functions in
4478 <threads.h>, the requirement is stated in terms of the call not being affected if the calling
4479 thread executes a signal handler during the call, since these functions return errors in a
4480 different way and cannot distinguish an [EINTR] condition from other error conditions.

4481 Ref (none)
4482 On page 3733 line 128128 section C.2.6.4 Arithmetic Expansion, change:

4483 Although the ISO/IEC 9899: 1999 standard now requires support for ...

4484 to:
4485 Although the ISO C standard requires support for ...

4486 Ref 7.17
4487 On page 3789 line 129986 section E.1 Subprofiling Option Groups, change:

4488 by collecting sets of related functions

4489 to:
4490 by collecting sets of related functions and generic functions

4491 Ref 7.22.3.1, 7.27.2.5, 7.22.4
4492 On page 3789, 3792 line 130022-130032, 130112-130114 section E.1 Subprofiling Option Groups,
4493 add new functions (in sorted order) to the existing groups as indicated:

4494 POSIX_C_LANG_SUPPORT
4495 *aligned_alloc(), timespec_get()*

4496 POSIX_MULTI_PROCESS
4497 *at_quick_exit(), quick_exit()*

4498 Ref 7.17

4499 On page 3789 line 129991 section E.1 Subprofiling Option Groups, add:

4500 POSIX_C_LANG_ATOMICS: ISO C Atomic Operations
4501 *atomic_compare_exchange_strong(), atomic_compare_exchange_strong_explicit(),*
4502 *atomic_compare_exchange_weak(), atomic_compare_exchange_weak_explicit(),*
4503 *atomic_exchange(), atomic_exchange_explicit(), atomic_fetch_add(),*
4504 *atomic_fetch_add_explicit(), atomic_fetch_and(), atomic_fetch_and_explicit(),*
4505 *atomic_fetch_or(), atomic_fetch_or_explicit(), atomic_fetch_sub(),*
4506 *atomic_fetch_sub_explicit(), atomic_fetch_xor(), atomic_fetch_xor_explicit(),*
4507 *atomic_flag_clear(), atomic_flag_clear_explicit(), atomic_flag_test_and_set(),*
4508 *atomic_flag_test_and_set_explicit(), atomic_init(), atomic_is_lock_free(),*
4509 *atomic_load(), atomic_load_explicit(), atomic_signal_fence(),*
4510 *atomic_thread_fence(), atomic_store(), atomic_store_explicit(), kill_dependency()*

4511 Ref 7.26

4512 On page 3790 line 1300349 section E.1 Subprofiling Option Groups, add:

4513 POSIX_C_LANG_THREADS: ISO C Threads
4514 *call_once(), cnd_broadcast(), cnd_signal(), cnd_destroy(), cnd_init(),*
4515 *cnd_timedwait(), cnd_wait(), mtx_destroy(), mtx_init(), mtx_lock(), mtx_timedlock(),*
4516 *mtx_trylock(), mtx_unlock(), thrd_create(), thrd_current(), thrd_detach(),*
4517 *thrd_equal(), thrd_exit(), thrd_join(), thrd_sleep(), thrd_yield(), tss_create(),*
4518 *tss_delete(), tss_get(), tss_set()*

4519 POSIX_C_LANG_UCHAR: ISO C Unicode Utilities
4520 *c16rtomb(), c32rtomb(), mbrtoc16(), mbrtoc32()*