

1 Introduction

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

7 Note that the removal of *gets()* is not included here, as it is has already been removed by bug 1330.

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

9 Global Change

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

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

14 Changes to XBD

15 Ref G.1 para 1

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

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

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

20 Ref (none)

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

22 the ISO/IEC 9899: 1999 standard

23 to:

24 the ISO C standard

25 Ref 6.2.8

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

27 See also the ISO C standard, Section B3.

28 to:

29 See also the ISO C standard, Section 6.2.8.

30 Ref 5.1.2.4

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

32 **3.31 Atomic Operation**

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

37 Ref 7.26.3

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

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

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

46 Ref 5.1.2.4

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

48 **3.125 Data Race**

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

52 Ref 5.1.2.4

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

54 **3.215 Lock-Free Operation**

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

57 Ref 7.26.5.1

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

59 the process can create additional threads using `pthread_create()` or `SIGEV_THREAD`
60 notifications.

61 to:

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

64 Ref 7.26.4

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

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

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

72 Ref 7.26.5.5

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

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

76 to:

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

79 Ref 7.26.5.1

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

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

83 to:

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

86 Ref 5.1.2.4

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

88 **3.382 Synchronization Operation**

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

90 Ref 7.26.5.1

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

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

94 to:

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

- 97 • The initial thread.
- 98 • A thread created using *pthread_create()*.
- 99 • A thread created using *thrd_create()*.
- 100 • A thread created via a SIGEV_THREAD notification.

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

107 Ref 5.1.2.4
108 On page 99 line 2752 section 3.407 Thread-Safe, change:

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

111 to:

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

114 Ref 5.1.2.4
115 On page 99 line 2756 section 3.407 Thread-Safe, add a new paragraph:

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

119 Ref 7.26.6
120 On page 99 line 2758 section 3.408 Thread-Specific Data Key, change:

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

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

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

128 to:

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

134 Although the same key value can be used by different threads, the values bound to the key
135 by *pthread_setspecific()* for keys of type **pthread_key_t**, and by *tss_set()* for keys of type
136 **tss_t**, are maintained on a per-thread basis and persist for the life of the calling thread.

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

139 Ref 5.1.2.4, 7.17.3

140 On page 111 line 3060 section 4.12 Memory Synchronization, after applying bug 1426 change:

141 **4.12 Memory Synchronization**

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

148 to:

149 **4.12 Memory Ordering and Synchronization**

150 **4.12.1 Memory Ordering**

151 *4.12.1.1 Data Races*

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

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

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

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

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

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

178 **Note:** There is a separate order for each atomic object. There is no requirement that these can be

179 combined into a single total order for all objects. In general this will be impossible since
180 different threads may observe modifications to different variables in inconsistent orders.

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

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

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

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

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

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

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

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

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

- 213 • *A* synchronizes with *X* and *X* is sequenced before *B*,
- 214 • *A* is sequenced before *X* and *X* inter-thread happens before *B*, or
- 215 • *A* inter-thread happens before *X* and *X* inter-thread happens before *B*.

216 **Note:** The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced
217 before”, “synchronizes with”, and “dependency-ordered before” relationships, with two
218 exceptions. The first exception is that a concatenation is not permitted to end with
219 “dependency-ordered before” followed by “sequenced before”. The reason for this limitation
220 is that a consume operation participating in a “dependency-ordered before” relationship

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

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

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

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

- 234 • *A* happens before *B*, and
- 235 • there is no other side effect *X* to *M* such that *A* happens before *X* and *X* happens
236 before *B*.

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

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

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

246
247 The value of an atomic object *M*, as determined by evaluation *B*, shall be the value stored by
248 some side effect *A* that modifies *M*, where *B* does not happen before *A*.

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

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

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

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

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

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

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

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

275 4.12.1.2 Memory Order and Consistency

276 The enumerated type **memory_order**, defined in `<stdatomic.h>` (if supported), specifies
277 the detailed regular (non-atomic) memory synchronization operations as defined in [xref to
278 4.12.1.1] and may provide for operation ordering. Its enumeration constants specify memory
279 order as follows:

280 For `memory_order_relaxed`, no operation orders memory.

281 For `memory_order_release`, `memory_order_acq_rel`, and
282 `memory_order_seq_cst`, a store operation performs a release operation on the affected
283 memory location.

284 For `memory_order_acquire`, `memory_order_acq_rel`, and
285 `memory_order_seq_cst`, a load operation performs an acquire operation on the affected
286 memory location.

287 For `memory_order_consume`, a load operation performs a consume operation on the
288 affected memory location.

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

- 293 • the result of the last modification *A* of *M* that precedes *B* in *S*, if it exists, or
- 294 • if *A* exists, the result of some modification of *M* that is not
295 `memory_order_seq_cst` and that does not happen before *A*, or
- 296 • if *A* does not exist, the result of some modification of *M* that is not
297 `memory_order_seq_cst`.

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

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

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

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

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

- 314 • there is a `memory_order_seq_cst` fence *X* such that *A* is sequenced before *X*, and
315 *X* precedes *B* in *S*, or
- 316 • there is a `memory_order_seq_cst` fence *Y* such that *Y* is sequenced before *B*, and
317 *A* precedes *Y* in *S*, or
- 318 • there are `memory_order_seq_cst` fences *X* and *Y* such that *A* is sequenced before
319 *X*, *Y* is sequenced before *B*, and *X* precedes *Y* in *S*.

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

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

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

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

333 4.12.2 Memory Synchronization

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

340 Ref 7.26.3, 7.26.4

341 On page 111 line 3066-3075 section 4.12 Memory Synchronization, add the following to the list of
342 functions that synchronize memory on all successful calls:

343 `cond_broadcast()` `thrd_create()`

344 *cond_signal()* *thrd_join()*

345 Ref 7.26.2.1, 7.26.4

346 On page 111 line 3076 section 4.12 Memory Synchronization, after applying bugs 1216 and 1426
347 change:

348 The *pthread_once()* function shall synchronize memory for the first successful call in each
349 thread for a given **pthread_once_t** object. If the *init_routine* called by *pthread_once()* is a
350 cancellation point and is canceled, a successful call to *pthread_once()* for the same
351 **pthread_once_t** object made from a cancellation cleanup handler shall also synchronize
352 memory.

353 The *pthread_mutex_clocklock()*, *pthread_mutex_lock()*,
354 [RPP|TPP]*pthread_mutex_setprioceiling()*, [TPP|TPP] *pthread_mutex_timedlock()*, and
355 *pthread_mutex_trylock()* functions shall synchronize memory on all calls that acquire the
356 mutex, including those that return [EOWNERDEAD]. The *pthread_mutex_unlock()* function
357 shall synchronize memory on all calls that release the mutex.

358 **Note:** If the mutex type is PTHREAD_MUTEX_RECURSIVE, calls to the locking functions do
359 not acquire the mutex if the calling thread already owns it, and calls to
360 *pthread_mutex_unlock()* do not release the mutex if it has a lock count greater than one.

361 The *pthread_cond_clockwait()*, *pthread_cond_wait()*, and *pthread_cond_timedwait()*
362 functions shall synchronize memory on all calls that release and re-acquire the specified
363 mutex, including calls that return [EOWNERDEAD], both when the mutex is released and
364 when it is re-acquired.

365 **Note:** If the mutex type is PTHREAD_MUTEX_RECURSIVE, calls to *pthread_cond_clockwait()*,
366 *pthread_cond_wait()*, and *pthread_cond_timedwait()* do not release and re-acquire the mutex
367 if it has a lock count greater than one.

368 to:

369 The *pthread_once()* and *call_once()* functions shall synchronize memory for the first
370 successful call in each thread for a given **pthread_once_t** or **once_flag** object, respectively.
371 If the *init_routine* called by *pthread_once()* or *call_once()* is a cancellation point and is
372 canceled, a successful call to *pthread_once()* for the same **pthread_once_t** object, or to
373 *call_once()* for the same **once_flag** object, made from a cancellation cleanup handler shall
374 also synchronize memory.

375 The *pthread_mutex_clocklock()*, *pthread_mutex_lock()*,
376 [RPP|TPP]*pthread_mutex_setprioceiling()*, [TPP|TPP] *pthread_mutex_timedlock()*, and
377 *pthread_mutex_trylock()* functions shall synchronize memory on all calls that acquire the
378 mutex, including those that return [EOWNERDEAD]. The *pthread_mutex_unlock()* function
379 shall synchronize memory on all calls that release the mutex.

380 **Note:** If the mutex type is PTHREAD_MUTEX_RECURSIVE, calls to the locking functions do
381 not acquire the mutex if the calling thread already owns it, and calls to
382 *pthread_mutex_unlock()* do not release the mutex if it has a lock count greater than one.

383 The *pthread_cond_clockwait()*, *pthread_cond_wait()*, and *pthread_cond_timedwait()*
384 functions shall synchronize memory on all calls that release and re-acquire the specified
385 mutex, including calls that return [EOWNERDEAD], both when the mutex is released and

386 when it is re-acquired.

387 **Note:** If the mutex type is PTHREAD_MUTEX_RECURSIVE, calls to *pthread_cond_clockwait()*,
388 *pthread_cond_wait()*, and *pthread_cond_timedwait()* do not release and re-acquire the mutex
389 if it has a lock count greater than one.

390 The *mtx_lock()*, *mtx_timedlock()*, and *mtx_trylock()* functions shall synchronize memory on
391 all calls that acquire the mutex. The *mtx_unlock()* function shall synchronize memory on all
392 calls that release the mutex.

393 **Note:** If the mutex is a recursive mutex, calls to the locking functions do not acquire the mutex if
394 the calling thread already owns it, and calls to *mtx_unlock()* do not release the mutex if it has
395 a lock count greater than one.

396 The *cond_wait()* and *cond_timedwait()* functions shall synchronize memory on all calls that
397 release and re-acquire the specified mutex, both when the mutex is released and when it is
398 re-acquired.

399 **Note:** If the mutex is a recursive mutex, calls to *cond_wait()* and *cond_timedwait()* do not release and
400 re-acquire the mutex if it has a lock count greater than one.

401 Ref 7.26.4

402 On page 111 line 3087 section 4.12 Memory Synchronization, add a new paragraph:

403 For purposes of determining the existence of a data race, all lock and unlock operations on a
404 particular synchronization object that synchronize memory shall behave as atomic
405 operations, and they shall occur in some particular total order (see [xref to 4.12.1]).

406 Ref 7.12.1 para 7

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

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

410 to:

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

416 Ref 7.12.1 para 3

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

418 A ``pole error'' occurs if the mathematical result of the function is an exact infinity (for
419 example, $\log(0.0)$).

420 to:

421 A ``pole error'' shall occur if the mathematical result of the function has an exact infinite
422 result as the finite input argument(s) are approached in the limit (for example, $\log(0.0)$). The
423 description of each function lists any required pole errors; an implementation may define

424 additional pole errors, provided that such errors are consistent with the mathematical
425 definition of the function.

426 Ref 7.12.1 para 4

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

428 A ``range error'' shall occur if the finite mathematical result of the function cannot be
429 represented in an object of the specified type, due to extreme magnitude.

430 add:

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

434 Ref 7.29.1 para 5

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

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

443 Ref 7.3.1 para 2

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

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

450 Ref G.6 para 1

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

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

454 add:

455 [MXC] Implementations that support the IEC 60559 Complex Floating-Point option shall
456 define the macros `imaginary` and `_Imaginary_I`, and the macro `I` shall expand to
457 `_Imaginary_I`.[/MXC]

458 Ref 7.3.9.3

459 On page 224 line 7553 section `<complex.h>`, add:

460 The following shall be defined as macros.

461 double complex CMPLX(double x, double y);
462 float complex CMPLXF(float x, float y);
463 long double complex CMPLXL(long double x, long double y);

464 Ref 7.3.1 para 2

465 On page 226 line 7623 section <complex.h>, add a new first paragraph to APPLICATION USAGE:

466 The <**complex.h**> header is optional in the ISO C standard but is mandated by POSIX.1-
467 20xx. Note however that subprofiles can choose to make this header optional (see [xref to
468 2.1.5.1 Subprofiling Considerations]), and therefore application portability to subprofile
469 implementations would benefit from checking whether `__STDC_NO_COMPLEX__` is
470 defined before inclusion of <**complex.h**>.

471 Ref 7.3.9.3

472 On page 226 line 7649 section <complex.h>, add CMPLX() to the SEE ALSO list before cabs().

473 Ref 7.5 para 2

474 On page 234 line 7876 section <errno.h>, change:

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

478 to:

479 The <**errno.h**> header shall provide a definition for the macro *errno*, which shall expand to
480 a modifiable lvalue of type **int** and thread local storage duration.

481 Ref (none)

482 On page 245 line 8290 section <fenv.h>, change:

483 the ISO/IEC 9899: 1999 standard

484 to:

485 the ISO C standard

486 Ref 5.2.4.2.2 para 11

487 On page 248 line 8369 section <float.h>, add the following new paragraphs:

488 The presence or absence of subnormal numbers is characterized by the implementation-
489 defined values of FLT_HAS_SUBNORM , DBL_HAS_SUBNORM , and
490 LDBL_HAS_SUBNORM :

-1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

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

495 Ref 5.2.4.2.2 para 12

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

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

500 [math stuff]

501 FLT_DECIMAL_DIG 6

502 DBL_DECIMAL_DIG 10

503 LDBL_DECIMAL_DIG 10

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

506 Ref 5.2.4.2.2 para 14

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

508 Minimum positive floating-point number.

509 FLT_TRUE_MIN 1E-37

510 DBL_TRUE_MIN 1E-37

511 LDBL_TRUE_MIN 1E-37

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

515 Ref (none)

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

517 the ISO/IEC 9899: 1999 standard

518 to:

519 the ISO C standard

520 Ref 7.22.4.3

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

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

523 to:

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

526 Ref 5.2.4.2.1 para 2

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

528 If the value of an object of type **char** is treated as a signed integer when used in an
529 expression, the value of {CHAR_MIN} is the same as that of {SCHAR_MIN} and the value
530 of {CHAR_MAX} is the same as that of {SCHAR_MAX}. Otherwise, the value of
531 {CHAR_MIN} is 0 and the value of {CHAR_MAX} is the same as that of
532 {UCHAR_MAX}.

533 to:

534 If an object of type **char** can hold negative values, the value of {CHAR_MIN} shall be the
535 same as that of {SCHAR_MIN} and the value of {CHAR_MAX} shall be the same as that
536 of {SCHAR_MAX}. Otherwise, the value of {CHAR_MIN} shall be 0 and the value of
537 {CHAR_MAX} shall be the same as that of {UCHAR_MAX}.

538 Ref (none)

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

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

541 to:

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

543 Ref 7.26.5.5

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

545 void pthread_exit(void *);

546 to:

547 _Noreturn void pthread_exit(void *);

548 Ref 7.13.2.1 para 1

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

550 void longjmp(jmp_buf, int);
551 [CX]void siglongjmp(sigjmp_buf, int);[/CX]

552 to:

553 _Noreturn void longjmp(jmp_buf, int);
554 [CX]_Noreturn void siglongjmp(sigjmp_buf, int);[/CX]

555 Ref 7.15

556 On page 343 line 11647 insert a new <stdalign.h> section:

557 **NAME**

558 stdalign.h — alignment macros

559 **SYNOPSIS**

560 #include <stdalign.h>

561 **DESCRIPTION**

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

565 The `<stdalign.h>` header shall define the following macros:

566 `alignas` Expands to `_Alignas`

567 `alignof` Expands to `_Alignof`

568 `__alignas_is_defined`
569 Expands to the integer constant 1

570 `__alignof_is_defined`
571 Expands to the integer constant 1

572 The `__alignas_is_defined` and `__alignof_is_defined` macros shall be suitable for use in `#if`
573 preprocessing directives.

574 **APPLICATION USAGE**

575 None.

576 **RATIONALE**

577 None.

578 **FUTURE DIRECTIONS**

579 None.

580 **SEE ALSO**

581 None.

582 **CHANGE HISTORY**

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

584 Ref 7.17, 7.31.8 para 2
585 On page 345 line 11733 insert a new `<stdatomic.h>` section:

586 **NAME**

587 `stdatomic.h` — atomics

588 **SYNOPSIS**

589 `#include <stdatomic.h>`

590 **DESCRIPTION**

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

594 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide this

595 header nor support any of its facilities.

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

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

602 **Note:** The same representation and alignment requirements are meant to imply interchangeability
603 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
atomic_uintptr_t	_Atomic uintptr_t
atomic_size_t	_Atomic size_t
atomic_ptrdiff_t	_Atomic ptrdiff_t
atomic_intmax_t	_Atomic intmax_t
atomic_uintmax_t	_Atomic uintmax_t

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

606 memory_order_relaxed
607 memory_order_consume
608 memory_order_acquire
609 memory_order_release
610 memory_order_acq_rel
611 memory_order_seq_cst

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

613 ATOMIC_BOOL_LOCK_FREE
614 ATOMIC_CHAR_LOCK_FREE
615 ATOMIC_CHAR16_T_LOCK_FREE
616 ATOMIC_CHAR32_T_LOCK_FREE
617 ATOMIC_WCHAR_T_LOCK_FREE
618 ATOMIC_SHORT_LOCK_FREE
619 ATOMIC_INT_LOCK_FREE
620 ATOMIC_LONG_LOCK_FREE
621 ATOMIC_LLONG_LOCK_FREE
622 ATOMIC_POINTER_LOCK_FREE

623 which shall expand to constant expressions suitable for use in **#if** preprocessing directives
624 and which shall indicate the lock-free property of the corresponding atomic types (both
625 signed and unsigned). A value of 0 shall indicate that the type is never lock-free; a value of 1
626 shall indicate that the type is sometimes lock-free; a value of 2 shall indicate that the type is
627 always lock-free.

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

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

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

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

642 **_Bool** atomic_compare_exchange_strong(volatile **A** *, **C** *, **C**);
643 **_Bool** atomic_compare_exchange_strong_explicit(volatile **A** *,
644 **C** *, **C**, memory_order, memory_order);
645 **_Bool** atomic_compare_exchange_weak(volatile **A** *, **C** *, **C**);
646 **_Bool** atomic_compare_exchange_weak_explicit(volatile **A** *, **C** *,
647 **C**, memory_order, memory_order);
648 **C** atomic_exchange(volatile **A** *, **C**);

```

649     C      atomic_exchange_explicit(volatile A *, C, memory_order);
650     C      atomic_fetch_add(volatile A *, M);
651     C      atomic_fetch_add_explicit(volatile A *, M,
652         memory_order);
653     C      atomic_fetch_and(volatile A *, M);
654     C      atomic_fetch_and_explicit(volatile A *, M,
655         memory_order);
656     C      atomic_fetch_or(volatile A *, M);
657     C      atomic_fetch_or_explicit(volatile A *, M, memory_order);
658     C      atomic_fetch_sub(volatile A *, M);
659     C      atomic_fetch_sub_explicit(volatile A *, M,
660         memory_order);
661     C      atomic_fetch_xor(volatile A *, M);
662     C      atomic_fetch_xor_explicit(volatile A *, M,
663         memory_order);
664     void    atomic_init(volatile A *, C);
665     _Bool   atomic_is_lock_free(const volatile A *);
666     C      atomic_load(const volatile A *);
667     C      atomic_load_explicit(const volatile A *, memory_order);
668     void    atomic_store(volatile A *, C);
669     void    atomic_store_explicit(volatile A *, C, memory_order);

```

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

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

```

676     void    atomic_flag_clear(volatile atomic_flag *);
677     void    atomic_flag_clear_explicit(volatile atomic_flag *,
678         memory_order);
679     _Bool   atomic_flag_test_and_set(volatile atomic_flag *);
680     _Bool   atomic_flag_test_and_set_explicit(
681         volatile atomic_flag *, memory_order);
682     void    atomic_signal_fence(memory_order);
683     void    atomic_thread_fence(memory_order);

```

684 APPLICATION USAGE

685 None.

686 RATIONALE

687 Since operations on the **atomic_flag** type are lock free, the operations should also be
688 address-free. No other type requires lock-free operations, so the **atomic_flag** type is the
689 minimum hardware-implemented type needed to conform to this standard. The remaining
690 types can be emulated with **atomic_flag**, though with less than ideal properties.

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

694 FUTURE DIRECTIONS

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

697 **SEE ALSO**

698 Section 4.12.1

699 XSH *atomic_compare_exchange_strong()*, *atomic_compare_exchange_weak()*,
700 *atomic_exchange()*, *atomic_fetch_key()*, *atomic_flag_clear()*, *atomic_flag_test_and_set()*,
701 *atomic_init()*, *atomic_is_lock_free()*, *atomic_load()*, *atomic_signal_fence()*, *atomic_store()*,
702 *atomic_thread_fence()*, *kill_dependency()*.

703 **CHANGE HISTORY**

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

705 Ref 7.31.9

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

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

708 Ref 7.19 para 2

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

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

711 Ref (none)

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

713 the ISO/IEC 9899: 1999 standard

714 to:

715 the ISO C standard

716 Ref 7.20.1.1 para 1

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

718 denotes a signed integer type

719 to:

720 denotes such a signed integer type

721 Ref 7.20.1.1 para 2

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

723 ... designates an unsigned integer type with width *N*. Thus, **uint24_t** denotes an unsigned
724 integer type ...

725 to:

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

728 Ref 7.21.1 para 2

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

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

732 to:

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

735 Ref 7.21.1 para 3

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

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

741 to:

742 None.

743 Ref 7.22.4.5 para 1

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

745 void _Exit(int);

746 to:

747 _Noreturn void _Exit(int);

748 Ref 7.22.4.1 para 1

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

750 void abort(void);

751 to:

752 _Noreturn void abort(void);

753 Ref 7.22.3.1, 7.22.4.3

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

755 void *aligned_alloc(size_t, size_t);
756 int at_quick_exit(void (*)(void));

757 Ref 7.22.4.4 para 1

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

759 void exit(int);

760 to:

761 `_Noreturn void exit(int);`

762 Ref 7.22.4.7

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

764 `_Noreturn void quick_exit(int);`

765 Ref 7.23

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

767 **NAME**

768 `stdnoreturn.h` — noreturn macro

769 **SYNOPSIS**

770 `#include <stdnoreturn.h>`

771 **DESCRIPTION**

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

775 The <**stdnoreturn.h**> header shall define the macro `noreturn` which shall expand to
776 **_Noreturn**.

777 **APPLICATION USAGE**

778 None.

779 **RATIONALE**

780 None.

781 **FUTURE DIRECTIONS**

782 None.

783 **SEE ALSO**

784 None.

785 **CHANGE HISTORY**

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

787 Ref G.7

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

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

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

798 $\cos(iy)$ = $\cosh(y)$
799 $\sin(iy)$ = $i \sinh(y)$
800 $\tan(iy)$ = $i \tanh(y)$
801 $\cosh(iy)$ = $\cos(y)$
802 $\sinh(iy)$ = $i \sin(y)$
803 $\tanh(iy)$ = $i \tan(y)$
804 $\operatorname{asin}(iy)$ = $i \operatorname{asinh}(y)$
805 $\operatorname{atan}(iy)$ = $i \operatorname{atanh}(y)$
806 $\operatorname{asinh}(iy)$ = $i \operatorname{asin}(y)$
807 $\operatorname{atanh}(iy)$ = $i \operatorname{atan}(y)$
808 [/MXC]

809 Ref (none)

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

811 the ISO/IEC 9899: 1999 standard

812 to:

813 the ISO C standard

814 Ref 7.26

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

816 **NAME**

817 threads.h — ISO C threads

818 **SYNOPSIS**

819 #include <threads.h>

820 **DESCRIPTION**

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

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

828 The <**threads.h**> header shall define the following macros:

829 `thread_local` Expands to **`_Thread_local`**.

830 `ONCE_FLAG_INIT` Expands to a value that can be used to initialize an object of
831 type **`once_flag`**.

832 `TSS_DTOR_ITERATIONS` Expands to an integer constant expression representing the
833 maximum number of times that destructors will be called
834 when a thread terminates and shall be suitable for use in **`#if`**
835 preprocessing directives.

836 [CX]If {PTHREAD_DESTRUCTOR_ITERATIONS} is defined in <limits.h>, the value of
837 TSS_DTOR_ITERATIONS shall be equal to
838 {PTHREAD_DESTRUCTOR_ITERATIONS}; otherwise, the value of
839 TSS_DTOR_ITERATIONS shall be greater than or equal to the value of
840 {_POSIX_THREAD_DESTRUCTOR_ITERATIONS} and shall be less than or equal to the
841 maximum positive value that can be returned by a call to
842 `sysconf(_SC_THREAD_DESTRUCTOR_ITERATIONS)` in any process.[CX]

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

847 The <threads.h> header shall define the enumeration constants `mtx_plain`,
848 `mtx_recursive`, `mtx_timed`, `thrd_busy`, `thrd_error`, `thrd_nomem`, `thrd_success`
849 and `thrd_timedout`.

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

```
852 void          call_once(once_flag *, void (*)(void));
853 int           cnd_broadcast(cnd_t *);
854 void         cnd_destroy(cnd_t *);
855 int          cnd_init(cnd_t *);
856 int          cnd_signal(cnd_t *);
857 int          cnd_timedwait(cnd_t * restrict, mtx_t * restrict,
858                          const struct timespec * restrict);
859 int          cnd_wait(cnd_t *, mtx_t *);
860 void         mtx_destroy(mtx_t *);
861 int          mtx_init(mtx_t *, int);
862 int          mtx_lock(mtx_t *);
863 int          mtx_timedlock(mtx_t * restrict,
864                          const struct timespec * restrict);
865 int          mtx_trylock(mtx_t *);
866 int          mtx_unlock(mtx_t *);
867 int          thrd_create(thrd_t *, thrd_start_t, void *);
868 thrd_t      thrd_current(void);
869 int          thrd_detach(thrd_t);
870 int          thrd_equal(thrd_t, thrd_t);
871 _Noreturn void thrd_exit(int);
872 int          thrd_join(thrd_t, int *);
873 int          thrd_sleep(const struct timespec *,
874                          struct timespec *);
875 void         thrd_yield(void);
876 int          tss_create(tss_t *, tss_dtor_t);
877 void         tss_delete(tss_t);
878 void         *tss_get(tss_t);
879 int          tss_set(tss_t, void *);
```

880 Inclusion of the <threads.h> header shall make symbols defined in the header <time.h>
881 visible.

882 APPLICATION USAGE

883 The <threads.h> header is optional in the ISO C standard but is mandated by POSIX.1-

884 20xx. Note however that subprofiles can choose to make this header optional (see [xref to
885 2.1.5.1 Subprofiling Considerations]), and therefore application portability to subprofile
886 implementations would benefit from checking whether `__STDC_NO_THREADS__` is
887 defined before inclusion of `<threads.h>`.

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

894 RATIONALE

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

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

906 The standard developers considered making it implementation-defined whether the types
907 `cond_t`, `mtx_t` and `tss_t` are interchangeable with the corresponding types `pthread_cond_t`,
908 `pthread_mutex_t` and `pthread_key_t` defined in `<pthread.h>` (that is, whether any
909 function that can be called with a valid `cond_t` can also be called with a valid
910 `pthread_cond_t`, and vice versa, and likewise for the other types). However, this would
911 have meant extending `mtx_lock()` to provide a way for it to indicate that the owner of a
912 mutex has terminated (equivalent to [EOWNERDEAD]). It was felt that such an extension
913 would be invention. Although there was no similar concern for `cond_t` and `tss_t`, they were
914 treated the same way as `mtx_t` for consistency. See also the RATIONALE for `mtx_lock()`
915 concerning the inability of `mtx_t` to contain information about whether or not a mutex
916 supports timeout if it is the same type as `pthread_mutex_t`.

917 FUTURE DIRECTIONS

918 None.

919 SEE ALSO

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

921 XSH Section 2.9, `call_once()`, `cond_broadcast()`, `cond_destroy()`, `cond_timedwait()`,
922 `mtx_destroy()`, `mtx_lock()`, `sysconf()`, `thrd_create()`, `thrd_current()`, `thrd_detach()`,
923 `thrd_equal()`, `thrd_exit()`, `thrd_join()`, `thrd_sleep()`, `thrd_yield()`, `tss_create()`, `tss_delete()`,
924 `tss_get()`.

925 CHANGE HISTORY

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

927 Ref 7.27.1 para 4
928 On page 425 line 14453 section <time.h>, remove the CX shading from:

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

931 `time_t` `tv_sec` Seconds.
932 `long` `tv_nsec` Nanoseconds.

933 and change the members to:

934 `time_t` `tv_sec` Whole seconds.
935 `long` `tv_nsec` Nanoseconds [0, 999 999 999].

936 Ref 7.27.1 para 2
937 On page 426 line 14467 section <time.h>, add to the list of macros:

938 `TIME_UTC` An integer constant greater than 0 that designates the UTC time base
939 in calls to *timespec_get()*. The value shall be suitable for use in **#if**
940 preprocessing directives.

941 Ref 7.27.2.5
942 On page 427 line 14524 section <time.h>, add to the list of functions:

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

944 Ref 7.28
945 On page 433 line 14736 insert a new <uchar.h> section:

946 **NAME**
947 `uchar.h` — Unicode character handling

948 **SYNOPSIS**
949 `#include <uchar.h>`

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

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

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

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

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

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

959 The following shall be declared as functions and may also be defined as macros. Function

960 prototypes shall be provided.

```
961     size_t      c16rtomb(char *restrict, char16_t,  
962                   mbstate_t *restrict);  
963     size_t      c32rtomb(char *restrict, char32_t,  
964                   mbstate_t *restrict);  
965     size_t      mbrtoc16(char16_t *restrict, const char *restrict,  
966                   size_t, mbstate_t *restrict);  
967     size_t      mbrtoc32(char32_t *restrict, const char *restrict,  
968                   size_t, mbstate_t *restrict);
```

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

971 **APPLICATION USAGE**

972 None.

973 **RATIONALE**

974 None.

975 **FUTURE DIRECTIONS**

976 None.

977 **SEE ALSO**

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

979 **XSH** `c16rtomb()`, `c32rtomb()`, `mbrtoc16()`, `mbrtoc32()`

980 **CHANGE HISTORY**

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

982 Ref 7.22.4.5 para 1

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

```
984     void          _exit(int);
```

985 to:

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

987 Ref 7.29.1 para 2

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

```
989     mbstate_t    An object type other than an array type ...
```

990 to:

```
991     mbstate_t    A complete object type other than an array type ...
```

992 **Changes to XSH**

993 Ref 7.1.4 paras 5, 6
994 On page 471 line 16224 section 2.1.1 Use and Implementation of Functions, add two numbered list
995 items:

996 6. Functions shall prevent data races as follows: A function shall not directly or indirectly
997 access objects accessible by threads other than the current thread unless the objects are
998 accessed directly or indirectly via the function's arguments. A function shall not directly or
999 indirectly modify objects accessible by threads other than the current thread unless the
1000 objects are accessed directly or indirectly via the function's non-const arguments.
1001 Implementations may share their own internal objects between threads if the objects are not
1002 visible to applications and are protected against data races.

1003 7. Functions shall perform all operations solely within the current thread if those operations
1004 have effects that are visible to applications.

1005 Ref K.3.1.1
1006 On page 473 line 16283 section 2.2.1, add a new subsection:

1007 2.2.1.3 *The `__STDC_WANT_LIB_EXT1__` Feature Test Macro*

1008 A POSIX-conforming [XSI] or XSI-conforming[/XSI] application can define the feature test
1009 macro `__STDC_WANT_LIB_EXT1__` before inclusion of any header.

1010 When an application includes a header described by POSIX.1-20xx, and when this feature
1011 test macro is defined to have the value 1, the header may make visible those symbols
1012 specified for the header in Annex K of the ISO C standard that are not already explicitly
1013 permitted by POSIX.1-20xx to be made visible in the header. These symbols are listed in
1014 [xref to 2.2.2].

1015 When an application includes a header described by POSIX.1-20xx, and when this feature
1016 test macro is either undefined or defined to have the value 0, the header shall not make any
1017 additional symbols visible that are not already made visible by the feature test macro
1018 `_POSIX_C_SOURCE` [XSI] or `_XOPEN_SOURCE`[/XSI] as described above, except when
1019 enabled by another feature test macro.

1020 Ref 7.31.8 para 1
1021 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>		
----------------------------------	---	--	--

1022 Ref 7.31.15 para 1
1023 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>		
--------------------------------	--	--	--

1024 Ref 7.31.8 para 1
1025 On page 477 line 16410 section 2.2.2, insert a row in the table:

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

- 1026 Ref 7.31.14 para 1
 1027 On page 477 line 16417 section 2.2.2, insert a row in the table:

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

- 1028 Ref K.3.4 - K.3.9
 1029 On page 477 line 16436 section 2.2.2 The Name Space, add:

- 1030 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
 1031 (see [xref to 2.2.1]), implementations may add symbols to the headers shown in the
 1032 following table provided the identifiers for those symbols have one of the corresponding
 1033 complete names in the table.

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

- 1034 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
 1035 (see [xref to 2.2.1]), if any header in the following table is included, macros with the
 1036 complete names shown may be defined.

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

- 1037 **Note:** The above two tables only include those symbols from Annex K of the ISO C standard that
 1038 are not already allowed to be visible by entries in earlier tables in this section.

- 1039 Ref 7.1.3 para 1
 1040 On page 478 line 16438 section 2.2.2, change:

- 1041 With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that
 1042 begin with an <underscore> and either an uppercase letter or another <underscore> are
 1043 always reserved for any use by the implementation.

- 1044 to:

- 1045 With the exception of identifiers beginning with the prefix `_POSIX_` and those identifiers
 1046 which are lexically identical to keywords defined by the ISO C standard (for example
 1047 `_Bool`), all identifiers that begin with an <underscore> and either an uppercase letter or
 1048 another <underscore> are always reserved for any use by the implementation.

1049 Ref 7.1.3 para 1

1050 On page 478 line 16448 section 2.2.2, change:

1051 that have external linkage are always reserved

1052 to:

1053 that have external linkage and *errno* are always reserved

1054 Ref 7.1.3 para 1

1055 On page 479 line 16453 section 2.2.2, add the following in the appropriate place in the list:

1056	aligned_alloc	c32rtomb
1057	at_quick_exit	call_once
1058	atomic_compare_exchange_strong	cnd_broadcast
1059	atomic_compare_exchange_strong_explicit	cnd_destroy
1060	atomic_compare_exchange_weak	cnd_init
1061	atomic_compare_exchange_weak_explicit	cnd_signal
1062	atomic_exchange	cnd_timedwait
1063	atomic_exchange_explicit	cnd_wait
1064	atomic_fetch_add	kill_dependency
1065	atomic_fetch_add_explicit	mbrtoc16
1066	atomic_fetch_and	mbrtoc32
1067	atomic_fetch_and_explicit	mtx_destroy
1068	atomic_fetch_or	mtx_init
1069	atomic_fetch_or_explicit	mtx_lock
1070	atomic_fetch_sub	mtx_timedlock
1071	atomic_fetch_sub_explicit	mtx_trylock
1072	atomic_fetch_xor	mtx_unlock
1073	atomic_fetch_xor_explicit	quick_exit
1074	atomic_flag_clear	thrd_create
1075	atomic_flag_clear_explicit	thrd_current
1076	atomic_flag_test_and_set	thrd_detach
1077	atomic_flag_test_and_set_explicit	thrd_equal
1078	atomic_init	thrd_exit
1079	atomic_is_lock_free	thrd_join
1080	atomic_load	thrd_sleep
1081	atomic_load_explicit	thrd_yield
1082	atomic_signal_fence	timespec_get
1083	atomic_store	tss_create
1084	atomic_store_explicit	tss_delete
1085	atomic_thread_fence	tss_get
1086	c16rtomb	tss_set

1087 Ref 7.1.2 para 4

1088 On page 480 line 16551 section 2.2.2, change:

1089 Prior to the inclusion of a header, the application shall not define any macros with names
1090 lexically identical to symbols defined by that header.

1091 to:

1092 Prior to the inclusion of a header, or when any macro defined in the header is expanded, the
1093 application shall not define any macros with names lexically identical to symbols defined by
1094 that header.

1095 Ref 7.26.5.1

1096 On page 490 line 16980 section 2.4.2 Realtime Signal Generation and Delivery, change:

1097 The function shall be executed in an environment as if it were the *start_routine* for a newly
1098 created thread with thread attributes specified by *sigev_notify_attributes*.

1099 to:

1100 The function shall be executed in a newly created thread as if it were the *start_routine* for a
1101 call to *pthread_create()* with the thread attributes specified by *sigev_notify_attributes*.

1102 Ref 7.14.1.1 para 5

1103 On page 493 line 17088 section 2.4.3 Signal Actions, change:

1104 with static storage duration

1105 to:

1106 with static or thread storage duration that is not a lock-free atomic object

1107 Ref 7.14.1.1 para 5

1108 On page 493 line 17090 section 2.4.3 Signal Actions, after applying bug 711 change:

1109 other than one of the functions and macros listed in the following table

1110 to:

1111 other than one of the functions and macros specified below as being async-signal-safe

1112 Ref 7.14.1.1 para 5

1113 On page 494 line 17133 section 2.4.3 Signal Actions, add *quick_exit()* to the table of async-signal-
1114 safe functions.

1115 Ref 7.14.1.1 para 5

1116 On page 494 line 17147 section 2.4.3 Signal Actions, change:

1117 Any function or function-like macro not in the above table may be unsafe with respect to
1118 signals.

1119 to:

1120 In addition, the functions in **<stdatomic.h>** other than *atomic_init()* shall be async-signal-
1121 safe when the atomic arguments are lock-free, and the *atomic_is_lock_free()* function shall
1122 be async-signal-safe when called with an atomic argument.

1123 All other functions (including generic functions) and function-like macros may be unsafe
1124 with respect to signals.

1125 Ref 7.21.2 para 7,8

1126 On page 496 line 17228 section 2.5 Standard I/O Streams, add a new paragraph:

1127 Each stream shall have an associated lock that is used to prevent data races when multiple
1128 threads of execution access a stream, and to restrict the interleaving of stream operations
1129 performed by multiple threads. Only one thread can hold this lock at a time. The lock shall
1130 be reentrant: a single thread can hold the lock multiple times at a given time. All functions
1131 that read, write, position, or query the position of a stream, [CX]except those with names
1132 ending *_unlocked*[/CX], shall lock the stream [CX] as if by a call to *flockfile*()[/CX] before
1133 accessing it and release the lock [CX] as if by a call to *funlockfile*()[/CX] when the access is
1134 complete.

1135 Ref (none)

1136 On page 498 line 17312 section 2.5.2 Stream Orientation and Encoding Rules, change:

1137 For conformance to the ISO/IEC 9899: 1999 standard, the definition of a stream includes an
1138 “orientation”.

1139 to:

1140 The definition of a stream includes an “orientation”.

1141 Ref 7.26.5.8

1142 On page 508 line 17720 section 2.8.4 Process Scheduling, change:

1143 When a running thread issues the *sched_yield*() function

1144 to:

1145 When a running thread issues the *sched_yield*() or *thrd_yield*() function

1146 Ref 7.17.2.2 para 3, 7.22.2.2 para 3

1147 On page 513 line 17907,17916 section 2.9.1 Thread-Safety, add *atomic_init*() and *srand*() to the list
1148 of functions that need not be thread-safe.

1149 Ref 7.12.8.3, 7.22.4.8

1150 On page 513 line 17907-17927 section 2.9.1 Thread-Safety, delete the following from the list of
1151 functions that need not be thread-safe:

1152 *lgamma*(), *lgammaf*(), *lgammal*(), *system*()

1153 [Note to reviewers: deletion of *mblen*\(\), *mbtowc*\(\), and *wctomb*\(\) from this list is the subject of](#)
1154 [Mantis bug 708.](#)

1155 Ref 7.28.1 para 1

1156 On page 513 line 17928 section 2.9.1 Thread-Safety, change:

1157 The *ctermid*() and *tmpnam*() functions need not be thread-safe if passed a NULL argument.
1158 The *mbrlen*(), *mbrtowc*(), *mbsnrrowcs*(), *mbsrtowcs*(), *wcrtomb*(), *wcsnrrowcs*(), and
1159 *wcsrtombs*() functions need not be thread-safe if passed a NULL *ps* argument.

1160 to:

1161 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a null pointer
1162 argument. The *c16rtomb()*, *c32rtomb()*, *mbrlen()*, *mbrtoc16()*, *mbrtoc32()*, *mbrtowc()*,
1163 *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and *wcsrtombs()* functions need not
1164 be thread-safe if passed a null *ps* argument. The *lgamma()*, *lgammaf()*, and *lgammal()*
1165 functions shall be thread-safe [XSI]except that they need not avoid data races when storing a
1166 value in the *signgam* variable[XSI].

1167 Ref 7.1.4 para 5

1168 On page 513 line 17934 section 2.9.1 Thread-Safety, change:

1169 Implementations shall provide internal synchronization as necessary in order to satisfy this
1170 requirement.

1171 to:

1172 Some functions that are not required to be thread-safe are nevertheless required to avoid data
1173 races with either all or some other functions, as specified on their individual reference pages.

1174 Implementations shall provide internal synchronization as necessary in order to satisfy
1175 thread-safety requirements.

1176 Ref 7.26.5

1177 On page 513 line 17944 section 2.9.2 Thread IDs, change:

1178 The lifetime of a thread ID ends after the thread terminates if it was created with the
1179 *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or if *pthread_detach()* or
1180 *pthread_join()* has been called for that thread.

1181 to:

1182 The lifetime of a thread ID ends after the thread terminates if it was created using
1183 *pthread_create()* with the *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or
1184 if *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()* has been called for that
1185 thread.

1186 Ref 7.26.5

1187 On page 514 line 17950 section 2.9.2 Thread IDs, change:

1188 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1189 *pthread_detach()* or *pthread_join()*.

1190 to:

1191 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1192 *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()*.

1193 Ref 7.26.4

1194 On page 514 line 17956 section 2.9.3 Thread Mutexes, change:

1195 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

1196 to:

1197 A thread shall become the owner of a mutex, *m*, of type **pthread_mutex_t** when one of the
1198 following occurs:

1199 Ref 7.26.3, 7.26.4

1200 On page 514 line 17972 section 2.9.3 Thread Mutexes, add two new paragraphs and lists:

1201 A thread shall become the owner of a mutex, *m*, of type **mtx_t** when one of the following
1202 occurs:

- 1203 • It calls *mtx_lock()* with *m* as the *mtx* argument and the call returns `thrd_success`.
- 1204 • It calls *mtx_trylock()* with *m* as the *mtx* argument and the call returns
1205 `thrd_success`.
- 1206 • It calls *mtx_timedlock()* with *m* as the *mtx* argument and the call returns
1207 `thrd_success`.
- 1208 • It calls *cond_wait()* with *m* as the *mtx* argument and the call returns `thrd_success`.
- 1209 • It calls *cond_timedwait()* with *m* as the *mtx* argument and the call returns
1210 `thrd_success` or `thrd_timedout`.

1211 The thread shall remain the owner of *m* until one of the following occurs:

- 1212 • It executes *mtx_unlock()* with *m* as the *mtx* argument.
- 1213 • It blocks in a call to *cond_wait()* with *m* as the *mtx* argument.
- 1214 • It blocks in a call to *cond_timedwait()* with *m* as the *mtx* argument.

1215 Ref 7.26.4

1216 On page 514 line 17980 section 2.9.3 Thread Mutexes, change:

1217 Robust mutexes provide a means to enable the implementation to notify other threads in the
1218 event of a process terminating while one of its threads holds a mutex lock.

1219 to:

1220 Robust mutexes provide a means to enable the implementation to notify other threads in the
1221 event of a process terminating while one of its threads holds a lock on a mutex of type
1222 **pthread_mutex_t**.

1223 Ref 7.26.5

1224 On page 517 line 18085 section 2.9.5 Thread Cancellation, change:

1225 The thread cancellation mechanism allows a thread to terminate the execution of any other
1226 thread in the process in a controlled manner.

1227 to:

1228 The thread cancellation mechanism allows a thread to terminate the execution of any thread
1229 in the process, except for threads created using *thrd_create()*, in a controlled manner.

1230 Ref 7.26.3, 7.26.5.6

1231 On page 518 line 18119-18137 section 2.9.5.2 Cancellation Points, add the following to the list of

1232 functions that are required to be cancellation points:

1233 *cnd_timedwait()*, *cnd_wait()*, *thrd_join()*, *thrd_sleep()*

1234 Ref 7.26.5

1235 On page 520 line 18225 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1236 Each thread maintains a list of cancellation cleanup handlers.

1237 to:

1238 Each thread that was not created using *thrd_create()* maintains a list of cancellation cleanup
1239 handlers.

1240 Ref 7.26.6.1

1241 On page 521 line 18240 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1242 as described for *pthread_key_create()*

1243 to:

1244 as described for *pthread_key_create()* and *tss_create()*

1245 Ref 7.26

1246 On page 523 line 18337 section 2.9.9 Synchronization Object Copies and Alternative Mappings,
1247 add a new sentence:

1248 For ISO C functions declared in **<threads.h>**, the above requirements shall apply as if
1249 condition variables of type **cnd_t** and mutexes of type **mtx_t** have a process-shared attribute
1250 that is set to **PTHREAD_PROCESS_PRIVATE**.

1251 Ref 7.26.3

1252 On page 547 line 19279 section 2.12.1 Defined Types, change:

1253 **pthread_cond_t**

1254 to

1255 **pthread_cond_t, cnd_t**

1256 Ref 7.26.6, 7.26.4

1257 On page 547 line 19281 section 2.12.1 Defined Types, change:

1258 **pthread_key_t**

1259 **pthread_mutex_t**

1260 to

1261 **pthread_key_t, tss_t**

1262 **pthread_mutex_t, mtx_t**

1263 Ref 7.26.2.1

1264 On page 547 line 19284 section 2.12.1 Defined Types, change:

1265 **pthread_once_t**

1266 to

1267 **pthread_once_t, once_flag**

1268 Ref 7.26.5

1269 On page 547 line 19287 section 2.12.1 Defined Types, change:

1270 **pthread_t**

1271 to

1272 **pthread_t, thrd_t**

1273 Ref 7.3.9.3

1274 On page 552 line 19370 insert a new CMPLX() section:

1275 **NAME**

1276 CMPLX — make a complex value

1277 **SYNOPSIS**

1278 `#include <complex.h>`

1279 `double complex CMPLX(double x, double y);`

1280 `float complex CMPLXF(float x, float y);`

1281 `long double complex CMPLXL(long double x, long double y);`

1282 **DESCRIPTION**

1283 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1284 Any conflict between the requirements described here and the ISO C standard is
1285 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1286 The CMPLX macros shall expand to an expression of the specified complex type, with the
1287 real part having the (converted) value of x and the imaginary part having the (converted)
1288 value of y . The resulting expression shall be suitable for use as an initializer for an object
1289 with static or thread storage duration, provided both arguments are likewise suitable.

1290 **RETURN VALUE**

1291 The CMPLX macros return the complex value $x + iy$ (where i is the imaginary unit).

1292 These macros shall behave as if the implementation supported imaginary types and the
1293 definitions were:

1294 `#define CMPLX(x, y) ((double complex)((double)(x) + \`
1295 `_Imaginary_I * (double)(y)))`

1296 `#define CMPLXF(x, y) ((float complex)((float)(x) + \`
1297 `_Imaginary_I * (float)(y)))`

1298 `#define CMPLXL(x, y) ((long double complex)((long double)(x) + \`
1299 `_Imaginary_I * (long double)(y)))`

1300 **ERRORS**

1301 No errors are defined.

1302 **EXAMPLES**

1303 None.

1304 **APPLICATION USAGE**

1305 None.

1306 **RATIONALE**

1307 None.

1308 **FUTURE DIRECTIONS**

1309 None.

1310 **SEE ALSO**

1311 XBD <**complex.h**>

1312 **CHANGE HISTORY**

1313 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1314 Ref 7.22.4.5 para 1

1315 On page 553 line 19384 section `_Exit()`, change:

1316 `void _Exit(int status);`

1317 `#include <unistd.h>`

1318 `void _exit(int status);`

1319 to:

1320 `_Noreturn void _Exit(int status);`

1321 `#include <unistd.h>`

1322 `_Noreturn void _exit(int status);`

1323 Ref 7.22.4.5 para 2

1324 On page 553 line 19396 section `_Exit()`, change:

1325 shall not call functions registered with `atexit()` nor any registered signal handlers

1326 to:

1327 shall not call functions registered with `atexit()` nor `at_quick_exit()`, nor any registered signal
1328 handlers

1329 Ref (none)

1330 On page 557 line 19562 section `_Exit()`, change:

1331 The ISO/IEC 9899: 1999 standard adds the `_Exit()` function

1332 to:

1333 The ISO/IEC 9899: 1999 standard added the `_Exit()` function

1334 Ref 7.22.4.3, 7.22.4.7

1335 On page 557 line 19568 section `_Exit()`, add `at_quick_exit` and `quick_exit` to the SEE ALSO section.

1336 Ref 7.22.4.1 para 1

1337 On page 565 line 19761 section `abort()`, change:

1338 `void abort(void);`

1339 to:

1340 `_Noreturn void abort(void);`

1341 Ref (none)

1342 On page 565 line 19785 section `abort()`, change:

1343 The ISO/IEC 9899: 1999 standard requires the `abort()` function to be async-signal-safe.

1344 to:

1345 The ISO/IEC 9899: 1999 standard required (and the current standard still requires) the
1346 `abort()` function to be async-signal-safe.

1347 Ref 7.22.3.1

1348 On page 597 line 20771 insert the following new `aligned_alloc()` section:

1349 **NAME**

1350 `aligned_alloc` — allocate memory with a specified alignment

1351 **SYNOPSIS**

1352 `#include <stdlib.h>`

1353 `void *aligned_alloc(size_t alignment, size_t size);`

1354 **DESCRIPTION**

1355 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1356 Any conflict between the requirements described here and the ISO C standard is
1357 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1358 The `aligned_alloc()` function shall allocate unused space for an object whose alignment is
1359 specified by `alignment`, whose size in bytes is specified by `size` and whose value is
1360 indeterminate.

1361 The order and contiguity of storage allocated by successive calls to `aligned_alloc()` is
1362 unspecified. Each such allocation shall yield a pointer to an object disjoint from any other
1363 object. The pointer returned shall point to the start (lowest byte address) of the allocated
1364 space. If the value of `alignment` is not a valid alignment supported by the implementation, a
1365 null pointer shall be returned. If the space cannot be allocated, a null pointer shall be
1366 returned. If the size of the space requested is 0, the behavior is implementation-defined:
1367 either a null pointer shall be returned to indicate an error, or the behavior shall be as if the

1368 size were some non-zero value, except that the behavior is undefined if the returned pointer
1369 is used to access an object.

1370 For purposes of determining the existence of a data race, *aligned_alloc()* shall behave as
1371 though it accessed only memory locations accessible through its arguments and not other
1372 static duration storage. The function may, however, visibly modify the storage that it
1373 allocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*,
1374 [ADV]*posix_memalign()*,[/ADV] [CX]*reallocarray()*,[/CX] and *realloc()* that allocate or
1375 deallocate a particular region of memory shall occur in a single total order (see [xref to XBD
1376 4.12.1]), and each such deallocation call shall synchronize with the next allocation (if any)
1377 in this order.

1378 **RETURN VALUE**

1379 Upon successful completion, *aligned_alloc()* shall return a pointer to the allocated space; if
1380 *size* is 0, the application shall ensure that the pointer is not used to access an object.

1381 Otherwise, it shall return a null pointer [CX]and set *errno* to indicate the error[/CX].

1382 **ERRORS**

1383 The *aligned_alloc()* function shall fail if:

1384 [CX][EINVAL] The value of *alignment* is not a valid alignment supported by the
1385 implementation.

1386 [ENOMEM] Insufficient storage space is available.[/CX]

1387 The *aligned_alloc()* function may fail if:

1388 [CX][EINVAL] *size* is 0 and the implementation does not support 0 sized allocations.[/
1389 CX]

1390 **EXAMPLES**

1391 None.

1392 **APPLICATION USAGE**

1393 None.

1394 **RATIONALE**

1395 See the RATIONALE for [xref to *malloc()*].

1396 **FUTURE DIRECTIONS**

1397 None.

1398 **SEE ALSO**

1399 *calloc*, *free*, *getrlimit*, *malloc*, *posix_memalign*, *realloc*

1400 XBD <stdlib.h>

1401 **CHANGE HISTORY**

1402 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1403 Ref 7.27.3, 7.1.4 para 5

1404 On page 600 line 20911 section `asctime()`, change:

1405 [CX]The `asctime()` function need not be thread-safe.[/CX]

1406 to:

1407 The `asctime()` function need not be thread-safe; however, `asctime()` shall avoid data races
1408 with all functions other than itself, `ctime()`, `gmtime()` and `localtime()`.

1409 Ref 7.22.4.3

1410 On page 618 line 21380 insert the following new `at_quick_exit()` section:

1411 **NAME**

1412 `at_quick_exit` — register a function to be called from `quick_exit()`

1413 **SYNOPSIS**

1414 `#include <stdlib.h>`

1415 `int at_quick_exit(void (*func)(void));`

1416 **DESCRIPTION**

1417 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1418 Any conflict between the requirements described here and the ISO C standard is
1419 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1420 The `at_quick_exit()` function shall register the function pointed to by `func`, to be called
1421 without arguments should `quick_exit()` be called. It is unspecified whether a call to the
1422 `at_quick_exit()` function that does not happen before the `quick_exit()` function is called will
1423 succeed.

1424 At least 32 functions can be registered with `at_quick_exit()`.

1425 [CX]After a successful call to any of the `exec` functions, any functions previously registered
1426 by `at_quick_exit()` shall no longer be registered.[/CX]

1427 **RETURN VALUE**

1428 Upon successful completion, `at_quick_exit()` shall return 0; otherwise, it shall return a non-
1429 zero value.

1430 **ERRORS**

1431 No errors are defined.

1432 **EXAMPLES**

1433 None.

1434 **APPLICATION USAGE**

1435 The `at_quick_exit()` function registrations are distinct from the `atexit()` registrations, so
1436 applications might need to call both registration functions with the same argument.

1437 The functions registered by a call to `at_quick_exit()` must return to ensure that all registered
1438 functions are called.

1439 The application should call `sysconf()` to obtain the value of `{ATEXIT_MAX}`, the number of
1440 functions that can be registered. There is no way for an application to tell how many
1441 functions have already been registered with `at_quick_exit()`.

1442 Since the behavior is undefined if the `quick_exit()` function is called more than once,
1443 portable applications calling `at_quick_exit()` must ensure that the `quick_exit()` function is not
1444 called when the functions registered by the `at_quick_exit()` function are called.

1445 If a function registered by the `at_quick_exit()` function is called and a portable application
1446 needs to stop further `quick_exit()` processing, it must call the `_exit()` function or the `_Exit()`
1447 function or one of the functions which cause abnormal process termination.

1448 **RATIONALE**

1449 None.

1450 **FUTURE DIRECTIONS**

1451 None.

1452 **SEE ALSO**

1453 `atexit`, `exec`, `exit`, `quick_exit`, `sysconf`

1454 XBD `<stdlib.h>`

1455 **CHANGE HISTORY**

1456 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1457 Ref 7.22.4.3

1458 On page 618 line 21381 section `atexit()`, change:

1459 `atexit` — register a function to run at process termination

1460 to:

1461 `atexit` — register a function to be called from `exit()` or after return from `main()`

1462 Ref 7.22.4.2 para 2, 7.22.4.3

1463 On page 618 line 21389 section `atexit()`, change:

1464 The `atexit()` function shall register the function pointed to by `func`, to be called without
1465 arguments at normal program termination. At normal program termination, all functions
1466 registered by the `atexit()` function shall be called, in the reverse order of their registration,
1467 except that a function is called after any previously registered functions that had already
1468 been called at the time it was registered. Normal termination occurs either by a call to `exit()`
1469 or a return from `main()`.

1470 to:

1471 The `atexit()` function shall register the function pointed to by `func`, to be called without
1472 arguments from `exit()`, or after return from the initial call to `main()`, or on the last thread
1473 termination. If the `exit()` function is called, it is unspecified whether a call to the `atexit()`
1474 function that does not happen before `exit()` is called will succeed.

1475 [Note to reviewers: the part about all registered functions being called in reverse order is duplicated](#)
1476 [on the exit\(\) page and is not needed here.](#)

1477 Ref 7.22.4.2 para 2

1478 On page 618 line 21405 section `atexit()`, insert a new first APPLICATION USAGE paragraph:

1479 The `atexit()` function registrations are distinct from the `at_quick_exit()` registrations, so
1480 applications might need to call both registration functions with the same argument.

1481 Ref 7.22.4.3

1482 On page 618 line 21410 section `atexit()`, change:

1483 Since the behavior is undefined if the `exit()` function is called more than once, portable
1484 applications calling `atexit()` must ensure that the `exit()` function is not called at normal
1485 process termination when all functions registered by the `atexit()` function are called.

1486 All functions registered by the `atexit()` function are called at normal process termination,
1487 which occurs by a call to the `exit()` function or a return from `main()` or on the last thread
1488 termination, when the behavior is as if the implementation called `exit()` with a zero argument
1489 at thread termination time.

1490 If, at normal process termination, a function registered by the `atexit()` function is called and a
1491 portable application needs to stop further `exit()` processing, it must call the `_exit()` function
1492 or the `_Exit()` function or one of the functions which cause abnormal process termination.

1493 to:

1494 Since the behavior is undefined if the `exit()` function is called more than once, portable
1495 applications calling `atexit()` must ensure that the `exit()` function is not called when the
1496 functions registered by the `atexit()` function are called.

1497 If a function registered by the `atexit()` function is called and a portable application needs to
1498 stop further `exit()` processing, it must call the `_exit()` function or the `_Exit()` function or one
1499 of the functions which cause abnormal process termination.

1500 Ref 7.22.4.3

1501 On page 619 line 21425 section `atexit()`, add `at_quick_exit` to the SEE ALSO section.

1502 Ref 7.16

1503 On page 624 line 21548 insert the following new `atomic_*` sections:

1504 **NAME**

1505 `atomic_compare_exchange_strong`, `atomic_compare_exchange_strong_explicit`,
1506 `atomic_compare_exchange_weak`, `atomic_compare_exchange_weak_explicit` — atomically
1507 compare and exchange the values of two objects

1508 **SYNOPSIS**

```
1509 #include <stdatomic.h>
1510 _Bool atomic_compare_exchange_strong(volatile A *object,
1511     C *expected, C desired);
1512 _Bool atomic_compare_exchange_strong_explicit(volatile A *object,
1513     C *expected, C desired, memory_order success,
```

```
1514     memory_order failure);
1515     _Bool atomic_compare_exchange_weak(volatile A *object,
1516     C *expected, C desired);
1517     _Bool atomic_compare_exchange_weak_explicit(volatile A *object,
1518     C *expected, C desired, memory_order success,
1519     memory_order failure);
```

1520 DESCRIPTION

1521 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1522 Any conflict between the requirements described here and the ISO C standard is
1523 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1524 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1525 `<stdatomic.h>` header nor support these generic functions.

1526 The `atomic_compare_exchange_strong_explicit()` generic function shall atomically compare
1527 the contents of the memory pointed to by *object* for equality with that pointed to by
1528 *expected*, and if true, shall replace the contents of the memory pointed to by *object*
1529 with *desired*, and if false, shall update the contents of the memory pointed to by *expected*
1530 with that pointed to by *object*. This operation shall be an atomic read-modify-write operation
1531 (see [xref to XBD 4.12.1]). If the comparison is true, memory shall be affected according to
1532 the value of *success*, and if the comparison is false, memory shall be affected according to
1533 the value of *failure*. The application shall ensure that *failure* is not
1534 `memory_order_release` nor `memory_order_acq_rel`, and shall ensure that *failure* is
1535 no stronger than *success*.

1536 The `atomic_compare_exchange_strong()` generic function shall be equivalent to
1537 `atomic_compare_exchange_strong_explicit()` called with *success* and *failure* both set to
1538 `memory_order_seq_cst`.

1539 The `atomic_compare_exchange_weak_explicit()` generic function shall be equivalent to
1540 `atomic_compare_exchange_strong_explicit()`, except that the compare-and-exchange
1541 operation may fail spuriously. That is, even when the contents of memory referred to by
1542 *expected* and *object* are equal, it may return zero and store back to *expected* the same
1543 memory contents that were originally there.

1544 The `atomic_compare_exchange_weak()` generic function shall be equivalent to
1545 `atomic_compare_exchange_weak_explicit()` called with *success* and *failure* both set to
1546 `memory_order_seq_cst`.

1547 RETURN VALUE

1548 These generic functions shall return the result of the comparison.

1549 ERRORS

1550 No errors are defined.

1551 EXAMPLES

1552 None.

1553 APPLICATION USAGE

1554 A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will
1555 be in a loop. For example:

```
1556     exp = atomic_load(&cur);
1557     do {
1558         des = function(exp);
1559     } while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

1560 When a compare-and-exchange is in a loop, the weak version will yield better performance
1561 on some platforms. When a weak compare-and-exchange would require a loop and a strong
1562 one would not, the strong one is preferable.

1563 **RATIONALE**

1564 None.

1565 **FUTURE DIRECTIONS**

1566 None.

1567 **SEE ALSO**

1568 XBD Section 4.12.1, <**stdatomic.h**>

1569 **CHANGE HISTORY**

1570 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1571 **NAME**

1572 `atomic_exchange`, `atomic_exchange_explicit` — atomically exchange the value of an object

1573 **SYNOPSIS**

```
1574     #include <stdatomic.h>
1575     C atomic_exchange(volatile A *object, C desired);
1576     C atomic_exchange_explicit(volatile A *object,
1577                               C desired, memory_order order);
```

1578 **DESCRIPTION**

1579 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1580 Any conflict between the requirements described here and the ISO C standard is
1581 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1582 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1583 <**stdatomic.h**> header nor support these generic functions.

1584 The `atomic_exchange_explicit()` generic function shall atomically replace the value pointed
1585 to by *object* with *desired*. This operation shall be an atomic read-modify-write operation (see
1586 [xref to XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1587 The `atomic_exchange()` generic function shall be equivalent to `atomic_exchange_explicit()`
1588 called with *order* set to `memory_order_seq_cst`.

1589 **RETURN VALUE**

1590 These generic functions shall return the value pointed to by *object* immediately before the
1591 effects.

1592 **ERRORS**

1593 No errors are defined.

1594 **EXAMPLES**

1595 None.

1596 **APPLICATION USAGE**

1597 None.

1598 **RATIONALE**

1599 None.

1600 **FUTURE DIRECTIONS**

1601 None.

1602 **SEE ALSO**

1603 XBD Section 4.12.1, <**stdatomic.h**>

1604 **CHANGE HISTORY**

1605 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1606 **NAME**

1607 `atomic_fetch_add`, `atomic_fetch_add_explicit`, `atomic_fetch_and`,
1608 `atomic_fetch_and_explicit`, `atomic_fetch_or`, `atomic_fetch_or_explicit`, `atomic_fetch_sub`,
1609 `atomic_fetch_sub_explicit`, `atomic_fetch_xor`, `atomic_fetch_xor_explicit` — atomically
1610 replace the value of an object with the result of a computation

1611 **SYNOPSIS**

```
1612 #include <stdatomic.h>
1613 C atomic_fetch_add(volatile A *object, M operand);
1614 C atomic_fetch_add_explicit(volatile A *object, M operand,
1615 memory_order order);
1616 C atomic_fetch_and(volatile A *object, M operand);
1617 C atomic_fetch_and_explicit(volatile A *object, M operand,
1618 memory_order order);
1619 C atomic_fetch_or(volatile A *object, M operand);
1620 C atomic_fetch_or_explicit(volatile A *object, M operand,
1621 memory_order order);
1622 C atomic_fetch_sub(volatile A *object, M operand);
1623 C atomic_fetch_sub_explicit(volatile A *object, M operand,
1624 memory_order order);
1625 C atomic_fetch_xor(volatile A *object, M operand);
1626 C atomic_fetch_xor_explicit(volatile A *object, M operand,
1627 memory_order order);
```

1628 **DESCRIPTION**

1629 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1630 Any conflict between the requirements described here and the ISO C standard is
1631 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1632 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1633 `<stdatomic.h>` header nor support these generic functions.

1634 The `atomic_fetch_add_explicit()` generic function shall atomically replace the value pointed
1635 to by `object` with the result of adding `operand` to this value. This operation shall be an
1636 atomic read-modify-write operation (see [xref to XBD 4.12.1]). Memory shall be affected

1637 according to the value of *order*.

1638 The *atomic_fetch_add()* generic function shall be equivalent to *atomic_fetch_add_explicit()*
1639 called with *order* set to *memory_order_seq_cst*.

1640 The other *atomic_fetch_**() generic functions shall be equivalent to
1641 *atomic_fetch_add_explicit()* if their name ends with *explicit*, or to *atomic_fetch_add()* if it
1642 does not, respectively, except that they perform the computation indicated in their name,
1643 instead of addition:

1644 *sub* subtraction
1645 *or* bitwise inclusive OR
1646 *xor* bitwise exclusive OR
1647 *and* bitwise AND

1648 For addition and subtraction, the application shall ensure that **A** is an atomic integer type or
1649 an atomic pointer type and is not **atomic_bool**. For the other operations, the application
1650 shall ensure that **A** is an atomic integer type and is not **atomic_bool**.

1651 For signed integer types, the computation shall silently wrap around on overflow; there are
1652 no undefined results. For pointer types, the result can be an undefined address, but the
1653 computations otherwise have no undefined behavior.

1654 **RETURN VALUE**

1655 These generic functions shall return the value pointed to by *object* immediately before the
1656 effects.

1657 **ERRORS**

1658 No errors are defined.

1659 **EXAMPLES**

1660 None.

1661 **APPLICATION USAGE**

1662 The operation of these generic functions is nearly equivalent to the operation of the
1663 corresponding compound assignment operators +=, -=, etc. The only differences are that the
1664 compound assignment operators are not guaranteed to operate atomically, and the value
1665 yielded by a compound assignment operator is the updated value of the object, whereas the
1666 value returned by these generic functions is the previous value of the atomic object.

1667 **RATIONALE**

1668 None.

1669 **FUTURE DIRECTIONS**

1670 None.

1671 **SEE ALSO**

1672 XBD Section 4.12.1, <**stdatomic.h**>

1673 **CHANGE HISTORY**

1674 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1675 **NAME**

1676 atomic_flag_clear, atomic_flag_clear_explicit — clear an atomic flag

1677 **SYNOPSIS**

```
1678 #include <stdatomic.h>
1679 void atomic_flag_clear(volatile atomic_flag *object);
1680 void atomic_flag_clear_explicit(
1681     volatile atomic_flag *object, memory_order order);
```

1682 **DESCRIPTION**

1683 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1684 Any conflict between the requirements described here and the ISO C standard is
1685 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[CX]

1686 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1687 `<stdatomic.h>` header nor support these functions.

1688 The `atomic_flag_clear_explicit()` function shall atomically place the atomic flag pointed to
1689 by `object` into the clear state. Memory shall be affected according to the value of `order`,
1690 which the application shall ensure is not `memory_order_acquire` nor
1691 `memory_order_acq_rel`.

1692 The `atomic_flag_clear()` function shall be equivalent to `atomic_flag_clear_explicit()` called
1693 with `order` set to `memory_order_seq_cst`.

1694 **RETURN VALUE**

1695 These functions shall not return a value.

1696 **ERRORS**

1697 No errors are defined.

1698 **EXAMPLES**

1699 None.

1700 **APPLICATION USAGE**

1701 None.

1702 **RATIONALE**

1703 None.

1704 **FUTURE DIRECTIONS**

1705 None.

1706 **SEE ALSO**

1707 XBD Section 4.12.1, `<stdatomic.h>`

1708 **CHANGE HISTORY**

1709 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1710 **NAME**

1711 atomic_flag_test_and_set, atomic_flag_test_and_set_explicit — test and set an atomic flag

1712 **SYNOPSIS**

```
1713     #include <stdatomic.h>
1714     _Bool atomic_flag_test_and_set(volatile atomic_flag *object);
1715     _Bool atomic_flag_test_and_set_explicit(
1716         volatile atomic_flag *object, memory_order order);
```

1717 **DESCRIPTION**

1718 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1719 Any conflict between the requirements described here and the ISO C standard is
1720 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1721 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1722 `<stdatomic.h>` header nor support these functions.

1723 The `atomic_flag_test_and_set_explicit()` function shall atomically place the atomic flag
1724 pointed to by `object` into the set state and return the value corresponding to the immediately
1725 preceding state. This operation shall be an atomic read-modify-write operation (see [xref to
1726 XBD 4.12.1]). Memory shall be affected according to the value of `order`.

1727 The `atomic_flag_test_and_set()` function shall be equivalent to
1728 `atomic_flag_test_and_set_explicit()` called with `order` set to `memory_order_seq_cst`.

1729 **RETURN VALUE**

1730 These functions shall return the value that corresponds to the state of the atomic flag
1731 immediately before the effects. The return value true shall correspond to the set state and the
1732 return value false shall correspond to the clear state.

1733 **ERRORS**

1734 No errors are defined.

1735 **EXAMPLES**

1736 None.

1737 **APPLICATION USAGE**

1738 None.

1739 **RATIONALE**

1740 None.

1741 **FUTURE DIRECTIONS**

1742 None.

1743 **SEE ALSO**

1744 XBD Section 4.12.1, `<stdatomic.h>`

1745 **CHANGE HISTORY**

1746 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1747 **NAME**

1748 `atomic_init` — initialize an atomic object

1749 **SYNOPSIS**

```
1750     #include <stdatomic.h>  
1751     void atomic_init(volatile A *obj, C value);
```

1752 **DESCRIPTION**

1753 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1754 Any conflict between the requirements described here and the ISO C standard is
1755 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1756 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1757 `<stdatomic.h>` header nor support this generic function.

1758 The `atomic_init()` generic function shall initialize the atomic object pointed to by `obj` to the
1759 value `value`, while also initializing any additional state that the implementation might need
1760 to carry for the atomic object.

1761 Although this function initializes an atomic object, it does not avoid data races; concurrent
1762 access to the variable being initialized, even via an atomic operation, constitutes a data race.

1763 **RETURN VALUE**

1764 The `atomic_init()` generic function shall not return a value.

1765 **ERRORS**

1766 No errors are defined.

1767 **EXAMPLES**

```
1768     atomic_int guide;  
1769     atomic_init(&guide, 42);
```

1770 **APPLICATION USAGE**

1771 None.

1772 **RATIONALE**

1773 None.

1774 **FUTURE DIRECTIONS**

1775 None.

1776 **SEE ALSO**

1777 XBD `<stdatomic.h>`

1778 **CHANGE HISTORY**

1779 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1780 **NAME**

1781 `atomic_is_lock_free` — indicate whether or not atomic operations are lock-free

1782 **SYNOPSIS**

```
1783     #include <stdatomic.h>  
1784     _Bool atomic_is_lock_free(const volatile A *obj);
```

1785 **DESCRIPTION**

1786 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1787 Any conflict between the requirements described here and the ISO C standard is
1788 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1789 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1790 `<stdatomic.h>` header nor support this generic function.

1791 The `atomic_is_lock_free()` generic function shall indicate whether or not atomic operations
1792 on objects of the type pointed to by `obj` are lock-free; `obj` can be a null pointer.

1793 RETURN VALUE

1794 The `atomic_is_lock_free()` generic function shall return a non-zero value if and only if
1795 atomic operations on objects of the type pointed to by `obj` are lock-free. During the lifetime
1796 of the calling process, the result of the lock-free query shall be consistent for all pointers of
1797 the same type.

1798 ERRORS

1799 No errors are defined.

1800 EXAMPLES

1801 None.

1802 APPLICATION USAGE

1803 None.

1804 RATIONALE

1805 Operations that are lock-free should also be address-free. That is, atomic operations on the
1806 same memory location via two different addresses will communicate atomically. The
1807 implementation should not depend on any per-process state. This restriction enables
1808 communication via memory mapped into a process more than once and memory shared
1809 between two processes.

1810 FUTURE DIRECTIONS

1811 None.

1812 SEE ALSO

1813 XBD `<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_load`, `atomic_load_explicit` — atomically obtain the value of an object

1818 SYNOPSIS

```
1819 #include <stdatomic.h>  
1820 C atomic_load(const volatile A *object);  
1821 C atomic_load_explicit(const volatile A *object,  
1822     memory_order order);
```

1823 DESCRIPTION

1824 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1825 Any conflict between the requirements described here and the ISO C standard is
1826 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1827 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1828 `<stdatomic.h>` header nor support these generic functions.

1829 The `atomic_load_explicit()` generic function shall atomically obtain the value pointed to by
1830 *object*. Memory shall be affected according to the value of *order*, which the application shall
1831 ensure is not `memory_order_release` nor `memory_order_acq_rel`.

1832 The `atomic_load()` generic function shall be equivalent to `atomic_load_explicit()` called with
1833 *order* set to `memory_order_seq_cst`.

1834 **RETURN VALUE**
1835 These generic functions shall return the value pointed to by *object*.

1836 **ERRORS**
1837 No errors are defined.

1838 **EXAMPLES**
1839 None.

1840 **APPLICATION USAGE**
1841 None.

1842 **RATIONALE**
1843 None.

1844 **FUTURE DIRECTIONS**
1845 None.

1846 **SEE ALSO**
1847 XBD Section 4.12.1, `<stdatomic.h>`

1848 **CHANGE HISTORY**
1849 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1850 **NAME**
1851 `atomic_signal_fence`, `atomic_thread_fence` — fence operations

1852 **SYNOPSIS**
1853 `#include <stdatomic.h>`
1854 `void atomic_signal_fence(memory_order order);`
1855 `void atomic_thread_fence(memory_order order);`

1856 **DESCRIPTION**
1857 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1858 Any conflict between the requirements described here and the ISO C standard is
1859 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1860 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1861 `<stdatomic.h>` header nor support these functions.

1862 The *atomic_signal_fence()* and *atomic_thread_fence()* functions provide synchronization
1863 primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A
1864 fence with acquire semantics is called an *acquire fence*; a fence with release semantics is
1865 called a *release fence*.

1866 A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X*
1867 and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X*
1868 modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* or a value written
1869 by any side effect in the hypothetical release sequence *X* would head if it were a release
1870 operation.

1871 A release fence *A* synchronizes with an atomic operation *B* that performs an acquire
1872 operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is
1873 sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by
1874 any side effect in the hypothetical release sequence *X* would head if it were a release
1875 operation.

1876 An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with
1877 an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced
1878 before *B* and reads the value written by *A* or a value written by any side effect in the release
1879 sequence headed by *A*.

1880 Depending on the value of *order*, the operation performed by *atomic_thread_fence()* shall:

- 1881 • have no effects, if *order* is equal to `memory_order_relaxed`;
- 1882 • be an acquire fence, if *order* is equal to `memory_order_acquire` or
1883 `memory_order_consume`;
- 1884 • be a release fence, if *order* is equal to `memory_order_release`;
- 1885 • be both an acquire fence and a release fence, if *order* is equal to
1886 `memory_order_acq_rel`;
- 1887 • be a sequentially consistent acquire and release fence, if *order* is equal to
1888 `memory_order_seq_cst`.

1889 The *atomic_signal_fence()* function shall be equivalent to *atomic_thread_fence()*, except
1890 that the resulting ordering constraints shall be established only between a thread and a signal
1891 handler executed in the same thread.

1892 RETURN VALUE

1893 These functions shall not return a value.

1894 ERRORS

1895 No errors are defined.

1896 EXAMPLES

1897 None.

1898 APPLICATION USAGE

1899 The *atomic_signal_fence()* function can be used to specify the order in which actions
1900 performed by the thread become visible to the signal handler. Implementation reorderings of
1901 loads and stores are inhibited in the same way as with *atomic_thread_fence()*, but the
1902 hardware fence instructions that *atomic_thread_fence()* would have inserted are not
1903 emitted.

1904 **RATIONALE**

1905 None.

1906 **FUTURE DIRECTIONS**

1907 None.

1908 **SEE ALSO**

1909 XBD Section 4.12.1, <**stdatomic.h**>

1910 **CHANGE HISTORY**

1911 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1912 **NAME**

1913 *atomic_store*, *atomic_store_explicit* — atomically store a value in an object

1914 **SYNOPSIS**

```
1915 #include <stdatomic.h>  
1916 void atomic_store(volatile A *object, C desired);  
1917 void atomic_store_explicit(volatile A *object, C desired,  
1918 memory_order order);
```

1919 **DESCRIPTION**

1920 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1921 Any conflict between the requirements described here and the ISO C standard is
1922 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1923 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1924 `<stdatomic.h>` header nor support these generic functions.

1925 The *atomic_store_explicit()* generic function shall atomically replace the value pointed to by
1926 *object* with the value of *desired*. Memory shall be affected according to the value of *order*,
1927 which the application shall ensure is not `memory_order_acquire`,
1928 `memory_order_consume`, nor `memory_order_acq_rel`.

1929 The *atomic_store()* generic function shall be equivalent to *atomic_store_explicit()* called
1930 with *order* set to `memory_order_seq_cst`.

1931 **RETURN VALUE**

1932 These generic functions shall not return a value.

1933 **ERRORS**

1934 No errors are defined.

1935 **EXAMPLES**

1936 None.

1937 **APPLICATION USAGE**

1938 None.

1939 **RATIONALE**

1940 None.

1941 **FUTURE DIRECTIONS**

1942 None.

1943 **SEE ALSO**

1944 XBD Section 4.12.1, <**stdatomic.h**>

1945 **CHANGE HISTORY**

1946 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1947 Ref 7.28.1, 7.1.4 para 5

1948 On page 633 line 21891 insert a new `c16rtomb()` section:

1949 **NAME**

1950 `c16rtomb`, `c32rtomb` — convert a Unicode character code to a character (restartable)

1951 **SYNOPSIS**

1952 `#include <uchar.h>`

1953 `size_t c16rtomb(char *restrict s, char16_t c16,`

1954 `mbstate_t *restrict ps);`

1955 `size_t c32rtomb(char *restrict s, char32_t c32,`

1956 `mbstate_t *restrict ps);`

1957 **DESCRIPTION**

1958 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1959 Any conflict between the requirements described here and the ISO C standard is

1960 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1961 If *s* is a null pointer, the `c16rtomb()` function shall be equivalent to the call:

1962 `c16rtomb(buf, L'\0', ps)`

1963 where *buf* is an internal buffer.

1964 If *s* is not a null pointer, the `c16rtomb()` function shall determine the number of bytes needed

1965 to represent the character that corresponds to the wide character given by *c16* (including any

1966 shift sequences), and store the resulting bytes in the array whose first element is pointed to

1967 by *s*. At most {MB_CUR_MAX} bytes shall be stored. If *c16* is a null wide character, a null

1968 byte shall be stored, preceded by any shift sequence needed to restore the initial shift state;

1969 the resulting state described shall be the initial conversion state.

1970 If *ps* is a null pointer, the `c16rtomb()` function shall use its own internal **mbstate_t** object,

1971 which shall be initialized at program start-up to the initial conversion state. Otherwise, the

1972 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current

1973 conversion state of the associated character sequence.

1974 The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

- 1975 The *mbrtoc16()* function shall not change the setting of *errno* if successful.
- 1976 The *c32rtomb()* function shall behave the same way as *c16rtomb()* except that the second
1977 parameter shall be an object of type **char32_t** instead of **char16_t**. References to *c16* in the
1978 above description shall apply as if they were *c32* when they are being read as describing
1979 *c32rtomb()*.
- 1980 If called with a null *ps* argument, the *c16rtomb()* function need not be thread-safe; however,
1981 such calls shall avoid data races with calls to *c16rtomb()* with a non-null argument and with
1982 calls to all other functions.
- 1983 If called with a null *ps* argument, the *c32rtomb()* function need not be thread-safe; however,
1984 such calls shall avoid data races with calls to *c32rtomb()* with a non-null argument and with
1985 calls to all other functions.
- 1986 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
1987 calls *c16rtomb()* or *c32rtomb()* with a null pointer for *ps*.
- 1988 **RETURN VALUE**
- 1989 These functions shall return the number of bytes stored in the array object (including any
1990 shift sequences). When *c16* or *c32* is not a valid wide character, an encoding error shall
1991 occur. In this case, the function shall store the value of the macro [EILSEQ] in *errno* and
1992 shall return (**size_t**)-1; the conversion state is unspecified.
- 1993 **ERRORS**
- 1994 These function shall fail if:
- 1995 [EILSEQ] An invalid wide-character code is detected.
- 1996 These functions may fail if:
- 1997 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]
- 1998 **EXAMPLES**
- 1999 None.
- 2000 **APPLICATION USAGE**
- 2001 None.
- 2002 **RATIONALE**
- 2003 None.
- 2004 **FUTURE DIRECTIONS**
- 2005 None.
- 2006 **SEE ALSO**
- 2007 *mbrtoc16*
- 2008 XBD <**uchar.h**>
- 2009 **CHANGE HISTORY**
- 2010 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2011 Ref G.6 para 6, F.10.4.3, F.10.4.2, F.10 para 11
2012 On page 633 line 21905 section `cabs()`, add:

2013 [MXC]`cabs(x + iy)`, `cabs(y + ix)`, and `cabs(x - iy)` shall return exactly the same value.

2014 If z is $\pm 0 \pm i0$, $+0$ shall be returned.

2015 If the real or imaginary part of z is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned, even if the other part is NaN .

2016 If the real or imaginary part of z is NaN and the other part is not $\pm\text{Inf}$, NaN shall be returned.
2017 [/MXC]

2018 Ref G.6.1.1
2019 On page 634 line 21935 section `cacos()`, add:

2020 [MXC]`cacos(conj(z))`, `cacosf(conjf(z))` and `cacosl(conjl(z))` shall return exactly the same
2021 value as `conj(cacos(z))`, `conjf(cacosf(z))` and `conjl(cacosl(z))`, respectively, including for the
2022 special values of z below.

2023 If z is $\pm 0 + i0$, $\pi/2 - i0$ shall be returned.

2024 If z is $\pm 0 + i\text{NaN}$, $\pi/2 + i\text{NaN}$ shall be returned.

2025 If z is $x + i\text{Inf}$ where x is finite, $\pi/2 - i\text{Inf}$ shall be returned.

2026 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
2027 floating-point exception may be raised.

2028 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $\pi - i\text{Inf}$ shall be returned.

2029 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 - i\text{Inf}$ shall be returned.

2030 If z is $-\text{Inf} + i\text{Inf}$, $3\pi/4 - i\text{Inf}$ shall be returned.

2031 If z is $+\text{Inf} + i\text{Inf}$, $\pi/4 - i\text{Inf}$ shall be returned.

2032 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
2033 is unspecified.

2034 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2035 point exception may be raised.

2036 If z is $\text{NaN} + i\text{Inf}$, $\text{NaN} - i\text{Inf}$ shall be returned.

2037 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} - i\text{NaN}$ shall be returned. [/MXC]

2038 Ref G.6.2.1
2039 On page 635 line 21966 section `cacosh()`, add:

2040 [MXC]`cacosh(conj(z))`, `cacoshf(conjf(z))` and `cacoshl(conjl(z))` shall return exactly the same
2041 value as `conj(cacosh(z))`, `conjf(cacoshf(z))` and `conjl(cacoshl(z))`, respectively, including for
2042 the special values of z below.

2043 If z is $\pm 0 + i0$, $+0 + i\pi/2$ shall be returned.

2044 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2045 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
2046 unspecified.

2047 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
2048 floating-point exception may be raised.

2049 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2050 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2051 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

2052 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2053 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2054 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2055 point exception may be raised.

2056 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2057 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2058 Ref 7.26.2.1

2059 On page 637 line 21989 insert the following new `call_once()` section:

2060 **NAME**

2061 `call_once` — dynamic package initialization

2062 **SYNOPSIS**

2063 `#include <threads.h>`

2064 `void call_once(once_flag *flag, void (*init_routine)(void));`
2065 `once_flag flag = ONCE_FLAG_INIT;`

2066 **DESCRIPTION**

2067 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2068 Any conflict between the requirements described here and the ISO C standard is
2069 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2070 The `call_once()` function shall use the **once_flag** pointed to by `flag` to ensure that
2071 `init_routine` is called exactly once, the first time the `call_once()` function is called with that
2072 value of `flag`. Completion of an effective call to the `call_once()` function shall synchronize
2073 with all subsequent calls to the `call_once()` function with the same value of `flag`.

2074 [CX]The `call_once()` function is not a cancellation point. However, if `init_routine` is a
2075 cancellation point and is canceled, the effect on `flag` shall be as if `call_once()` was never
2076 called.

2077 If the call to *init_routine* is terminated by a call to *longjmp()* or *siglongjmp()*, the behavior is
2078 undefined.

2079 The behavior of *call_once()* is undefined if *flag* has automatic storage duration or is not
2080 initialized by `ONCE_FLAG_INIT`.

2081 The *call_once()* function shall not be affected if the calling thread executes a signal handler
2082 during the call.[/CX]

2083 RETURN VALUE

2084 The *call_once()* function shall not return a value.

2085 ERRORS

2086 No errors are defined.

2087 EXAMPLES

2088 None.

2089 APPLICATION USAGE

2090 If *init_routine* recursively calls *call_once()* with the same *flag*, the recursive call will not call
2091 the specified *init_routine*, and thus the specified *init_routine* will not complete, and thus the
2092 recursive call to *call_once()* will not return. Use of *longjmp()* or *siglongjmp()* within an
2093 *init_routine* to jump to a point outside of *init_routine* prevents *init_routine* from returning.

2094 RATIONALE

2095 For dynamic library initialization in a multi-threaded process, if an initialization flag is used
2096 the flag needs to be protected against modification by multiple threads simultaneously
2097 calling into the library. This can be done by using a statically-initialized mutex. However,
2098 the better solution is to use *call_once()* or *pthread_once()* which are designed for exactly
2099 this purpose, for example:

```
2100 #include <threads.h>
2101 static once_flag random_is_initialized = ONCE_FLAG_INIT;
2102 extern void initialize_random(void);

2103 int random_function()
2104 {
2105     call_once(&random_is_initialized, initialize_random);
2106     ...
2107     /* Operations performed after initialization. */
2108 }
```

2109 The *call_once()* function is not affected by signal handlers for the reasons stated in [xref to
2110 XRAT B.2.3].

2111 FUTURE DIRECTIONS

2112 None.

2113 SEE ALSO

2114 *pthread_once*

- 2115 XBD Section 4.12.2, <**threads.h**>
- 2116 **CHANGE HISTORY**
- 2117 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.
- 2118 Ref 7.22.3 para 1
- 2119 On page 637 line 22002 section `calloc()`, change:
- 2120 a pointer to any type of object
- 2121 to:
- 2122 a pointer to any type of object with a fundamental alignment requirement
- 2123 Ref 7.22.3 para 2
- 2124 On page 637 line 22008 section `calloc()`, add a new paragraph:
- 2125 For purposes of determining the existence of a data race, `calloc()` shall behave as though it
- 2126 accessed only memory locations accessible through its arguments and not other static
- 2127 duration storage. The function may, however, visibly modify the storage that it allocates.
- 2128 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV]
- 2129 [CX]`reallocarray()`,[/CX] and `realloc()` that allocate or deallocate a particular region of
- 2130 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
- 2131 deallocation call shall synchronize with the next allocation (if any) in this order.
- 2132 Ref 7.22.3.1
- 2133 On page 637 line 22029 section `calloc()`, add `aligned_alloc` to the SEE ALSO section.
- 2134 Ref G.6 para 6, F.10.1.4, F.10 para 11
- 2135 On page 639 line 22055 section `carg()`, add:
- 2136 [MXC]If z is $-0 \pm i0$, $\pm\pi$ shall be returned.
- 2137 If z is $+0 \pm i0$, ± 0 shall be returned.
- 2138 If z is $x \pm i0$ where x is negative, $\pm\pi$ shall be returned.
- 2139 If z is $x \pm i0$ where x is positive, ± 0 shall be returned.
- 2140 If z is $\pm 0 + iy$ where y is negative, $-\pi/2$ shall be returned.
- 2141 If z is $\pm 0 + iy$ where y is positive, $\pi/2$ shall be returned.
- 2142 If z is $-\text{Inf} \pm iy$ where y is positive and finite, $\pm\pi$ shall be returned.
- 2143 If z is $+\text{Inf} \pm iy$ where y is positive and finite, ± 0 shall be returned.
- 2144 If z is $x \pm i\text{Inf}$ where x is finite, $\pm\pi/2$ shall be returned.
- 2145 If z is $-\text{Inf} \pm i\text{Inf}$, $\pm 3\pi/4$ shall be returned.

- 2146 If z is $+\text{Inf} \pm i\text{Inf}$, $\pm\pi/4$ shall be returned.
- 2147 If the real or imaginary part of z is NaN, NaN shall be returned.[/MXC]
- 2148 Ref G.6 para 7, G.6.2.2
2149 On page 640 line 22086 section `casin()`, add:
- 2150 [MXC]`casin(conj(iz))`, `casinf(conjf(iz))` and `casinl(conjl(iz))` shall return exactly the same
2151 value as `conj(casin(iz))`, `conjf(casinf(iz))` and `conjl(casinl(iz))`, respectively, and `casin(-iz)`,
2152 `casinf(-iz)` and `casinl(-iz)` shall return exactly the same value as `-casin(iz)`, `-casinf(iz)` and
2153 `-casinl(iz)`, respectively, including for the special values of iz below.
- 2154 If iz is $+0 + i0$, $-i(0 + i0)$ shall be returned.
- 2155 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i(+\text{Inf} + i\pi/2)$ shall be returned.
- 2156 If iz is $x + i\text{NaN}$ where x is finite, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the invalid
2157 floating-point exception may be raised.
- 2158 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i(+\text{Inf} + i0)$ shall be returned.
- 2159 If iz is $+\text{Inf} + i\text{Inf}$, $-i(+\text{Inf} + i\pi/4)$ shall be returned.
- 2160 If iz is $+\text{Inf} + i\text{NaN}$, $-i(+\text{Inf} + i\text{NaN})$ shall be returned.
- 2161 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.
- 2162 If iz is $\text{NaN} + iy$ where y is non-zero and finite, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
2163 invalid floating-point exception may be raised.
- 2164 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
2165 result is unspecified.
- 2166 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2167 Ref G.6 para 7
2168 On page 640 line 22094 section `casin()`, change RATIONALE from:
- 2169 None.
- 2170 to:
- 2171 The MXC special cases for `casin()` are derived from those for `casinh()` by applying the
2172 formula $\text{casin}(z) = -i \text{casinh}(iz)$.
- 2173 Ref G.6.2.2
2174 On page 641 line 22118 section `casinh()`, add:
- 2175 [MXC]`casinh(conj(z))`, `casinhf(conjf(z))` and `casinhl(conjl(z))` shall return exactly the same
2176 value as `conj(casinh(z))`, `conjf(casinhf(z))` and `conjl(casinhl(z))`, respectively, and `casinh(-z)`,
2177 `casinhf(-z)` and `casinhl(-z)` shall return exactly the same value as `-casinh(z)`, `-casinhf(z)`
2178 and `-casinhl(z)`, respectively, including for the special values of z below.

- 2179 If z is $+0 + i0$, $0 + i0$ shall be returned.
- 2180 If z is $x + i\text{Inf}$ where x is positive-signed and finite, $+\text{Inf} + i\pi/2$ shall be returned.
- 2181 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2182 point exception may be raised.
- 2183 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.
- 2184 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.
- 2185 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2186 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.
- 2187 If z is $\text{NaN} + iy$ where y is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2188 floating-point exception may be raised.
- 2189 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
2190 unspecified.
- 2191 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2192 Ref G.6 para 7, G.6.2.3
2193 On page 643 line 22157 section *catan*, add:
- 2194 [MXC]*catan(conj(iz))*, *catanf(conjf(iz))* and *catanl(conjl(iz))* shall return exactly the same
2195 value as *conj(catan(iz))*, *conjf(catanf(iz))* and *conjl(catanl(iz))*, respectively, and *catan(-iz)*,
2196 *catanf(-iz)* and *catanl(-iz)* shall return exactly the same value as $-\text{catan}(iz)$, $-\text{catanf}(iz)$ and
2197 $-\text{catanl}(iz)$, respectively, including for the special values of iz below.
- 2198 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.
- 2199 If iz is $+0 + i\text{NaN}$, $-i (+0 + i\text{NaN})$ shall be returned.
- 2200 If iz is $+1 + i0$, $-i (+\text{Inf} + i0)$ shall be returned and the divide-by-zero floating-point
2201 exception shall be raised.
- 2202 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2203 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
2204 invalid floating-point exception may be raised.
- 2205 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2206 If iz is $+\text{Inf} + i\text{Inf}$, $-i (+0 + i\pi/2)$ shall be returned.
- 2207 If iz is $+\text{Inf} + i\text{NaN}$, $-i (+0 + i\text{NaN})$ shall be returned.
- 2208 If iz is $\text{NaN} + iy$ where y is finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the invalid
2209 floating-point exception may be raised.

2210 If iz is NaN + $i\text{Inf}$, $-i(\pm 0 + i\pi/2)$ shall be returned; the sign of the imaginary part of the
2211 result is unspecified.

2212 If iz is NaN + $i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2213 Ref G.6 para 7
2214 On page 643 line 22165 section `catan()`, change RATIONALE from:

2215 None.

2216 to:

2217 The MXC special cases for `catan()` are derived from those for `catanh()` by applying the
2218 formula $\text{catan}(z) = -i \text{catanh}(iz)$.

2219 Ref G.6.2.3
2220 On page 644 line 22189 section `catanh`, add:

2221 [MXC]`catanh(conj(z))`, `catanhf(conjf(z))` and `catanhl(conj(z))` shall return exactly the same
2222 value as `conj(catanh(z))`, `conjf(catanhf(z))` and `conjl(catanhl(z))`, respectively, and
2223 `catanh(-z)`, `catanhf(-z)` and `catanhl(-z)` shall return exactly the same value as $-\text{catanh}(z)$,
2224 $-\text{catanhf}(z)$ and $-\text{catanhl}(z)$, respectively, including for the special values of z below.

2225 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2226 If z is $+0 + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.

2227 If z is $+1 + i0$, $+\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2228 shall be raised.

2229 If z is $x + i\text{Inf}$ where x is positive-signed and finite, $+0 + i\pi/2$ shall be returned.

2230 If z is $x + i\text{NaN}$ where x is non-zero and finite, NaN + $i\text{NaN}$ shall be returned and the invalid
2231 floating-point exception may be raised.

2232 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\pi/2$ shall be returned.

2233 If z is $+\text{Inf} + i\text{Inf}$, $+0 + i\pi/2$ shall be returned.

2234 If z is $+\text{Inf} + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.

2235 If z is NaN + iy where y is finite, NaN + $i\text{NaN}$ shall be returned and the invalid floating-
2236 point exception may be raised.

2237 If z is NaN + $i\text{Inf}$, $\pm 0 + i\pi/2$ shall be returned; the sign of the real part of the result is
2238 unspecified.

2239 If z is NaN + $i\text{NaN}$, NaN + $i\text{NaN}$ shall be returned.[/MXC]

2240 Ref G.6 para 7, G.6.2.4
2241 On page 652 line 22426 section `ccos()`, add:

2242 [MXC] $ccos(conj(iz))$, $ccosf(conjf(iz))$ and $ccosl(conjl(iz))$ shall return exactly the same value
2243 as $conj(ccos(iz))$, $conjf(ccosf(iz))$ and $conjl(ccosl(iz))$, respectively, and $ccos(-iz)$, $ccosf(-iz)$
2244 and $ccosl(-iz)$ shall return exactly the same value as $ccos(iz)$, $ccosf(iz)$ and $ccosl(iz)$,
2245 respectively, including for the special values of iz below.

2246 If iz is $+0 + i0$, $1 + i0$ shall be returned.

2247 If iz is $+0 + iInf$, $NaN \pm i0$ shall be returned and the invalid floating-point exception shall be
2248 raised; the sign of the imaginary part of the result is unspecified.

2249 If iz is $+0 + iNaN$, $NaN \pm i0$ shall be returned; the sign of the imaginary part of the result is
2250 unspecified.

2251 If iz is $x + iInf$ where x is non-zero and finite, $NaN + iNaN$ shall be returned and the invalid
2252 floating-point exception shall be raised.

2253 If iz is $x + iNaN$ where x is non-zero and finite, $NaN + iNaN$ shall be returned and the
2254 invalid floating-point exception may be raised.

2255 If iz is $+Inf + i0$, $+Inf + i0$ shall be returned.

2256 If iz is $+Inf + iy$ where y is non-zero and finite, $+Inf (\cos(y) + isin(y))$ shall be returned.

2257 If iz is $+Inf + iInf$, $\pm Inf + iNaN$ shall be returned and the invalid floating-point exception
2258 shall be raised; the sign of the real part of the result is unspecified.

2259 If iz is $+Inf + iNaN$, $+Inf + iNaN$ shall be returned.

2260 If iz is $NaN + i0$, $NaN \pm i0$ shall be returned; the sign of the imaginary part of the result is
2261 unspecified.

2262 If iz is $NaN + iy$ where y is any non-zero number, $NaN + iNaN$ shall be returned and the
2263 invalid floating-point exception may be raised.

2264 If iz is $NaN + iNaN$, $NaN + iNaN$ shall be returned.[/MXC]

2265 Ref G.6 para 7
2266 On page 652 line 22434 section $ccos()$, change RATIONALE from:

2267 None.

2268 to:

2269 The MXC special cases for $ccos()$ are derived from those for $ccosh()$ by applying the
2270 formula $ccos(z) = ccosh(iz)$.

2271 Ref G.6.2.4
2272 On page 653 line 22455 section $ccosh()$, add:

2273 [MXC] $ccosh(conj(z))$, $ccoshf(conjf(z))$ and $ccoshl(conjl(z))$ shall return exactly the same
2274 value as $conj(ccosh(z))$, $conjf(ccoshf(z))$ and $conjl(ccoshl(z))$, respectively, and $ccosh(-z)$,

2275 $ccoshf(-z)$ and $ccoshl(-z)$ shall return exactly the same value as $ccosh(z)$, $ccoshf(z)$ and
2276 $ccoshl(z)$, respectively, including for the special values of z below.

2277 If z is $+0 + i0$, $1 + i0$ shall be returned.

2278 If z is $+0 + i\text{Inf}$, $\text{NaN} \pm i0$ shall be returned and the invalid floating-point exception shall be
2279 raised; the sign of the imaginary part of the result is unspecified.

2280 If z is $+0 + i\text{NaN}$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2281 unspecified.

2282 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
2283 floating-point exception shall be raised.

2284 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
2285 floating-point exception may be raised.

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

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

2288 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2289 shall be raised; the sign of the real part of the result is unspecified.

2290 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2291 If z is $\text{NaN} + i0$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2292 unspecified.

2293 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2294 invalid floating-point exception may be raised.

2295 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2296 Ref F.10.6.1 para 4
2297 On page 655 line 22489 section `ceil()`, add a new paragraph:

2298 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2299 arguments.[/MX]

2300 Ref F.10.6.1 para 2
2301 On page 655 line 22491 section `ceil()`, change:

2302 [MX]The result shall have the same sign as x .[/MX]

2303 to:

2304 [MX]The returned value shall be independent of the current rounding direction mode and
2305 shall have the same sign as x .[/MX]

2306 Ref F.10.6.1 para 4
2307 On page 655 line 22504 section `ceil()`, delete from APPLICATION USAGE:

2308 These functions may raise the inexact floating-point exception if the result differs in value
2309 from the argument.

2310 Ref G.6.3.1
2311 On page 657 line 22539 section `cexp()`, add:

2312 [MXC]`cexp(conj(z))`, `cexpf(conjf(z))` and `cexpl(conjl(z))` shall return exactly the same value
2313 as `conj(cexp(z))`, `conjf(cexpf(z))` and `conjl(cexpl(z))`, respectively, including for the special
2314 values of z below.

2315 If z is $\pm 0 + i0$, $1 + i0$ shall be returned.

2316 If z is $x + i\text{Inf}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-point
2317 exception shall be raised.

2318 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2319 point exception may be raised.

2320 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2321 If z is $-\text{Inf} + iy$ where y is finite, $+0 (\cos(y) + i\sin(y))$ shall be returned.

2322 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2323 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
2324 result are unspecified.

2325 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2326 shall be raised; the sign of the real part of the result is unspecified.

2327 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
2328 result are unspecified.

2329 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
2330 unspecified.

2331 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2332 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2333 invalid floating-point exception may be raised.

2334 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2335 Ref 7.26.5.7
2336 On page 679 line 23268 section `clock_getres()`, change:

2337 including the `nanosleep()` function

2338 to:

2339 including the `nanosleep()` and `thrd_sleep()` functions

2340 Ref G.6.3.2
2341 On page 687 line 23495 section `clog()`, add:

2342 [MXC]`clog(conj(z))`, `clogf(conjf(z))` and `clogl(conjl(z))` shall return exactly the same value as
2343 `conj(clog(z))`, `conjf(clogf(z))` and `conjl(clogl(z))`, respectively, including for the special
2344 values of z below.

2345 If z is $-0 + i0$, $-\text{Inf} + i\pi$ shall be returned and the divide-by-zero floating-point exception
2346 shall be raised.

2347 If z is $+0 + i0$, $-\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2348 shall be raised.

2349 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2350 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2351 point exception may be raised.

2352 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2353 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2354 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

2355 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2356 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2357 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2358 point exception may be raised.

2359 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2360 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2361 Ref 7.26.3
2362 On page 698 line 23854 insert the following new `cond_*`() sections:

2363 **NAME**
2364 `cond_broadcast`, `cond_signal` — broadcast or signal a condition

2365 **SYNOPSIS**
2366 `#include <threads.h>`

2367 `int cond_broadcast(cond_t *cond);`
2368 `int cond_signal(cond_t *cond);`

2369 **DESCRIPTION**
2370 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2371 Any conflict between the requirements described here and the ISO C standard is
2372 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2373 The *cond_broadcast()* function shall unblock all of the threads that are blocked on the
2374 condition variable pointed to by *cond* at the time of the call.

2375 The *cond_signal()* function shall unblock one of the threads that are blocked on the condition
2376 variable pointed to by *cond* at the time of the call (if any threads are blocked on *cond*).

2377 If no threads are blocked on the condition variable pointed to by *cond* at the time of the call,
2378 these functions shall have no effect and shall return *thrd_success*.

2379 [CX]If more than one thread is blocked on a condition variable, the scheduling policy shall
2380 determine the order in which threads are unblocked. When each thread unblocked as a result
2381 of a *cond_broadcast()* or *cond_signal()* returns from its call to *cond_wait()* or *cond_timedwait()*,
2382 the thread shall own the mutex with which it called *cond_wait()* or *cond_timedwait()*. The
2383 thread(s) that are unblocked shall contend for the mutex according to the scheduling policy
2384 (if applicable), and as if each had called *mtx_lock()*.

2385 The *cond_broadcast()* and *cond_signal()* functions can be called by a thread whether or not it
2386 currently owns the mutex that threads calling *cond_wait()* or *cond_timedwait()* have associated
2387 with the condition variable during their waits; however, if predictable scheduling behavior is
2388 required, then that mutex shall be locked by the thread calling *cond_broadcast()* or
2389 *cond_signal()*.

2390 These functions shall not be affected if the calling thread executes a signal handler during
2391 the call.[/CX]

2392 The behavior is undefined if the value specified by the *cond* argument to *cond_broadcast()* or
2393 *cond_signal()* does not refer to an initialized condition variable.

2394 **RETURN VALUE**

2395 These functions shall return *thrd_success* on success, or *thrd_error* if the request
2396 could not be honored.

2397 **ERRORS**

2398 No errors are defined.

2399 **EXAMPLES**

2400 None.

2401 **APPLICATION USAGE**

2402 See the APPLICATION USAGE section for *pthread_cond_broadcast()*, substituting
2403 *cond_broadcast()* for *pthread_cond_broadcast()* and *cond_signal()* for *pthread_cond_signal()*.

2404 **RATIONALE**

2405 As for *pthread_cond_broadcast()* and *pthread_cond_signal()*, spurious wakeups may occur
2406 with *cond_broadcast()* and *cond_signal()*, necessitating that applications code a predicate-
2407 testing-loop around the condition wait. (See the RATIONALE section for
2408 *pthread_cond_broadcast()*.)

2409 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2410 B.2.3].

2411 **FUTURE DIRECTIONS**

2412 None.

2413 **SEE ALSO**

2414 *cond_destroy*, *cond_timedwait*, *pthread_cond_broadcast*

2415 XBD Section 4.12.2, <**threads.h**>

2416 **CHANGE HISTORY**

2417 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2418 **NAME**

2419 *cond_destroy*, *cond_init* — destroy and initialize condition variables

2420 **SYNOPSIS**

2421 `#include <threads.h>`

2422 `void cond_destroy(cond_t *cond);`

2423 `int cond_init(cond_t *cond);`

2424 **DESCRIPTION**

2425 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2426 Any conflict between the requirements described here and the ISO C standard is
2427 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2428 The *cond_destroy()* function shall release all resources used by the condition variable pointed
2429 to by *cond*. It shall be safe to destroy an initialized condition variable upon which no threads
2430 are currently blocked. Attempting to destroy a condition variable upon which other threads
2431 are currently blocked results in undefined behavior. A destroyed condition variable object
2432 can be reinitialized using *cond_init()*; the results of otherwise referencing the object after it
2433 has been destroyed are undefined. The behavior is undefined if the value specified by the
2434 *cond* argument to *cond_destroy()* does not refer to an initialized condition variable.

2435 The *cond_init()* function shall initialize a condition variable. If it succeeds it shall set the
2436 variable pointed to by *cond* to a value that uniquely identifies the newly initialized condition
2437 variable. Attempting to initialize an already initialized condition variable results in
2438 undefined behavior. A thread that calls *cond_wait()* on a newly initialized condition variable
2439 shall block.

2440 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
2441 further requirements.

2442 These functions shall not be affected if the calling thread executes a signal handler during
2443 the call.[/CX]

2444 **RETURN VALUE**

2445 The *cond_destroy()* function shall not return a value.

2446 The *cond_init()* function shall return *thrd_success* on success, or *thrd_nomem* if no
2447 memory could be allocated for the newly created condition, or *thrd_error* if the request
2448 could not be honored.

2449 **ERRORS**

2450 See RETURN VALUE.

2451 **EXAMPLES**

2452 None.

2453 **APPLICATION USAGE**

2454 None.

2455 **RATIONALE**

2456 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2457 B.2.3].

2458 **FUTURE DIRECTIONS**

2459 None.

2460 **SEE ALSO**

2461 *cond_broadcast*, *cond_timedwait*

2462 XBD <**threads.h**>

2463 **CHANGE HISTORY**

2464 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2465 **NAME**

2466 *cond_timedwait*, *cond_wait* — wait on a condition

2467 **SYNOPSIS**

```
2468 #include <threads.h>  
2469 int cond_timedwait(cond_t * restrict cond, mtx_t * restrict mtx,  
2470                   const struct timespec * restrict ts);  
2471 int cond_wait(cond_t *cond, mtx_t *mtx);
```

2472 **DESCRIPTION**

2473 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2474 Any conflict between the requirements described here and the ISO C standard is
2475 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2476 The *cond_timedwait()* function shall atomically unlock the mutex pointed to by *mtx* and block
2477 until the condition variable pointed to by *cond* is signaled by a call to *cond_signal()* or to
2478 *cond_broadcast()*, or until after the TIME_UTC-based calendar time pointed to by *ts*, or until
2479 it is unblocked due to an unspecified reason.

2480 The *cond_wait()* function shall atomically unlock the mutex pointed to by *mtx* and block until
2481 the condition variable pointed to by *cond* is signaled by a call to *cond_signal()* or to
2482 *cond_broadcast()*, or until it is unblocked due to an unspecified reason.

2483 [CX]Atomically here means "atomically with respect to access by another thread to the
2484 mutex and then the condition variable". That is, if another thread is able to acquire the mutex
2485 after the about-to-block thread has released it, then a subsequent call to *cond_broadcast()* or
2486 *cond_signal()* in that thread shall behave as if it were issued after the about-to-block thread
2487 has blocked.[/CX]

2488 When the calling thread becomes unblocked, these functions shall lock the mutex pointed to
2489 by *mtx* before they return. The application shall ensure that the mutex pointed to by *mtx* is
2490 locked by the calling thread before it calls these functions.

2491 When using condition variables there is always a Boolean predicate involving shared
2492 variables associated with each condition wait that is true if the thread should proceed.
2493 Spurious wakeups from the *cond_timedwait()* and *cond_wait()* functions may occur. Since the
2494 return from *cond_timedwait()* or *cond_wait()* does not imply anything about the value of this
2495 predicate, the predicate should be re-evaluated upon such return.

2496 When a thread waits on a condition variable, having specified a particular mutex to either
2497 the *cond_timedwait()* or the *cond_wait()* operation, a dynamic binding is formed between that
2498 mutex and condition variable that remains in effect as long as at least one thread is blocked
2499 on the condition variable. During this time, the effect of an attempt by any thread to wait on
2500 that condition variable using a different mutex is undefined. Once all waiting threads have
2501 been unblocked (as by the *cond_broadcast()* operation), the next wait operation on
2502 that condition variable shall form a new dynamic binding with the mutex specified by that
2503 wait operation. Even though the dynamic binding between condition variable and mutex
2504 might be removed or replaced between the time a thread is unblocked from a wait on the
2505 condition variable and the time that it returns to the caller or begins cancellation cleanup, the
2506 unblocked thread shall always re-acquire the mutex specified in the condition wait operation
2507 call from which it is returning.

2508 [CX]A condition wait (whether timed or not) is a cancellation point. When the cancelability
2509 type of a thread is set to *PTHREAD_CANCEL_DEFERRED*, a side-effect of acting upon a
2510 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired
2511 before calling the first cancellation cleanup handler. The effect is as if the thread were
2512 unblocked, allowed to execute up to the point of returning from the call to *cond_timedwait()*
2513 or *cond_wait()*, but at that point notices the cancellation request and instead of returning to
2514 the caller of *cond_timedwait()* or *cond_wait()*, starts the thread cancellation activities, which
2515 includes calling cancellation cleanup handlers.

2516 A thread that has been unblocked because it has been canceled while blocked in a call to
2517 *cond_timedwait()* or *cond_wait()* shall not consume any condition signal that may be directed
2518 concurrently at the condition variable if there are other threads blocked on the condition
2519 variable.[/CX]

2520 When *cond_timedwait()* times out, it shall nonetheless release and re-acquire the mutex
2521 referenced by *mutex*, and may consume a condition signal directed concurrently at the
2522 condition variable.

2523 [CX]These functions shall not be affected if the calling thread executes a signal handler
2524 during the call, except that if a signal is delivered to a thread waiting for a condition
2525 variable, upon return from the signal handler either the thread shall resume waiting for the
2526 condition variable as if it was not interrupted, or it shall return *thrd_success* due to
2527 spurious wakeup.[/CX]

2528 The behavior is undefined if the value specified by the *cond* or *mtx* argument to these
2529 functions does not refer to an initialized condition variable or an initialized mutex object,
2530 respectively.

2531 **RETURN VALUE**

2532 The *cn_d_timedwait()* function shall return *thrd_success* upon success, or
2533 *thrd_timedout* if the time specified in the call was reached without acquiring the
2534 requested resource, or *thrd_error* if the request could not be honored.

2535 The *cn_d_wait()* function shall return *thrd_success* upon success or *thrd_error* if the
2536 request could not be honored.

2537 **ERRORS**

2538 See RETURN VALUE.

2539 **EXAMPLES**

2540 None.

2541 **APPLICATION USAGE**

2542 None.

2543 **RATIONALE**

2544 These functions are not affected by signal handlers (except as stated in the DESCRIPTION)
2545 for the reasons stated in [xref to XRAT B.2.3].

2546 **FUTURE DIRECTIONS**

2547 None.

2548 **SEE ALSO**

2549 *cn_d_broadcast*, *cn_d_destroy*, *timespec_get*

2550 XBD Section 4.12.2, <**threads.h**>

2551 **CHANGE HISTORY**

2552 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2553 Ref F.10.8.1 para 2

2554 On page 705 line 24155 section *copysign()*, add a new paragraph:

2555 [MX]The returned value shall be exact and shall be independent of the current rounding
2556 direction mode.[/MX]

2557 Ref G.6.4.1 para 1

2558 On page 711 line 24308 section *cpow()*, add a new paragraph:

2559 [MXC]These functions shall raise floating-point exceptions if appropriate for the calculation
2560 of the parts of the result, and may also raise spurious floating-point exceptions.[/MXC]

2561 Ref G.6.4.1 footnote 386

2562 On page 711 line 24318 section *cpow()*, change RATIONALE from:

2563 None.

2564 to:

2565 Permitting spurious floating-point exceptions allows *cpow(z, c)* to be implemented as *cexp(c*

2566 $clog(z)$ without precluding implementations that treat special cases more carefully.

2567 Ref G.6 para 7, G.6.2.5
2568 On page 718 line 24545 section `csin()`, add:

2569 [MXC]`csin(conj(iz))`, `csinf(conjf(iz))` and `csinl(conjl(iz))` shall return exactly the same value
2570 as `conj(csin(iz))`, `conjf(csinf(iz))` and `conjl(csinl(iz))`, respectively, and `csin(-iz)`, `csinf(-iz)`
2571 and `csinl(-iz)` shall return exactly the same value as `-csin(iz)`, `-csinf(iz)` and `-csinl(iz)`,
2572 respectively, including for the special values of iz below.

2573 If iz is $+0 + i0$, $-i(+0 + i0)$ shall be returned.

2574 If iz is $+0 + i\text{Inf}$, $-i(\pm 0 + i\text{NaN})$ shall be returned and the invalid floating-point exception
2575 shall be raised; the sign of the imaginary part of the result is unspecified.

2576 If iz is $+0 + i\text{NaN}$, $-i(\pm 0 + i\text{NaN})$ shall be returned; the sign of the imaginary part of the
2577 result is unspecified.

2578 If iz is $x + i\text{Inf}$ where x is positive and finite, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
2579 invalid floating-point exception shall be raised.

2580 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
2581 invalid floating-point exception may be raised.

2582 If iz is $+\text{Inf} + i0$, $-i(+\text{Inf} + i0)$ shall be returned.

2583 If iz is $+\text{Inf} + iy$ where y is positive and finite, $-i\text{Inf}(\cos(y) + i\sin(y))$ shall be returned.

2584 If iz is $+\text{Inf} + i\text{Inf}$, $-i(\pm\text{Inf} + i\text{NaN})$ shall be returned and the invalid floating-point exception
2585 shall be raised; the sign of the imaginary part of the result is unspecified.

2586 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
2587 result is unspecified.

2588 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.

2589 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
2590 invalid floating-point exception may be raised.

2591 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2592 Ref G.6 para 7
2593 On page 718 line 24553 section `csin()`, change RATIONALE from:

2594 None.

2595 to:

2596 The MXC special cases for `csin()` are derived from those for `csinh()` by applying the formula
2597 $csin(z) = -i csinh(iz)$.

2598 Ref G.6.2.5

2599 On page 719 line 24574 section `csinh()`, add:

2600 [MXC]`csinh(conj(z))`, `csinhf(conjf(z))` and `csinhl(conjl(z))` shall return exactly the same
2601 value as `conj(csinh(z))`, `conjf(csinhf(z))` and `conjl(csinhl(z))`, respectively, and `csinh(-z)`,
2602 `csinhf(-z)` and `csinhl(-z)` shall return exactly the same value as `-csinh(z)`, `-csinhf(z)` and
2603 `-csinhl(z)`, respectively, including for the special values of z below.

2604 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2605 If z is $+0 + i\text{Inf}$, $\pm 0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2606 raised; the sign of the real part of the result is unspecified.

2607 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
2608 unspecified.

2609 If z is $x + i\text{Inf}$ where x is positive and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2610 floating-point exception shall be raised.

2611 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
2612 floating-point exception may be raised.

2613 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2614 If z is $+\text{Inf} + iy$ where y is positive and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2615 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2616 shall be raised; the sign of the real part of the result is unspecified.

2617 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
2618 unspecified.

2619 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2620 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2621 invalid floating-point exception may be raised.

2622 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2623 Ref G.6.4.2

2624 On page 721 line 24612 section `csqrt()`, add:

2625 [MXC]`csqrt(conj(z))`, `csqrtf(conjf(z))` and `csqrtl(conjl(z))` shall return exactly the same value
2626 as `conj(csqrt(z))`, `conjf(csqrtf(z))` and `conjl(csqrtl(z))`, respectively, including for the special
2627 values of z below.

2628 If z is $\pm 0 + i0$, $+0 + i0$ shall be returned.

2629 If the imaginary part of z is Inf , $+\text{Inf} + i\text{Inf}$, shall be returned.

2630 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2631 point exception may be raised.

- 2632 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\text{Inf}$ shall be returned.
- 2633 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.
- 2634 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
2635 is unspecified.
- 2636 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2637 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2638 point exception may be raised.
- 2639 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2640 Ref G.6 para 7, G.6.2.6
2641 On page 722 line 24641 section $\text{ctan}()$, add:
- 2642 [MXC] $\text{ctan}(\text{conj}(iz))$, $\text{ctanf}(\text{conj}(iz))$ and $\text{ctanl}(\text{conjl}(iz))$ shall return exactly the same value
2643 as $\text{conj}(\text{ctan}(iz))$, $\text{conj}(\text{ctanf}(iz))$ and $\text{conjl}(\text{ctanl}(iz))$, respectively, and $\text{ctan}(-iz)$, $\text{ctanf}(-iz)$
2644 and $\text{ctanl}(-iz)$ shall return exactly the same value as $-\text{ctan}(iz)$, $-\text{ctanf}(iz)$ and $-\text{ctanl}(iz)$,
2645 respectively, including for the special values of iz below.
- 2646 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.
- 2647 If iz is $0 + i\text{Inf}$, $-i (0 + i\text{NaN})$ shall be returned and the invalid floating-point exception shall
2648 be raised.
- 2649 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
2650 invalid floating-point exception shall be raised.
- 2651 If iz is $0 + i\text{NaN}$, $-i (0 + i\text{NaN})$ shall be returned.
- 2652 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
2653 invalid floating-point exception may be raised.
- 2654 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i (1 + i0 \sin(2y))$ shall be returned.
- 2655 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
2656 unspecified.
- 2657 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
2658 unspecified.
- 2659 If iz is $\text{NaN} + i0$, $-i (\text{NaN} + i0)$ shall be returned.
- 2660 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
2661 invalid floating-point exception may be raised.
- 2662 If iz is $\text{NaN} + i\text{NaN}$, $-i (\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2663 Ref G.6 para 7
2664 On page 722 line 24649 section $\text{ctan}()$, change RATIONALE from:

2665 None.

2666 to:

2667 The MXC special cases for *ctan()* are derived from those for *ctanh()* by applying the
2668 formula $ctan(z) = -i ctanh(iz)$.

2669 Ref G.6.2.6
2670 On page 723 line 24670 section *ctanh()*, add:

2671 [MXC]*ctanh(conj(z))*, *ctanhf(conjf(z))* and *ctanhl(conjl(z))* shall return exactly the same
2672 value as *conj(ctanh(z))*, *conjf(ctanhf(z))* and *conjl(ctanhl(z))*, respectively, and *ctanh(-z)*,
2673 *ctanhf(-z)* and *ctanhl(-z)* shall return exactly the same value as $-ctanh(z)$, $-ctanhf(z)$ and
2674 $-ctanhl(z)$, respectively, including for the special values of *z* below.

2675 If *z* is $+0 + i0$, $+0 + i0$ shall be returned.

2676 If *z* is $0 + i\text{Inf}$, $0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2677 raised.

2678 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
2679 floating-point exception shall be raised.

2680 If *z* is $0 + i\text{NaN}$, $0 + i\text{NaN}$ shall be returned.

2681 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
2682 floating-point exception may be raised.

2683 If *z* is $+\text{Inf} + iy$ where *y* is positive-signed and finite, $1 + i0 \sin(2y)$ shall be returned.

2684 If *z* is $+\text{Inf} + i\text{Inf}$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2685 unspecified.

2686 If *z* is $+\text{Inf} + i\text{NaN}$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2687 unspecified.

2688 If *z* is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2689 If *z* is $\text{NaN} + iy$ where *y* is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2690 invalid floating-point exception may be raised.

2691 If *z* is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2692 Ref 7.27.3, 7.1.4 para 5
2693 On page 727 line 24774 section *ctime()*, change:

2694 [CX]The *ctime()* function need not be thread-safe.[/CX]

2695 to:

2696 The *ctime()* function need not be thread-safe; however, *ctime()* shall avoid data races with all
2697 functions other than itself, *asctime()*, *gmtime()* and *localtime()*.

2698 Ref 7.5 para 2
2699 On page 781 line 26447 section `errno`, change:

2700 The lvalue *errno* is used by many functions to return error values.

2701 to:

2702 The lvalue to which the macro *errno* expands is used by many functions to return error
2703 values.

2704 Ref 7.5 para 3
2705 On page 781 line 26449 section `errno`, change:

2706 The value of *errno* shall be defined only after a call to a function for which it is explicitly
2707 stated to be set and until it is changed by the next function call or if the application assigns it
2708 a value.

2709 to:

2710 The value of *errno* in the initial thread shall be zero at program startup (the initial value of
2711 *errno* in other threads is an indeterminate value) and shall otherwise be defined only after a
2712 call to a function for which it is explicitly stated to be set and until it is changed by the next
2713 function call or if the application assigns it a value.

2714 Ref 7.5 para 2
2715 On page 781 line 26456 section `errno`, delete:

2716 It is unspecified whether *errno* is a macro or an identifier declared with external linkage.

2717 Ref 7.22.4.4 para 2
2718 On page 796 line 27057 section `exit()`, add a new (unshaded) paragraph:

2719 The *exit()* function shall cause normal process termination to occur. No functions registered
2720 by the *at_quick_exit()* function shall be called. If a process calls the *exit()* function more
2721 than once, or calls the *quick_exit()* function in addition to the *exit()* function, the behavior is
2722 undefined.

2723 Ref 7.22.4.4 para 2
2724 On page 796 line 27068 section `exit()`, delete:

2725 If *exit()* is called more than once, the behavior is undefined.

2726 Ref 7.22.4.3, 7.22.4.7
2727 On page 796 line 27086 section `exit()`, add *at_quick_exit* and *quick_exit* to the SEE ALSO section.

2728 Ref F.10.4.2 para 2
2729 On page 804 line 27323 section `fabs()`, add a new paragraph:

2730 [MX]The returned value shall be exact and shall be independent of the current rounding
2731 direction mode.[/MX]

2732 Ref 7.21.2 para 7,8
2733 On page 874 line 29483 section flockfile(), change:

2734 These functions shall provide for explicit application-level locking of stdio (**FILE ***)
2735 objects.

2736 to:

2737 These functions shall provide for explicit application-level locking of the locks associated
2738 with standard I/O streams (see [xref to 2.5]).

2739 Ref 7.21.2 para 7,8
2740 On page 874 line 29499 section flockfile(), delete:

2741 All functions that reference (**FILE ***) objects, except those with names ending in *_unlocked*,
2742 shall behave as if they use *flockfile()* and *funlockfile()* internally to obtain ownership of these
2743 (**FILE ***) objects.

2744 Ref F.10.6.2 para 3
2745 On page 876 line 29560 section floor(), add a new paragraph:

2746 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2747 arguments.[/MX]

2748 Ref F.10.6.2 para 2
2749 On page 876 line 29562 section floor(), change:

2750 [MX]The result shall have the same sign as *x*.[/MX]

2751 to:

2752 [MX]The returned value shall be independent of the current rounding direction mode and
2753 shall have the same sign as *x*.[/MX]

2754 Ref F.10.6.2 para 3
2755 On page 876 line 29576 section floor(), delete from APPLICATION USAGE:

2756 These functions may raise the inexact floating-point exception if the result differs in value
2757 from the argument.

2758 Ref F.10.9.2 para 2
2759 On page 880 line 29695 section fmax(), add a new paragraph:

2760 [MX]The returned value shall be exact and shall be independent of the current rounding
2761 direction mode.[/MX]

2762 Ref F.10.9.3 para 2
2763 On page 884 line 29844 section fmin(), add a new paragraph:

2764 [MX]The returned value shall be exact and shall be independent of the current rounding
2765 direction mode.[/MX]

2766 Ref F.10.7.1 para 2
2767 On page 885 line 29892 section `fmod()`, change:

2768 [MXX]If the correct value would cause underflow, and is representable, a range error may
2769 occur and the correct value shall be returned.[/MXX]

2770 to:

2771 [MX]When subnormal results are supported, the returned value shall be exact and shall be
2772 independent of the current rounding direction mode.[/MX]

2773 Ref 7.21.5.3 para 5
2774 On page 892 line 30117 section `fopen()`, change:

2775 [CX]The functionality described on this reference page is aligned with the ISO C standard.
2776 Any conflict between the requirements described here and the ISO C standard is
2777 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2778 to:

2779 [CX]Except for the “exclusive access” requirement (see below), the functionality described
2780 on this reference page is aligned with the ISO C standard. Any other conflict between the
2781 requirements described here and the ISO C standard is unintentional. This volume of
2782 POSIX.1-202x defers to the ISO C standard for all `fopen()` functionality except in relation to
2783 “exclusive access”.[/CX]

2784 Ref 7.21.5.3 para 5
2785 On page 892 line 30122 section `fopen()`, after applying bug 411, change:

2786 The *mode* argument points to a character string. If the string begins with one of the following
2787 prefixes, followed by a (possibly empty) suffix consisting of the additional characters
2788 documented below, then the file shall be opened in the mode indicated by the prefix.
2789 Otherwise, the behavior is undefined.

2790	<i>r</i> or <i>rb</i>	Open file for reading.
2791	<i>w</i> or <i>wb</i>	Truncate to zero length or create file for writing.
2792	<i>a</i> or <i>ab</i>	Append; open or create file for writing at end-of-file.
2793	<i>r+</i> or <i>rb+</i> or <i>r+b</i>	Open file for update (reading and writing).
2794	<i>w+</i> or <i>wb+</i> or <i>w+b</i>	Truncate to zero length or create file for update.
2795	<i>a+</i> or <i>ab+</i> or <i>a+b</i>	Append; open or create file for update, writing at end-of-file.

2796 [CX]The character 'b' shall have no effect, but is allowed for ISO C standard
2797 conformance.[/CX]

2798 Additionally, the following characters can appear anywhere in the suffix of the *mode* string,
2799 to further affect how the file is opened. Behavior is unspecified if a character occurs more
2800 than once.

2801 [CX]e The underlying file descriptor shall have the FD_CLOEXEC flag atomically set, as if
2802 by the O_CLOEXEC flag to *open()*.[/CX]

2803 x If specified with a prefix beginning with *w* [CX]or *a*[/CX], then the function shall
2804 fail if the file already exists, [CX]as if by the O_EXCL flag to *open()*. If specified
2805 with a prefix beginning with *r*, this modifier shall have no effect. [/CX]

2806 Opening a file with read mode (*r* as the first character in the *mode* argument) shall fail if the
2807 file does not exist or cannot be read.

2808 Opening a file with append mode (*a* as the first character in the *mode* argument) shall cause
2809 all subsequent writes to the file to be forced to the then current end-of-file, regardless of
2810 intervening calls to *fseek()*.

2811 When a file is opened with update mode ('+' as the second or third character in the *mode*
2812 argument), both input and output may be performed on the associated stream.

2813 to:

2814 The *mode* argument points to a character string. The behavior is unspecified if any character
2815 occurs more than once in the string. If the string begins with one of the following characters,
2816 then the file shall be opened in the indicated mode. Otherwise, the behavior is undefined.

2817 'r' Open file for reading.

2818 'w' Truncate to zero length or create file for writing.

2819 'a' Append; open or create file for writing at end-of-file.

2820 The remainder of the string can contain any of the following characters, [CX]in any
2821 order[/CX], and further affect how the file is opened:

2822 'b' [CX]This character shall have no effect, but is allowed for ISO C standard
2823 conformance. [/CX]

2824 [CX]'e' The underlying file descriptor shall have the FD_CLOEXEC flag atomically
2825 set. [/CX]

2826 'x' If the first character of mode is 'w' [CX]or 'a'[/CX], then the function shall fail if the
2827 file already exists or cannot be created; if the file does not exist and can be created,
2828 it shall be created with [CX]an implementation-defined form of[/CX] exclusive
2829 (also known as non-shared) access, [CX]if supported by the underlying file system,
2830 provided the resulting file permissions are the same as they would be without the
2831 'x' modifier. If the first character of mode is 'r', the effect is implementation-
2832 defined. [/CX]

2833 **Note:** The ISO C standard requires exclusive access “to the extent that the underlying file
2834 system supports exclusive access”, but does not define what it means by this.
2835 Taken at face value—that systems must do whatever they are capable of, at the file
2836 system level, in order to exclude access by others—this would require POSIX.1
2837 systems to set the file permissions in a way that prevents access by other users and

2838 groups. Consequently, this volume of POSIX.1-202x does not defer to the ISO C
2839 standard as regards the “exclusive access” requirement.

2840 '+' The file shall be opened for update (both reading and writing), rather than just
2841 reading or just writing.

2842 Opening a file with read mode ('r' as the first character in the *mode* argument) shall fail if the
2843 file does not exist or cannot be read.

2844 Opening a file with append mode ('a' as the first character in the *mode* argument) shall cause
2845 all subsequent writes to the file to be forced to the then current end-of-file, regardless of
2846 intervening calls to *fseek()*.

2847 When a file is opened with update mode ('+' in the *mode* argument), both input and output
2848 can be performed on the associated stream.

2849 Ref 7.21.5.3 para 3
2850 On page 892 line 30144 section *fopen()*, after applying bug 411, change:

2851 If the *mode* prefix is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and ...

2852 to:

2853 If the first character in *mode* is 'w' or 'a', and ...

2854 Ref 7.21.5.3 para 3,5
2855 On page 892 line 30148 section *fopen()*, after applying bug 411, change:

2856 If the *mode* prefix is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not
2857 previously exist, the *fopen()* function shall create a file as if it called the *creat()* function
2858 with a value appropriate for the *path* argument interpreted from *pathname* and a value of
2859 S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH for the *mode*
2860 argument.

2861 If the mode prefix is *w*, *wb*, *w+*, *wb+*, or *w+b*, and the file did previously exist, upon
2862 successful completion, *fopen()* shall mark for update the last data modification and last file
2863 status change timestamps of the file.

2864 to:

2865 If the first character in *mode* is 'w' or 'a', and the file did not previously exist, the *fopen()*
2866 function shall create a file as if it called the *open()* function with a value appropriate for the
2867 *path* argument interpreted from *pathname*, a value for the *oflag* argument as specified below,
2868 and a value of S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH for
2869 the third argument.

2870 If the first character in *mode* is 'w', and the file did previously exist, upon successful
2871 completion, *fopen()* shall mark for update the last data modification and last file status
2872 change timestamps of the file.

2873 Ref 7.21.5.3 para 5
2874 On page 893 line 30158 section *fopen()*, change:

2875 The file descriptor associated with the opened stream shall be allocated and opened as if by
 2876 a call to *open()* with the following flags:

<i>fopen()</i> Mode Prefix	<i>open()</i> Flags
<i>r</i> or <i>rb</i>	O_RDONLY
<i>w</i> or <i>wb</i>	O_WRONLY O_CREAT O_TRUNC
<i>a</i> or <i>ab</i>	O_WRONLY O_CREAT O_APPEND
<i>r+</i> or <i>rb+</i> or <i>r+b</i>	O_RDWR
<i>w+</i> or <i>wb+</i> or <i>w+b</i>	O_RDWR O_CREAT O_TRUNC
<i>a+</i> or <i>ab+</i> or <i>a+b</i>	O_RDWR O_CREAT O_APPEND

2877 to:

2878 The file descriptor associated with the opened stream shall be allocated and opened as if by
 2879 a call to *open()* using the following flags, with the addition of the O_CLOEXEC flag if
 2880 mode includes 'e', and the O_EXCL flag if mode includes 'x' and either 'w' or 'a':

<i>fopen()</i> Mode First Character	<i>fopen()</i> Mode Includes '+'	<i>open()</i> Flags
'r'	no	O_RDONLY
'w'	no	O_WRONLY O_CREAT O_TRUNC
'a'	no	O_WRONLY O_CREAT O_APPEND
'r'	yes	O_RDWR
'w'	yes	O_RDWR O_CREAT O_TRUNC
'a'	yes	O_RDWR O_CREAT O_APPEND

2881 If *mode* includes 'x' and the underlying file system supports exclusive access (see above)
 2882 enabled by the use of implementation-specific flags to *open()*, then the behavior shall be as if
 2883 those flags are also included.

2884 Ref 7.21.5.3 para 5
 2885 On page 895 line 30236 section *fopen()*, change APPLICATION USAGE from:

2886 None.

2887 to:

2888 If an application needs to create a file in a way that fails if the file already exists, and either
 2889 requires that it does not have exclusive access to the file or does not need exclusive access, it
 2890 should use *open()* with the O_CREAT and O_EXCL flags instead of using *fopen()* with an *x*
 2891 in the *mode*. A stream can then be created, if needed, by calling *fdopen()* on the file
 2892 descriptor returned by *open()*.

2893 Ref 7.21.5.3 para 5

2894 On page 895 line 30238 section `fopen()`, after applying bug 411, change:

2895 The *x* mode suffix character was added by the ISO C standard only for files opened with a
2896 *mode* string beginning with *w*. However, this standard requires that it also work for *mode*
2897 strings beginning with *a*, as well as being silently ignored rather than being an error for
2898 *mode* strings beginning with *r*. Therefore, while `open()` has undefined behavior if `O_EXCL`
2899 is specified without `O_CREAT`, the same is not true of `fopen()`.

2900 to:

2901 The ISO C standard only recognizes the '+', 'b', and 'x' characters in certain positions of the
2902 *mode* string, leaving other arrangements as unspecified, and only permits 'x' in *mode* strings
2903 beginning with 'w'. This standard specifically requires support for all characters other than
2904 the first in the *mode* string to be recognized in any order. Thus, "wxe" and "wex" behave the
2905 same, and while "wx+" is unspecified in the ISO C standard, this standard requires it to have
2906 the same behavior as "w+x". This standard also requires that 'x' work for *mode* strings
2907 beginning with 'a', as well as having implementation-defined behavior for *mode* strings
2908 beginning with 'r'. Therefore, while `open()` has undefined behavior if `O_EXCL` is specified
2909 without `O_CREAT`, the same is not true of `fopen()`.

2910 When 'x' is in *mode*, the ISO C standard requires that the file is created with exclusive
2911 access to the extent that the underlying system supports exclusive access. Although POSIX.1
2912 does not specify any method of enabling exclusive access, it allows for the existence of an
2913 implementation-specific flag, or flags, that enable it. Note that they should be file creation
2914 flags if a file is being created, not file access mode flags (that is, ones that are included in
2915 `O_ACCMODE`) or file status flags, so that they do not affect the value returned by `fcntl()`
2916 with `F_GETFL`. On implementations that have such flags, if support for them is file system
2917 dependent and exclusive access is requested when using `fopen()` to create a file on a file
2918 system that does not support it, the flags must not be used if they would cause `fopen()` to fail.

2919 Some implementations support mandatory file locking as a means of enabling exclusive
2920 access to a file. Locks are set in the normal way, but instead of only preventing others from
2921 setting conflicting locks they prevent others from accessing the contents of the locked part
2922 of the file in a way that conflicts with the lock. However, unless the implementation has a
2923 way of setting a whole-file write lock on file creation, this does not satisfy the requirement
2924 in the ISO C standard that the file is "created with exclusive access to the extent that the
2925 underlying system supports exclusive access". (Having `fopen()` create the file and set a lock
2926 on the file as two separate operations is not the same, and it would introduce a race
2927 condition whereby another process could open the file and write to it (or set a lock) in
2928 between the two operations.) However, on all implementations that support mandatory file
2929 locking, its use is discouraged; therefore, it is recommended that implementations which
2930 support mandatory file locking do **not** add a means of creating a file with a whole-file
2931 exclusive lock set, so that `fopen()` is not required to enable mandatory file locking in order to
2932 conform to the ISO C standard. An implementation that has a means of creating a file with a
2933 whole-file exclusive lock set would need to provide a way to change the behavior of `fopen()`
2934 depending on whether the calling process is executing in a POSIX.1 conforming
2935 environment or an ISO C conforming environment.

2936 The typical implementation-defined behavior for mode "rx" is to ignore the 'x', but the
2937 standard developers did not wish to mandate this behavior. For example, an implementation
2938 could allow shared access for reading; that is, disallow a file that has been opened this way

2939 from also being opened for writing.

2940 Ref 7.22.3.3 para 2

2941 On page 933 line 31673 section `free()`, after applying bug 1218 change:

2942 Otherwise, if the argument does not match a pointer earlier returned by a function in
2943 POSIX.1-2017 that allocates memory as if by `malloc()`, or if the space has been deallocated
2944 by a call to `free()`, `realloc()`, [CX]or `reallocarray()`,[/CX] the behavior is undefined.

2945 to:

2946 Otherwise, if the argument does not match a pointer earlier returned by `aligned_alloc()`,
2947 `calloc()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV] `realloc()`, [CX]`reallocarray()`, or a
2948 function in POSIX.1-20xx that allocates memory as if by `malloc()`,[/CX] or if the space has
2949 been deallocated by a call to `free()`, [CX]`reallocarray()`,[/CX] or `realloc()`, the behavior is
2950 undefined.

2951 Ref 7.22.3 para 2

2952 On page 933 line 31677 section `free()`, add a new paragraph:

2953 For purposes of determining the existence of a data race, `free()` shall behave as though it
2954 accessed only memory locations accessible through its argument and not other static
2955 duration storage. The function may, however, visibly modify the storage that it deallocates.
2956 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV]
2957 [CX]`reallocarray()`,[/CX] and `realloc()` that allocate or deallocate a particular region of
2958 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
2959 deallocation call shall synchronize with the next allocation (if any) in this order.

2960 Ref 7.22.3.1

2961 On page 933 line 31691 section `free()`, add `aligned_alloc` to the SEE ALSO section.

2962 Ref 7.21.5.3 para 5

2963 On page 942 line 31988 section `freopen()`, change:

2964 [CX]The functionality described on this reference page is aligned with the ISO C standard.
2965 Any conflict between the requirements described here and the ISO C standard is
2966 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2967 to:

2968 [CX]Except for the “exclusive access” requirement (see [xref to `fopen()`]), the functionality
2969 described on this reference page is aligned with the ISO C standard. Any other conflict
2970 between the requirements described here and the ISO C standard is unintentional. This
2971 volume of POSIX.1-202x defers to the ISO C standard for all `freopen()` functionality except
2972 in relation to “exclusive access”.[/CX]

2973 Ref 7.21.5.3 para 3,5; 7.21.5.4 para 2

2974 On page 942 line 32010 section `freopen()`, replace the following text:

2975 shall be allocated and opened as if by a call to `open()` with the following flags:

2976 and the table that follows it, and the paragraph added by bug 411 after the table, with:

2977 shall be allocated and opened as if by a call to *open()* with the flags specified for *fopen()*
2978 with the same *mode* argument.

2979 Ref (none)
2980 On page 944 line 32094 section *freopen()*, change:

2981 It is possible that these side-effects are an unintended consequence of the way the feature is
2982 specified in the ISO/IEC 9899: 1999 standard, but unless or until the ISO C standard is
2983 changed, ...

2984 to:

2985 It is possible that these side-effects are an unintended consequence of the way the feature
2986 was specified in the ISO/IEC 9899: 1999 standard (and still is in the current standard), but
2987 unless or until the ISO C standard is changed, ...

2988 Ref (none)
2989 On page 944 line 32100 section *freopen()*, add a new paragraph to APPLICATION USAGE:

2990 See also the APPLICATION USAGE for [xref to *fopen()*].

2991 Ref (none)
2992 On page 944 line 32102 section *freopen()*, replace the RATIONALE additions made by bug 411
2993 with:

2994 See the RATIONALE for [xref to *fopen()*].

2995 Ref 7.12.6.4 para 3
2996 On page 947 line 32161 section *frexp()*, change:

2997 The integer exponent shall be stored in the **int** object pointed to by *exp*.

2998 to:

2999 The integer exponent shall be stored in the **int** object pointed to by *exp*; if the integer
3000 exponent is outside the range of **int**, the results are unspecified.

3001 Ref F.10.3.4 para 3
3002 On page 947 line 32164 section *frexp()*, add a new paragraph:

3003 [MX]When the radix of the argument is a power of 2, the returned value shall be exact and
3004 shall be independent of the current rounding direction mode.[/MX]

3005 Ref 7.21.6.2 para 4
3006 On page 950 line 32239 section *fscanf()*, change:

3007 If a directive fails, as detailed below, the function shall return.

3008 to:

3009 When all directives have been executed, or if a directive fails (as detailed below), the

3010 function shall return.

3011 Ref 7.21.6.2 para 5

3012 On page 950 line 32242 section fscanf(), after applying bug 1163 change:

3013 A directive composed of one or more white-space bytes shall be executed by reading input
3014 until no more valid input can be read, or up to the first non-white-space byte , which remains
3015 unread.

3016 to:

3017 A directive composed of one or more white-space bytes shall be executed by reading input
3018 up to the first non-white-space byte, which shall remain unread, or until no more bytes can
3019 be read. The directive shall never fail.

3020 Ref (none)

3021 On page 955 line 32471 section fscanf(), change:

3022 This function is aligned with the ISO/IEC 9899: 1999 standard, and in doing so a few
3023 “obvious” things were not included. Specifically, the set of characters allowed in a scanset is
3024 limited to single-byte characters. In other similar places, multi-byte characters have been
3025 permitted, but for alignment with the ISO/IEC 9899: 1999 standard, it has not been done
3026 here.

3027 to:

3028 The set of characters allowed in a scanset is limited to single-byte characters. In other
3029 similar places, multi-byte characters have been permitted, but for alignment with the ISO C
3030 standard, it has not been done here.

3031 Ref 7.29.2.2 para 4

3032 On page 1004 line 34144 section fwscanf(), change:

3033 If a directive fails, as detailed below, the function shall return.

3034 to:

3035 When all directives have been executed, or if a directive fails (as detailed below), the
3036 function shall return.

3037 Ref 7.29.2.2 para 5

3038 On page 1004 line 34147 section fwscanf(), change:

3039 A directive composed of one or more white-space wide characters is executed by reading
3040 input until no more valid input can be read, or up to the first wide character which is not a
3041 white-space wide character, which remains unread.

3042 to:

3043 A directive composed of one or more white-space wide characters shall be executed by
3044 reading input up to the first wide character that is not a white-space wide character, which
3045 shall remain unread, or until no more wide characters can be read. The directive shall never

3046 fail.

3047 Ref 7.27.3, 7.1.4 para 5

3048 On page 1113 line 37680 section `gmtime()`, change:

3049 ~~[CX]The `gmtime()` function need not be thread-safe.[/CX]~~

3050 to:

3051 The `gmtime()` function need not be thread-safe; however, `gmtime()` shall avoid data races

3052 with all functions other than itself, `asctime()`, `ctime()` and `localtime()`.

3053 Ref F.10.3.5 para 1

3054 On page 1133 line 38281 section `ilogb()`, add a new paragraph:

3055 ~~[MX]When the correct result is representable in the range of the return type, the returned~~

3056 ~~value shall be exact and shall be independent of the current rounding direction mode.[/MX]~~

3057 Ref F.10.3.5 para 3

3058 On page 1133 line 38282,38285,38288 section `ilogb()`, change:

3059 ~~[XSI]On XSI-conformant systems, a domain error shall occur[/XSI]~~

3060 to:

3061 ~~[XSI|MX]On XSI-conformant systems and on systems that support the IEC 60559 Floating-~~

3062 ~~Point option, a domain error shall occur[/XSI|MX]~~

3063 Ref 7.12.6.5 para 2

3064 On page 1133 line 38291 section `ilogb()`, change:

3065 If the correct value is greater than `{INT_MAX}`, ~~[MX]a domain error shall occur and[/MX]~~

3066 ~~an unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error~~

3067 ~~shall occur and `{INT_MAX}` shall be returned.[/XSI]~~

3068 If the correct value is less than `{INT_MIN}`, ~~[MX]a domain error shall occur and[/MX]~~

3069 ~~an unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error shall~~

3070 ~~occur and `{INT_MIN}` shall be returned.[/XSI]~~

3071 to:

3072 If the correct value is greater than `{INT_MAX}` or less than `{INT_MIN}`, an unspecified

3073 value shall be returned. ~~[XSI]On XSI-conformant systems, a domain error shall occur and~~

3074 ~~`{INT_MAX}` or `{INT_MIN}`, respectively, shall be returned;[/XSI] [MX]if the IEC 60559~~

3075 ~~Floating-Point option is supported, a domain error shall occur;[/MX] otherwise, a domain~~

3076 ~~error or range error may occur.~~

3077 Ref F.10.3.5 para 3

3078 On page 1133 line 38300 section `ilogb()`, change:

3079 ~~[XSI]The `x` argument is zero, NaN, or $\pm\text{Inf}$.[/XSI]~~

3080 to:

3081 [XSI|MX]The *x* argument is zero, NaN, or $\pm\text{Inf}$.[/XSI|MX]

3082 Ref F.10.11 para 1

3083 On page 1174 line 39604 section `isgreater()`,
3084 and page 1175 line 39642 section `isgreaterequal()`,
3085 and page 1177 line 39708 section `isless()`,
3086 and page 1178 line 39746 section `islessequal()`,
3087 and page 1179 line 39784 section `islessgreater()`, add a new paragraph:

3088 [MX]Relational operators and their corresponding comparison macros shall produce
3089 equivalent result values, even if argument values are represented in wider formats. Thus,
3090 comparison macro arguments represented in formats wider than their semantic types shall
3091 not be converted to the semantic types, unless the wide evaluation method converts operands
3092 of relational operators to their semantic types. The standard wide evaluation methods
3093 characterized by `FLT_EVAL_METHOD` equal to 1 or 2 (see [xref to <float.h>]) do not
3094 convert operands of relational operators to their semantic types. [/MX]

3095 (The editors may wish to merge the pages for the above interfaces to reduce duplication – they have
3096 duplicate APPLICATION USAGE as well.)

3097 Ref 7.30.2.2.1 para 4

3098 On page 1202 line 40411 section `iswctype()`, remove the CX shading from:

3099 If *charclass* is (`wctype_t`)0, these functions shall return 0.

3100 Ref 7.17.3.1

3101 On page 1229 line 41126 insert a new `kill_dependency()` section:

3102 NAME

3103 `kill_dependency` — terminate a dependency chain

3104 SYNOPSIS

```
3105 #include <stdatomic.h>  
3106 type kill_dependency(type y);
```

3107 DESCRIPTION

3108 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3109 Any conflict between the requirements described here and the ISO C standard is
3110 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard. [/CX]

3111 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
3112 `<stdatomic.h>` header nor support this macro.

3113 The `kill_dependency()` macro shall terminate a dependency chain (see [xref to XBD 4.12.1
3114 Memory Ordering]). The argument shall not carry a dependency to the return value.

3115 RETURN VALUE

3116 The `kill_dependency()` macro shall return the value of *y*.

3117 ERRORS

3118 No errors are defined.

3119 **EXAMPLES**

3120 None.

3121 **APPLICATION USAGE**

3122 None.

3123 **RATIONALE**

3124 None.

3125 **FUTURE DIRECTIONS**

3126 None.

3127 **SEE ALSO**

3128 XBD Section 4.12.1, <stdatomic.h>

3129 **CHANGE HISTORY**

3130 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3131 Ref 7.12.8.3, 7.1.4 para 5

3132 On page 1241 line 41433 section `lgamma()`, change:

3133 [CX]These functions need not be thread-safe.[/CX]

3134 to:

3135 [XSI]If concurrent calls are made to these functions, the value of *signgam* is indeterminate.[/
3136 XSI]

3137 Ref 7.12.8.3, 7.1.4 para 5

3138 On page 1242 line 41464 section `lgamma()`, add a new paragraph to APPLICATION USAGE:

3139 If the value of *signgam* will be obtained after a call to *lgamma()*, *lgammaf()*, or *lgammal()*,
3140 in order to ensure that the value will not be altered by another call in a different thread,
3141 applications should either restrict calls to these functions to be from a single thread or use a
3142 lock such as a mutex or spin lock to protect a critical section starting before the function call
3143 and ending after the value of *signgam* has been obtained.

3144 Ref 7.12.8.3, 7.1.4 para 5

3145 On page 1242 line 41466 section `lgamma()`, change RATIONALE from:

3146 None.

3147 to:

3148 Earlier versions of this standard did not require *lgamma()*, *lgammaf()*, and *lgammal()* to be
3149 thread-safe because *signgam* was a global variable. They are now required to be thread-safe
3150 to align with the ISO C standard (which, since the introduction of threads in 2011, requires
3151 that they avoid data races), with the exception that they need not avoid data races when
3152 storing a value in the *signgam* variable. Since *signgam* is not specified by the ISO C
3153 standard, this exception is not a conflict with that standard.

3154 Ref 7.11.2.1, 7.1.4 para 5
3155 On page 1262 line 42124 section `localeconv()`, change:

3156 [CX]The `localeconv()` function need not be thread-safe.[/CX]

3157 to:

3158 The `localeconv()` function need not be thread-safe; however, `localeconv()` shall avoid data
3159 races with all other functions.

3160 Ref 7.27.3, 7.1.4 para 5
3161 On page 1265 line 42217 section `localtime()`, change:

3162 [CX]The `localtime()` function need not be thread-safe.[/CX]

3163 to:

3164 The `localtime()` function need not be thread-safe; however, `localtime()` shall avoid data races
3165 with all functions other than itself, `asctime()`, `ctime()` and `gmtime()`.

3166 Ref F.10.3.11 para 2
3167 On page 1280 line 42723 section `logb()`, add a new paragraph:

3168 [MX]The returned value shall be exact and shall be independent of the current rounding
3169 direction mode.[/MX]

3170 Ref 7.13.2.1 para 1
3171 On page 1283 line 42780 section `longjmp()`, change:

3172 `void longjmp(jmp_buf env, int val);`

3173 to:

3174 `_Noreturn void longjmp(jmp_buf env, int val);`

3175 Ref 7.13.2.1 para 2
3176 On page 1283 line 42804 section `longjmp()`, remove the CX shading from:

3177 The effect of a call to `longjmp()` where initialization of the **jmp_buf** structure was not
3178 performed in the calling thread is undefined.

3179 Ref 7.13.2.1 para 4
3180 On page 1283 line 42807 section `longjmp()`, change:

3181 After `longjmp()` is completed, program execution continues ...

3182 to:

3183 After `longjmp()` is completed, thread execution shall continue ...

3184 Ref 7.22.3 para 1
3185 On page 1295 line 43144 section `malloc()`, change:

3186 a pointer to any type of object

3187 to:

3188 a pointer to any type of object with a fundamental alignment requirement

3189 Ref 7.22.3 para 2

3190 On page 1295 line 43150 section `malloc()`, add a new paragraph:

3191 For purposes of determining the existence of a data race, `malloc()` shall behave as though it
3192 accessed only memory locations accessible through its argument and not other static
3193 duration storage. The function may, however, visibly modify the storage that it allocates.
3194 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV]
3195 [CX]`reallocarray()`,[/CX] and `realloc()` that allocate or deallocate a particular region of
3196 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
3197 deallocation call shall synchronize with the next allocation (if any) in this order.

3198 Ref 7.22.3.1

3199 On page 1295 line 43171 section `malloc()`, add `aligned_alloc` to the SEE ALSO section.

3200 Ref 7.22.7.1 para 2

3201 On page 1297 line 43194 section `mblen()`, change:

3202 `mbtowc((wchar_t *)0, s, n);`

3203 to:

3204 `mbtowc((wchar_t *)0, (const char *)0, 0);`

3205 `mbtowc((wchar_t *)0, s, n);`

3206 Ref 7.22.7 para 1

3207 On page 1297 line 43198 section `mblen()`, change:

3208 this function shall be placed into its initial state by a call for which

3209 to:

3210 this function shall be placed into its initial state at program startup and can be returned to
3211 that state by a call for which

3212 Ref 7.22.7 para 1, 7.1.4 para 5

3213 On page 1297 line 43206 section `mblen()`, change:

3214 [CX]The `mblen()` function need not be thread-safe.[/CX]

3215 to:

3216 The `mblen()` function need not be thread-safe; however, it shall avoid data races with all
3217 other functions.

3218 Ref 7.29.6.3 para 1, 7.1.4 para 5

3219 On page 1299 line 43254 section `mbrlen()`, change:

3220 [CX]The *mbrlen()* function need not be thread-safe if called with a NULL *ps*
3221 argument.[/CX]

3222 to:

3223 If called with a null *ps* argument, the *mbrlen()* function need not be thread-safe; however,
3224 such calls shall avoid data races with calls to *mbrlen()* with a non-null argument and with
3225 calls to all other functions.

3226 Ref 7.28.1, 7.1.4 para 5

3227 On page 1301 line 43296 insert a new *mbrtoc16()* section:

3228 **NAME**

3229 *mbrtoc16*, *mbrtoc32* — convert a character to a Unicode character code (restartable)

3230 **SYNOPSIS**

3231 `#include <uchar.h>`

3232 `size_t mbrtoc16(char16_t *restrict pc16, const char *restrict s,`
3233 `size_t n, mbstate_t *restrict ps);`

3234 `size_t mbrtoc32(char32_t *restrict pc32, const char *restrict s,`
3235 `size_t n, mbstate_t *restrict ps);`

3236 **DESCRIPTION**

3237 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3238 Any conflict between the requirements described here and the ISO C standard is
3239 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3240 If *s* is a null pointer, the *mbrtoc16()* function shall be equivalent to the call:

3241 `mbrtoc16(NULL, "", 1, ps)`

3242 In this case, the values of the parameters *pc16* and *n* are ignored.

3243 If *s* is not a null pointer, the *mbrtoc16()* function shall inspect at most *n* bytes beginning with
3244 the byte pointed to by *s* to determine the number of bytes needed to complete the next
3245 character (including any shift sequences). If the function determines that the next character
3246 is complete and valid, it shall determine the values of the corresponding wide characters and
3247 then, if *pc16* is not a null pointer, shall store the value of the first (or only) such character in
3248 the object pointed to by *pc16*. Subsequent calls shall store successive wide characters
3249 without consuming any additional input until all the characters have been stored. If the
3250 corresponding wide character is the null wide character, the resulting state described shall be
3251 the initial conversion state.

3252 If *ps* is a null pointer, the *mbrtoc16()* function shall use its own internal **mbstate_t** object,
3253 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
3254 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current
3255 conversion state of the associated character sequence.

3256 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

3257 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

3258 The *mbrtoc32()* function shall behave the same way as *mbrtoc16()* except that the first
3259 parameter shall point to an object of type **char32_t** instead of **char16_t**. References to *pc16*
3260 in the above description shall apply as if they were *pc32* when they are being read as
3261 describing *mbrtoc32()*.

3262 If called with a null *ps* argument, the *mbrtoc16()* function need not be thread-safe; however,
3263 such calls shall avoid data races with calls to *mbrtoc16()* with a non-null argument and with
3264 calls to all other functions.

3265 If called with a null *ps* argument, the *mbrtoc32()* function need not be thread-safe; however,
3266 such calls shall avoid data races with calls to *mbrtoc32()* with a non-null argument and with
3267 calls to all other functions.

3268 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
3269 calls *mbrtoc16()* or *mbrtoc32()* with a null pointer for *ps*.

3270 RETURN VALUE

3271 These functions shall return the first of the following that applies:

3272 0 If the next *n* or fewer bytes complete the character that corresponds to the null
3273 wide character (which is the value stored).

3274 between 1 and *n* inclusive

3275 If the next *n* or fewer bytes complete a valid character (which is the value
3276 stored); the value returned shall be the number of bytes that complete the
3277 character.

3278 (**size_t**)-3 If the next character resulting from a previous call has been stored, in which
3279 case no bytes from the input shall be consumed by the call.

3280 (**size_t**)-2 If the next *n* bytes contribute to an incomplete but potentially valid character,
3281 and all *n* bytes have been processed (no value is stored). When *n* has at least
3282 the value of the {MB_CUR_MAX} macro, this case can only occur if *s*
3283 points at a sequence of redundant shift sequences (for implementations with
3284 state-dependent encodings).

3285 (**size_t**)-1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
3286 contribute to a complete and valid character (no value is stored). In this case,
3287 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

3288 ERRORS

3289 These function shall fail if:

3290 [EILSEQ] An invalid character sequence is detected. [CX]In the POSIX locale
3291 an [EILSEQ] error cannot occur since all byte values are valid
3292 characters.[/CX]

3293 These functions may fail if:

3294 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

3295 EXAMPLES

3296 None.

3297 **APPLICATION USAGE**

3298 None.

3299 **RATIONALE**

3300 None.

3301 **FUTURE DIRECTIONS**

3302 None.

3303 **SEE ALSO**

3304 *c16rtomb*

3305 XBD <**uchar.h**>

3306 **CHANGE HISTORY**

3307 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3308 Ref 7.29.6.3 para 1, 7.1.4 para 5

3309 On page 1301 line 43322 section *mbrtowc()*, change:

3310 [CX]The *mbrtowc()* function need not be thread-safe if called with a NULL *ps*
3311 argument.[/CX]

3312 to:

3313 If called with a null *ps* argument, the *mbrtowc()* function need not be thread-safe; however,
3314 such calls shall avoid data races with calls to *mbrtowc()* with a non-null argument and with
3315 calls to all other functions.

3316 Ref 7.29.6.4 para 1, 7.1.4 para 5

3317 On page 1304 line 43451 section *mbsrtowcs()*, change:

3318 [CX]The *mbsnrtowcs()* and *mbsrtowcs()* functions need not be thread-safe if called with a
3319 NULL *ps* argument.[/CX]

3320 to:

3321 [CX]If called with a null *ps* argument, the *mbsnrtowcs()* function need not be thread-safe;
3322 however, such calls shall avoid data races with calls to *mbsnrtowcs()* with a non-null
3323 argument and with calls to all other functions.[/CX]

3324 If called with a null *ps* argument, the *mbsrtowcs()* function need not be thread-safe;
3325 however, such calls shall avoid data races with calls to *mbsrtowcs()* with a non-null
3326 argument and with calls to all other functions.

3327 Ref 7.22.7 para 1

3328 On page 1308 line 43557 section *mbtowc()*, change:

3329 this function is placed into its initial state by a call for which

3330 to:

3331 this function shall be placed into its initial state at program startup and can be returned to
3332 that state by a call for which

3333 Ref 7.22.7 para 1, 7.1.4 para 5
3334 On page 1308 line 43567 section `mbtowc()`, change:

3335 [CX]The `mbtowc()` function need not be thread-safe.[/CX]

3336 to:

3337 The `mbtowc()` function need not be thread-safe; however, it shall avoid data races with all
3338 other functions.

3339 Ref 7.24.5.1 para 2
3340 On page 1311 line 43642 section `memchr()`, change:

3341 Implementations shall behave as if they read the memory byte by byte from the beginning of
3342 the bytes pointed to by `s` and stop at the first occurrence of `c` (if it is found in the initial `n`
3343 bytes).

3344 to:

3345 The implementation shall behave as if it reads the bytes sequentially and stops as soon as a
3346 matching byte is found.

3347 Ref F.10.3.12 para 2
3348 On page 1346 line 44854 section `modf()`, add a new paragraph:

3349 [MX]The returned value shall be exact and shall be independent of the current rounding
3350 direction mode.[/MX]

3351 Ref 7.26.4
3352 On page 1384 line 46032 insert the following new `mtx_*` sections:

3353 **NAME**

3354 `mtx_destroy`, `mtx_init` — destroy and initialize a mutex

3355 **SYNOPSIS**

3356 `#include <threads.h>`

3357 `void mtx_destroy(mtx_t *mtx);`
3358 `int mtx_init(mtx_t *mtx, int type);`

3359 **DESCRIPTION**

3360 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3361 Any conflict between the requirements described here and the ISO C standard is
3362 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3363 The `mtx_destroy()` function shall release any resources used by the mutex pointed to by `mtx`.
3364 A destroyed mutex object can be reinitialized using `mtx_init()`; the results of otherwise
3365 referencing the object after it has been destroyed are undefined. It shall be safe to destroy an

3366 initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that
3367 another thread is attempting to lock, or a mutex that is being used in a *cond_timedwait()* or
3368 *cond_wait()* call by another thread, results in undefined behavior. The behavior is undefined if
3369 the value specified by the *mtx* argument to *mtx_destroy()* does not refer to an initialized
3370 mutex.

3371 The *mtx_init()* function shall initialize a mutex object with properties indicated by *type*,
3372 whose valid values include:

3373 *mtx_plain* for a simple non-recursive mutex,

3374 *mtx_timed* for a non-recursive mutex that supports timeout,

3375 *mtx_plain* | *mtx_recursive* for a simple recursive mutex, or

3376 *mtx_timed* | *mtx_recursive* for a recursive mutex that supports timeout.

3377 If the *mtx_init()* function succeeds, it shall set the mutex pointed to by *mtx* to a value that
3378 uniquely identifies the newly initialized mutex. Upon successful initialization, the state of
3379 the mutex becomes initialized and unlocked. Attempting to initialize an already initialized
3380 mutex results in undefined behavior.

3381 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
3382 further requirements.

3383 These functions shall not be affected if the calling thread executes a signal handler during
3384 the call.[/CX]

3385 **RETURN VALUE**

3386 The *mtx_destroy()* function shall not return a value.

3387 The *mtx_init()* function shall return *thrd_success* on success or *thrd_error* if the
3388 request could not be honored.

3389 **ERRORS**

3390 No errors are defined.

3391 **EXAMPLES**

3392 None.

3393 **APPLICATION USAGE**

3394 A mutex can be destroyed immediately after it is unlocked. However, since attempting to
3395 destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that
3396 is being used in a *cond_timedwait()* or *cond_wait()* call by another thread results in undefined
3397 behavior, care must be taken to ensure that no other thread may be referencing the mutex.

3398 **RATIONALE**

3399 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3400 B.2.3].

3401 **FUTURE DIRECTIONS**

3402 None.

3403 **SEE ALSO**

3404 *mtx_lock*

3405 XBD <**threads.h**>

3406 **CHANGE HISTORY**

3407 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3408 **NAME**

3409 *mtx_lock*, *mtx_timedlock*, *mtx_trylock*, *mtx_unlock* — lock and unlock a mutex

3410 **SYNOPSIS**

3411 `#include <threads.h>`

3412 `int mtx_lock(mtx_t *mtx);`

3413 `int mtx_timedlock(mtx_t * restrict mtx,`

3414 `const struct timespec * restrict ts);`

3415 `int mtx_trylock(mtx_t *mtx);`

3416 `int mtx_unlock(mtx_t *mtx);`

3417 **DESCRIPTION**

3418 [CX] The functionality described on this reference page is aligned with the ISO C standard.

3419 Any conflict between the requirements described here and the ISO C standard is

3420 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3421 The *mtx_lock()* function shall block until it locks the mutex pointed to by *mtx*. If the mutex

3422 is non-recursive, the application shall ensure that it is not already locked by the calling

3423 thread.

3424 The *mtx_timedlock()* function shall block until it locks the mutex pointed to by *mtx* or until

3425 after the `TIME_UTC` -based calendar time pointed to by *ts*. The application shall ensure that

3426 the specified mutex supports timeout. [CX]Under no circumstance shall the function fail

3427 with a timeout if the mutex can be locked immediately. The validity of the *ts* parameter need

3428 not be checked if the mutex can be locked immediately.[/CX]

3429 The *mtx_trylock()* function shall endeavor to lock the mutex pointed to by *mtx*. If the mutex

3430 is already locked (by any thread, including the current thread), the function shall return

3431 without blocking. If the mutex is recursive and the mutex is currently owned by the calling

3432 thread, the mutex lock count (see below) shall be incremented by one and the *mtx_trylock()*

3433 function shall immediately return success.

3434 [CX]These functions shall not be affected if the calling thread executes a signal handler

3435 during the call; if a signal is delivered to a thread waiting for a mutex, upon return from the

3436 signal handler the thread shall resume waiting for the mutex as if it was not

3437 interrupted.[/CX]

3438 If a call to *mtx_lock()*, *mtx_timedlock()* or *mtx_trylock()* locks the mutex, prior calls to

3439 *mtx_unlock()* on the same mutex shall synchronize with this lock operation.

3440 The *mtx_unlock()* function shall unlock the mutex pointed to by *mtx* . The application shall

3441 ensure that the mutex pointed to by *mtx* is locked by the calling thread. [CX]If there are

3442 threads blocked on the mutex object referenced by *mtx* when *mtx_unlock()* is called,
3443 resulting in the mutex becoming available, the scheduling policy shall determine which
3444 thread shall acquire the mutex.[/CX]

3445 A recursive mutex shall maintain the concept of a lock count. When a thread successfully
3446 acquires a mutex for the first time, the lock count shall be set to one. Every time a thread
3447 relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks
3448 the mutex, the lock count shall be decremented by one. When the lock count reaches zero,
3449 the mutex shall become available for other threads to acquire.

3450 For purposes of determining the existence of a data race, mutex lock and unlock operations
3451 on mutexes of type **mtx_t** behave as atomic operations. All lock and unlock operations on a
3452 particular mutex occur in some particular total order.

3453 If *mtx* does not refer to an initialized mutex object, the behavior of these functions is
3454 undefined.

3455 RETURN VALUE

3456 The *mtx_lock()* and *mtx_unlock()* functions shall return *thrd_success* on success, or
3457 *thrd_error* if the request could not be honored.

3458 The *mtx_timedlock()* function shall return *thrd_success* on success, or *thrd_timedout*
3459 if the time specified was reached without acquiring the requested resource, or *thrd_error*
3460 if the request could not be honored.

3461 The *mtx_trylock()* function shall return *thrd_success* on success, or *thrd_busy* if the
3462 resource requested is already in use, or *thrd_error* if the request could not be honored.
3463 The *mtx_trylock()* function can spuriously fail to lock an unused resource, in which case it
3464 shall return *thrd_busy*.

3465 ERRORS

3466 See RETURN VALUE.

3467 EXAMPLES

3468 None.

3469 APPLICATION USAGE

3470 None.

3471 RATIONALE

3472 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3473 B.2.3].

3474 Since **<pthread.h>** has no equivalent of the *mtx_timed* mutex property, if the **<threads.h>**
3475 interfaces are implemented as a thin wrapper around **<pthread.h>** interfaces (meaning
3476 **mtx_t** and **pthread_mutex_t** are the same type), all mutexes support timeout and
3477 *mtx_timedlock()* will not fail for a mutex that was not initialized with *mtx_timed*.
3478 Alternatively, implementations can use a less thin wrapper where **mtx_t** contains additional
3479 properties that are not held in **pthread_mutex_t** in order to be able to return a failure
3480 indication from *mtx_timedlock()* calls where the mutex was not initialized with
3481 *mtx_timed*.

3482 **FUTURE DIRECTIONS**

3483 None.

3484 **SEE ALSO**

3485 *mtx_destroy*, *timespec_get*

3486 XBD Section 4.12.2, <**threads.h**>

3487 **CHANGE HISTORY**

3488 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3489 Ref F.10.8.2 para 2

3490 On page 1388 line 46143 section *nan()*, add a new paragraph:

3491 [MX]The returned value shall be exact and shall be independent of the current rounding
3492 direction mode.[/MX]

3493 Ref F.10.8.3 para 2, F.10.8.4 para 2

3494 On page 1395 line 46388 section *nextafter()*, add a new paragraph:

3495 [MX]Even though underflow or overflow can occur, the returned value shall be independent
3496 of the current rounding direction mode.[/MX]

3497 Ref 7.22.3 para 2

3498 On page 1448 line 48069 section *posix_memalign()*, add a new (unshaded) paragraph:

3499 For purposes of determining the existence of a data race, *posix_memalign()* shall behave as
3500 though it accessed only memory locations accessible through its arguments and not other
3501 static duration storage. The function may, however, visibly modify the storage that it
3502 allocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, *posix_memalign()*, *realloc()*,
3503 and *reallocarray()* that allocate or deallocate a particular region of memory shall occur in a
3504 single total order (see [xref to XBD 4.12.1]), and each such deallocation call shall
3505 synchronize with the next allocation (if any) in this order.

3506 Ref 7.22.3.1

3507 On page 1449 line 48107 section *posix_memalign()*, add *aligned_alloc* to the SEE ALSO section.

3508 Ref F.10.4.4 para 1

3509 On page 1548 line 50724 section *pow()*, change:

3510 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 , a pole error shall
3511 occur and *pow()*, *powf()*, and *powl()* shall return \pm HUGE_VAL, \pm HUGE_VALF, and
3512 \pm HUGE_VALL, respectively if y is an odd integer, or HUGE_VAL, HUGE_VALF, and
3513 HUGE_VALL, respectively if y is not an odd integer.

3514 to:

3515 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 :

3516 • if y is an odd integer, a pole error shall occur and *pow()*, *powf()*, and *powl()* shall

3517 return \pm HUGE_VAL, \pm HUGE_VALF, and \pm HUGE_VALL, respectively;

3518 • if y is finite and is not an odd integer, a pole error shall occur and $pow()$, $powf()$, and
3519 $powl()$ shall return HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively;

3520 • if y is $-\text{Inf}$, a pole error may occur and $pow()$, $powf()$, and $powl()$ shall return
3521 HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

3522 Ref 7.26

3523 On page 1603 line 52244 section `pthread_cancel()`, add a new paragraph:

3524 If *thread* refers to a thread that was created using *thrd_create()*, the behavior is undefined.

3525 Ref 7.26.5.6

3526 On page 1603 line 52277 section `pthread_cancel()`, add a new RATIONALE paragraph:

3527 Use of *pthread_cancel()* to cancel a thread that was created using *thrd_create()* is undefined
3528 because *thrd_join()* has no way to indicate a thread was cancelled. The standard developers
3529 considered adding a `thrd_cancelled` enumeration constant that *thrd_join()* would return in
3530 this case. However, this return would be unexpected in code that is written to conform to the
3531 ISO C standard, and it would also not solve the problem that threads which use only ISO C
3532 `<threads.h>` interfaces (such as ones created by third party libraries written to conform to
3533 the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not
3534 provide cancellation cleanup handlers.

3535 Ref 7.26.5.5

3536 On page 1639 line 53422 section `pthread_exit()`, change:

3537 `void pthread_exit(void *value_ptr);`

3538 to:

3539 `_Noreturn void pthread_exit(void *value_ptr);`

3540 Ref 7.26.6

3541 On page 1639 line 53427 section `pthread_exit()`, change:

3542 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3543 specific data, appropriate destructor functions shall be called in an unspecified order.

3544 to:

3545 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3546 specific data (whether associated with key type `tss_t` or `pthread_key_t`), appropriate
3547 destructor functions shall be called in an unspecified order.

3548 Ref 7.26.5.5

3549 On page 1639 line 53432 section `pthread_exit()`, change:

3550 An implicit call to *pthread_exit()* is made when a thread other than the thread in which
3551 *main()* was first invoked returns from the start routine that was used to create it.

3552 to:

3553 An implicit call to *pthread_exit()* is made when a thread that was not created using
3554 *thrd_create()*, and is not the thread in which *main()* was first invoked, returns from the start
3555 routine that was used to create it.

3556 Ref 7.26.5.5

3557 On page 1639 line 53451 section *pthread_exit()*, change APPLICATION USAGE from:

3558 None.

3559 to:

3560 Calls to *pthread_exit()* should not be made from threads created using *thrd_create()*, as their
3561 exit status has a different type (**int** instead of **void ***). If *pthread_exit()* is called from the
3562 initial thread and it is not the last thread to terminate, other threads should not try to obtain
3563 its exit status using *thrd_join()*.

3564 Ref 7.26.5.5

3565 On page 1639 line 53453 section *pthread_exit()*, change:

3566 The normal mechanism by which a thread terminates is to return from the routine that was
3567 specified in the *pthread_create()* call that started it.

3568 to:

3569 The normal mechanism by which a thread that was started using *pthread_create()* terminates
3570 is to return from the routine that was specified in the *pthread_create()* call that started it.

3571 Ref 7.26.5.5, 7.26.6

3572 On page 1640 line 53470 section *pthread_exit()*, add *pthread_key_create*, *thrd_create*, *thrd_exit* and
3573 *tss_create* to the SEE ALSO section.

3574 Ref 7.26.5.5

3575 On page 1649 line 53748 section *pthread_join()*, add a new paragraph:

3576 If *thread* refers to a thread that was created using *thrd_create()* and the thread terminates, or
3577 has already terminated, by returning from its start routine, the behavior of *pthread_join()* is
3578 undefined. If *thread* refers to a thread that terminates, or has already terminated, by calling
3579 *thrd_exit()*, the behavior of *pthread_join()* is undefined.

3580 Ref 7.26.5.5

3581 On page 1651 line 53819 section *pthread_join()*, add a new RATIONALE paragraph:

3582 The *pthread_join()* function cannot be used to obtain the exit status of a thread that was
3583 created using *thrd_create()* and which terminates by returning from its start routine, or of a
3584 thread that terminates by calling *thrd_exit()*, because such threads have an **int** exit status,
3585 instead of the **void *** that *pthread_join()* returns via its *value_ptr* argument.

3586 Ref 7.22.4.7

3587 On page 1765 line 57040 insert the following new *quick_exit()* section:

3588 **NAME**

3589 `quick_exit` — terminate a process

3590 **SYNOPSIS**

3591 `#include <stdlib.h>`

3592 `_Noreturn void quick_exit(int status);`

3593 **DESCRIPTION**

3594 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3595 Any conflict between the requirements described here and the ISO C standard is
3596 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3597 The `quick_exit()` function shall cause normal process termination to occur. It shall not call
3598 functions registered with `atexit()` nor any registered signal handlers. If a process calls the
3599 `quick_exit()` function more than once, or calls the `exit()` function in addition to the
3600 `quick_exit()` function, the behavior is undefined. If a signal is raised while the `quick_exit()`
3601 function is executing, the behavior is undefined.

3602 The `quick_exit()` function shall first call all functions registered by `at_quick_exit()`, in the
3603 reverse order of their registration, except that a function is called after any previously
3604 registered functions that had already been called at the time it was registered. If, during the
3605 call to any such function, a call to the `longjmp()` [CX] or `siglongjmp()`[/CX] function is made
3606 that would terminate the call to the registered function, the behavior is undefined.

3607 If a function registered by a call to `at_quick_exit()` fails to return, the remaining registered
3608 functions shall not be called and the rest of the `quick_exit()` processing shall not be
3609 completed.

3610 Finally, the `quick_exit()` function shall terminate the process as if by a call to `_Exit(status)`.

3611 **RETURN VALUE**

3612 The `quick_exit()` function does not return.

3613 **ERRORS**

3614 No errors are defined.

3615 **EXAMPLES**

3616 None.

3617 **APPLICATION USAGE**

3618 None.

3619 **RATIONALE**

3620 None.

3621 **FUTURE DIRECTIONS**

3622 None.

3623 **SEE ALSO**

3624 `_Exit`, `at_quick_exit`, `atexit`, `exit`

3625 XBD `<stdlib.h>`

3626 **CHANGE HISTORY**

3627 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3628 Ref 7.22.2.1 para 3, 7.1.4 para 5

3629 On page 1767 line 57095 section `rand()`, change:

3630 [CX]The `rand()` function need not be thread-safe.[/CX]

3631 to:

3632 The `rand()` function need not be thread-safe; however, `rand()` shall avoid data races with all
3633 functions other than non-thread-safe pseudo-random sequence generation functions.

3634 Ref 7.22.2.2 para 3, 7.1.4 para 5

3635 On page 1767 line 57105 section `rand()`, add a new paragraph:

3636 The `srand()` function need not be thread-safe; however, `srand()` shall avoid data races with
3637 all functions other than non-thread-safe pseudo-random sequence generation functions.

3638 Ref 7.22.3 para 1,2; 7.22.3.5 para 2,3,4; 7.31.12 para 2

3639 On page 1788 line 57862-57892 section `realloc()`, after applying bugs 374 and 1218 replace the

3640 DESCRIPTION and RETURN VALUE sections with:

3641 **DESCRIPTION**

3642 For `realloc()`: [CX] The functionality described on this reference page is aligned with the
3643 ISO C standard. Any conflict between the requirements described here and the ISO C
3644 standard is unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3645 The `realloc()` function shall deallocate the old object pointed to by `ptr` and return a pointer to
3646 a new object that has the size specified by `size`. The contents of the new object shall be the
3647 same as that of the old object prior to deallocation, up to the lesser of the new and old sizes.
3648 Any bytes in the new object beyond the size of the old object have indeterminate values.

3649 [CX]The `reallocarray()` function shall be equivalent to the call `realloc(ptr, nelem *
3650 elsize)` except that overflow in the multiplication shall be an error.[/CX]

3651 If `ptr` is a null pointer, `realloc()` [CX]or `reallocarray()`[/CX] shall be equivalent to `malloc()`
3652 function for the specified size. Otherwise, if `ptr` does not match a pointer returned earlier by
3653 `aligned_alloc()`, `calloc()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV] `realloc()`,
3654 [CX]`reallocarray()`, or a function in POSIX.1-20xx that allocates memory as if by `malloc()`,
3655 [/CX] or if the space has been deallocated by a call to `free()`, [CX]`reallocarray()`,[/CX] or
3656 `realloc()`, the behavior is undefined.

3657 If `size` is non-zero and memory for the new object is not allocated, the old object shall not be
3658 deallocated.

3659 The order and contiguity of storage allocated by successive calls to `realloc()` [CX]or
3660 `reallocarray()`[/CX] is unspecified. The pointer returned if the allocation succeeds shall be
3661 suitably aligned so that it may be assigned to a pointer to any type of object with a
3662 fundamental alignment requirement and then used to access such an object in the space

3663 allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a
3664 pointer to an object disjoint from any other object. The pointer returned shall point to the
3665 start (lowest byte address) of the allocated space. If the space cannot be allocated, a null
3666 pointer shall be returned.

3667 For purposes of determining the existence of a data race, *realloc()* [CX] or
3668 *reallocarray()*[/CX] shall behave as though it accessed only memory locations accessible
3669 through its arguments and not other static duration storage. The function may, however,
3670 visibly modify the storage that it allocates or deallocates. Calls to *aligned_alloc()*, *calloc()*,
3671 *free()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] [CX]*reallocarray()*,[/CX] and *realloc()*
3672 that allocate or deallocate a particular region of memory shall occur in a single total order
3673 (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the next
3674 allocation (if any) in this order.

3675 RETURN VALUE

3676 Upon successful completion, *realloc()* [CX] and *reallocarray()*[/CX] shall return a pointer to
3677 the new object (which can have the same value as a pointer to the old object), or a null
3678 pointer if the new object has not been allocated.

3679 [OB]If size is zero, [/OB]
3680 [OB CX] or either *nelem* or *elsize* is 0, [/OB CX]
3681 [OB] either:

- 3682 • A null pointer shall be returned [CX] and, if *ptr* is not a null pointer, *errno* shall be set
3683 to [EINVAL].[/CX]
- 3684 • A pointer to the allocated space shall be returned, and the memory object pointed to
3685 by *ptr* shall be freed. The application shall ensure that the pointer is not used to
3686 access an object. [/OB]

3687 If there is not enough available memory, *realloc()* [CX] and *reallocarray()*[/CX] shall return
3688 a null pointer [CX] and set *errno* to [ENOMEM] [/CX].

3689 Ref 7.22.3.5 para 3,4
3690 On page 1789 line 57899 section *realloc()*, change:

3691 The description of *realloc()* has been modified from previous versions of this standard to
3692 align with the ISO/IEC 9899: 1999 standard. Previous versions explicitly permitted a call to
3693 *realloc(p, 0)* to free the space pointed to by *p* and return a null pointer. While this behavior
3694 could be interpreted as permitted by this version of the standard, the C language committee
3695 have indicated that this interpretation is incorrect. Applications should assume that if
3696 *realloc()* returns a null pointer, the space pointed to by *p* has not been freed. Since this could
3697 lead to double-frees, implementations should also set *errno* if a null pointer actually
3698 indicates a failure, and applications should only free the space if *errno* was changed.

3699 to:

3700 The ISO C standard makes it implementation-defined whether a call to *realloc(p, 0)* frees the
3701 space pointed to by *p* if it returns a null pointer because memory for the new object was not
3702 allocated. POSIX.1 instead requires that implementations set *errno* if a null pointer is
3703 returned and the space has not been freed, and POSIX applications should only free the
3704 space if *errno* was changed.

3705 Ref 7.31.12 para 2
3706 On page 1789 line 57909-57912 section `realloc()`, change FUTURE DIRECTIONS to:

3707 The ISO C standard states that invoking `realloc()` with a `size` argument equal to zero is an
3708 obsolescent feature. This feature may be removed in a future version of this standard.

3709 Ref 7.22.3.1
3710 On page 1789 line 57914 section `realloc()`, add `aligned_alloc` to the SEE ALSO section.

3711 Ref F.10.7.2 para 2
3712 On page 1809 line 58638 section `remainder()`, add a new paragraph:

3713 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3714 Ref F.10.7.3 para 2
3715 On page 1814 line 58758 section `remquo()`, add a new paragraph:

3716 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3717 Ref F.10.6.6 para 3
3718 On page 1828 line 59258 section `round()`, add a new paragraph:

3719 [MX]These functions may raise the inexact floating-point exception for finite non-integer
3720 arguments.[/MX]

3721 Ref F.10.6.6 para 3
3722 On page 1828 line 59272 section `round()`, delete from APPLICATION USAGE:

3723 These functions may raise the inexact floating-point exception if the result differs in value
3724 from the argument.

3725 Ref F.10.3.13 para 2
3726 On page 1829 line 59306 section `scalbln()`, add a new paragraph:

3727 [MX]If the calculation does not overflow or underflow, the returned value shall be exact and
3728 shall be independent of the current rounding direction mode.[/MX]

3729 Ref 7.11.1.1 para 5
3730 On page 1903 line 61520 section `setlocale()`, change:

3731 [CX]The `setlocale()` function need not be thread-safe.[/CX]

3732 to:

3733 The `setlocale()` function need not be thread-safe; however, it shall avoid data races with all
3734 function calls that do not affect and are not affected by the global locale.

3735 Ref 7.13.2.1 para 1
3736 On page 1970 line 63497 section `siglongjmp()`, change:

3737

```
void siglongjmp(sigjmp_buf env, int val);
```


3738 to:

3739 `_Noreturn void siglongjmp(sigjmp_buf env, int val);`

3740 Ref 7.13.2.1 para 4

3741 On page 1970 line 63504 section `siglongjmp()`, change:

3742 *After `siglongjmp()` is completed, program execution shall continue ...*

3743 to:

3744 *After `siglongjmp()` is completed, thread execution shall continue ...*

3745 Ref 7.14.1.1 para 5

3746 On page 1971 line 63564 section `signal()`, change:

3747 with static storage duration

3748 to:

3749 with static or thread storage duration that is not a lock-free atomic object

3750 Ref 7.14.1.1 para 7

3751 On page 1972 line 63573 section `signal()`, add a new paragraph:

3752 [CX]The `signal()` function is required to be thread-safe. (See [xref to 2.9.1 Thread-Safety].)

3753 [/CX]

3754 Ref 7.14.1.1 para 7

3755 On page 1972 line 63591 section `signal()`, change RATIONALE from:

3756 None.

3757 to:

3758 The ISO C standard says that the use of `signal()` in a multi-threaded program results in

3759 undefined behavior. However, POSIX.1 has required `signal()` to be thread-safe since before

3760 threads were added to the ISO C standard.

3761 Ref F.10.4.5 para 1

3762 On page 2009 line 64624 section `sqrt()`, add:

3763 [MX]The returned value shall be dependent on the current rounding direction mode.[/MX]

3764 Ref 7.24.6.2 para 3, 7.1.4 para 5

3765 On page 2035 line 65231 section `strerror()`, change:

3766 [CX]The `strerror()` function need not be thread-safe.[/CX]

3767 to:

3768 The `strerror()` function need not be thread-safe; however, `strerror()` shall avoid data races

3769 with all other functions.

3770 Ref 7.22.1.3 para 10

3771 On page 2073 line 66514 section strtod(), change:

3772 If the correct value is outside the range of representable values

3773 to:

3774 If the correct value would cause an overflow and default rounding is in effect

3775 Ref 7.24.5.8 para 6, 7.1.4 para 5

3776 On page 2078 line 66674 section strtok(), change:

3777 [CX]The *strtok()* function need not be thread-safe.[/CX]

3778 to:

3779 The *strtok()* function need not be thread-safe; however, *strtok()* shall avoid data races with
3780 all other functions.

3781 Ref 7.22.4.8, 7.1.4 para 5

3782 On page 2107 line 67579 section system(), change:

3783 The *system()* function need not be thread-safe.

3784 to:

3785 [CX]If concurrent calls to *system()* are made from multiple threads, it is unspecified
3786 whether:

- 3787 • each call saves and restores the dispositions of the SIGINT and SIGQUIT signals
3788 independently, or
- 3789 • in a set of concurrent calls the dispositions in effect after the last call returns are
3790 those that were in effect on entry to the first call.

3791 If a thread is cancelled while it is in a call to *system()*, it is unspecified whether the child
3792 process is terminated and waited for, or is left running.[/CX]

3793 Ref 7.22.4.8, 7.1.4 para 5

3794 On page 2108 line 67627 section system(), change:

3795 Using the *system()* function in more than one thread in a process or when the SIGCHLD
3796 signal is being manipulated by more than one thread in a process may produce unexpected
3797 results.

3798 to:

3799 Although *system()* is required to be thread-safe, it is recommended that concurrent calls
3800 from multiple threads are avoided, since *system()* is not required to coordinate the saving
3801 and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of
3802 overlapping calls, and therefore the signals might end up being set to ignored after the last
3803 call returns. Applications should also avoid cancelling a thread while it is in a call to
3804 *system()* as the child process may be left running in that event. In addition, if another thread

3805 alters the disposition of the SIGCHLD signal, a call to *signal()* may produce unexpected
3806 results.

3807 Ref 7.22.4.8, 7.1.4 para 5
3808 On page 2109 line 67675 section *system()*, delete:

```
3809           #include <signal.h>
```

3810 Ref 7.22.4.8, 7.1.4 para 5
3811 On page 2109 line 67692,67696,67712 section *system()*, change *sigprocmask* to
3812 *pthread_sigmask*.

3813 Ref 7.22.4.8, 7.1.4 para 5
3814 On page 2110 line 67718 section *system()*, change:

3815 Note also that the above example implementation is not thread-safe. Implementations can
3816 provide a thread-safe *system()* function, but doing so involves complications such as how to
3817 restore the signal dispositions for SIGINT and SIGQUIT correctly if there are overlapping
3818 calls, and how to deal with cancellation. The example above would not restore the signal
3819 dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-
3820 safe implementation since canceling a non-thread-safe function results in undefined
3821 behavior (see Section 2.9.5.2, on page 518). To avoid leaking a process ID, a thread-safe
3822 implementation would need to terminate the child process when acting on a cancellation.

3823 to:

3824 Earlier versions of this standard did not require *system()* to be thread-safe because it alters
3825 the process-wide disposition of the SIGINT and SIGQUIT signals. It is now required to be
3826 thread-safe to align with the ISO C standard, which (since the introduction of threads in
3827 2011) requires that it avoids data races. However, the function is not required to coordinate
3828 the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set
3829 of overlapping calls, and the above example does not do so. The example also does not
3830 terminate and wait for the child process if the calling thread is cancelled, and so would leak
3831 a process ID in that event.

3832 Ref 7.26.5
3833 On page 2148 line 68796 insert the following new *thrd_**() sections:

3834 **NAME**

3835 *thrd_create* — thread creation

3836 **SYNOPSIS**

```
3837           #include <threads.h>
```

```
3838           int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

3839 **DESCRIPTION**

3840 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3841 Any conflict between the requirements described here and the ISO C standard is
3842 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3843 The *thrd_create()* function shall create a new thread executing *func(arg)*. If the *thrd_create()*
3844 function succeeds, it shall set the object pointed to by *thr* to the identifier of the newly

3845 created thread. (A thread's identifier might be reused for a different thread once the original
3846 thread has exited and either been detached or joined to another thread.) The completion of
3847 the *thrd_create()* function shall synchronize with the beginning of the execution of the new
3848 thread.

3849 [CX]The signal state of the new thread shall be initialized as follows:

- 3850 • The signal mask shall be inherited from the creating thread.
- 3851 • The set of signals pending for the new thread shall be empty.

3852 The thread-local current locale shall not be inherited from the creating thread.

3853 The floating-point environment shall be inherited from the creating thread.[/CX]

3854 [XSI] The alternate stack shall not be inherited from the creating thread.[/XSI]

3855 Returning from *func* shall have the same behavior as invoking *thrd_exit()* with the value
3856 returned from *func*.

3857 If *thrd_create()* fails, no new thread shall be created and the contents of the location
3858 referenced by *thr* are undefined.

3859 [CX]The *thrd_create()* function shall not be affected if the calling thread executes a signal
3860 handler during the call.[/CX]

3861 **RETURN VALUE**

3862 The *thrd_create()* function shall return *thrd_success* on success; or *thrd_nomem* if no
3863 memory could be allocated for the thread requested; or *thrd_error* if the request could not
3864 be honored, [CX]such as if the system-imposed limit on the total number of threads in a
3865 process {*PTHREAD_THREADS_MAX*} would be exceeded.[/CX]

3866 **ERRORS**

3867 See RETURN VALUE.

3868 **EXAMPLES**

3869 None.

3870 **APPLICATION USAGE**

3871 There is no requirement on the implementation that the ID of the created thread be available
3872 before the newly created thread starts executing. The calling thread can obtain the ID of the
3873 created thread through the *thr* argument of the *thrd_create()* function, and the newly created
3874 thread can obtain its ID by a call to *thrd_current()*.

3875 **RATIONALE**

3876 The *thrd_create()* function is not affected by signal handlers for the reasons stated in [xref to
3877 XRAT B.2.3].

3878 **FUTURE DIRECTIONS**

3879 None.

3880 **SEE ALSO**

3881 *pthread_create, thrd_current, thrd_detach, thrd_exit, thrd_join*

3882 XBD Section 4.12.2, <**threads.h**>

3883 **CHANGE HISTORY**

3884 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3885 **NAME**

3886 *thrd_current* — get the calling thread ID

3887 **SYNOPSIS**

3888 `#include <threads.h>`

3889 `thrd_t thrd_current(void);`

3890 **DESCRIPTION**

3891 [CX] The functionality described on this reference page is aligned with the ISO C standard.

3892 Any conflict between the requirements described here and the ISO C standard is

3893 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3894 The *thrd_current()* function shall identify the thread that called it.

3895 **RETURN VALUE**

3896 The *thrd_current()* function shall return the thread ID of the thread that called it.

3897 The *thrd_current()* function shall always be successful. No return value is reserved to

3898 indicate an error.

3899 **ERRORS**

3900 No errors are defined.

3901 **EXAMPLES**

3902 None.

3903 **APPLICATION USAGE**

3904 None.

3905 **RATIONALE**

3906 None.

3907 **FUTURE DIRECTIONS**

3908 None.

3909 **SEE ALSO**

3910 *pthread_self, thrd_create, thrd_equal*

3911 XBD Section 4.12.2, <**threads.h**>

3912 **CHANGE HISTORY**

3913 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3914 **NAME**

3915 `thrd_detach` — detach a thread

3916 **SYNOPSIS**

3917 `#include <threads.h>`

3918 `int thrd_detach(thrd_t thr);`

3919 **DESCRIPTION**

3920 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3921 Any conflict between the requirements described here and the ISO C standard is
3922 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3923 The `thrd_detach()` function shall change the thread *thr* from joinable to detached, indicating
3924 to the implementation that any resources allocated to the thread can be reclaimed when that
3925 thread terminates. The application shall ensure that the thread identified by *thr* has not been
3926 previously detached or joined with another thread.

3927 [CX]The `thrd_detach()` function shall not be affected if the calling thread executes a signal
3928 handler during the call.[/CX]

3929 **RETURN VALUE**

3930 The `thrd_detach()` function shall return `thrd_success` on success or `thrd_error` if the
3931 request could not be honored.

3932 **ERRORS**

3933 No errors are defined.

3934 **EXAMPLES**

3935 None.

3936 **APPLICATION USAGE**

3937 None.

3938 **RATIONALE**

3939 The `thrd_detach()` function is not affected by signal handlers for the reasons stated in [xref
3940 to XRAT B.2.3].

3941 **FUTURE DIRECTIONS**

3942 None.

3943 **SEE ALSO**

3944 `pthread_detach`, `thrd_create`, `thrd_join`

3945 XBD <**threads.h**>

3946 **CHANGE HISTORY**

3947 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3948 **NAME**

3949 `thrd_equal` — compare thread IDs

3950 **SYNOPSIS**

3951 `#include <threads.h>`

3952 `int thrd_equal(thrd_t thr0, thrd_t thr1);`

3953 **DESCRIPTION**

3954 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3955 Any conflict between the requirements described here and the ISO C standard is
3956 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3957 The *thrd_equal()* function shall determine whether the thread identified by *thr0* refers to the
3958 thread identified by *thr1*.

3959 [CX]The *thrd_equal()* function shall not be affected if the calling thread executes a signal
3960 handler during the call.[/CX]

3961 **RETURN VALUE**

3962 The *thrd_equal()* function shall return a non-zero value if *thr0* and *thr1* are equal; otherwise,
3963 zero shall be returned.

3964 If either *thr0* or *thr1* is not a valid thread ID [CX]and is not equal to PTHREAD_NULL
3965 (which is defined in **<pthread.h>**)[/CX], the behavior is undefined.

3966 **ERRORS**

3967 No errors are defined.

3968 **EXAMPLES**

3969 None.

3970 **APPLICATION USAGE**

3971 None.

3972 **RATIONALE**

3973 See the RATIONALE section for *pthread_equal()*.

3974 The *thrd_equal()* function is not affected by signal handlers for the reasons stated in [xref to
3975 XRAT B.2.3].

3976 **FUTURE DIRECTIONS**

3977 None.

3978 **SEE ALSO**

3979 *pthread_equal*, *thrd_current*

3980 XBD **<pthread.h>**, **<threads.h>**

3981 **CHANGE HISTORY**

3982 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3983 **NAME**

3984 *thrd_exit* — thread termination

3985 **SYNOPSIS**

3986 `#include <threads.h>`

3987 `_Noreturn void thrd_exit(int res);`

3988 **DESCRIPTION**

3989 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3990 Any conflict between the requirements described here and the ISO C standard is
3991 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3992 For every thread-specific storage key [CX](regardless of whether it has type `tss_t` or
3993 `pthread_key_t`[/CX] which was created with a non-null destructor and for which the value
3994 is non-null, `thrd_exit()` shall set the value associated with the key to a null pointer value and
3995 then invoke the destructor with its previous value. The order in which destructors are
3996 invoked is unspecified.

3997 If after this process there remain keys with both non-null destructors and values, the
3998 implementation shall repeat this process up to [CX]
3999 `{PTHREAD_DESTRUCTOR_ITERATIONS}`[/CX] times.

4000 Following this, the `thrd_exit()` function shall terminate execution of the calling thread and
4001 shall set its exit status to `res`. [CX]Thread termination shall not release any application
4002 visible process resources, including, but not limited to, mutexes and file descriptors, nor
4003 shall it perform any process-level cleanup actions, including, but not limited to, calling any
4004 `atexit()` routines that might exist.[/CX]

4005 An implicit call to `thrd_exit()` is made when a thread that was created using `thrd_create()`
4006 returns from the start routine that was used to create it (see [xref to `thrd_create()`]).

4007 [CX]The behavior of `thrd_exit()` is undefined if called from a destructor function that was
4008 invoked as a result of either an implicit or explicit call to `thrd_exit()`.[/CX]

4009 The process shall exit with an exit status of zero after the last thread has been terminated.
4010 The behavior shall be as if the implementation called `exit()` with a zero argument at thread
4011 termination time.

4012 **RETURN VALUE**

4013 This function shall not return a value.

4014 **ERRORS**

4015 No errors are defined.

4016 **EXAMPLES**

4017 None.

4018 **APPLICATION USAGE**

4019 Calls to `thrd_exit()` should not be made from threads created using `pthread_create()` or via a
4020 `SIGEV_THREAD` notification, as their exit status has a different type (**void *** instead of
4021 **int**). If `thrd_exit()` is called from the initial thread and it is not the last thread to terminate,
4022 other threads should not try to obtain its exit status using `pthread_join()`.

4023 **RATIONALE**

4024 The normal mechanism by which a thread that was started using `thrd_create()` terminates is

4025 to return from the function that was specified in the *thrd_create()* call that started it. The
4026 *thrd_exit()* function provides the capability for such a thread to terminate without requiring a
4027 return from the start routine of that thread, thereby providing a function analogous to *exit()*.

4028 Regardless of the method of thread termination, the destructors for any existing thread-
4029 specific data are executed.

4030 **FUTURE DIRECTIONS**

4031 None.

4032 **SEE ALSO**

4033 *exit, pthread_create, thrd_join*

4034 XBD <**threads.h**>

4035 **CHANGE HISTORY**

4036 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4037 **NAME**

4038 *thrd_join* — wait for thread termination

4039 **SYNOPSIS**

4040 `#include <threads.h>`

4041 `int thrd_join(thrd_t thr, int *res);`

4042 **DESCRIPTION**

4043 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4044 Any conflict between the requirements described here and the ISO C standard is
4045 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4046 The *thrd_join()* function shall join the thread identified by *thr* with the current thread by
4047 blocking until the other thread has terminated. If the parameter *res* is not a null pointer,
4048 *thrd_join()* shall store the thread's exit status in the integer pointed to by *res*. The
4049 termination of the other thread shall synchronize with the completion of the *thrd_join()*
4050 function. The application shall ensure that the thread identified by *thr* has not been
4051 previously detached or joined with another thread.

4052 The results of multiple simultaneous calls to *thrd_join()* specifying the same target thread
4053 are undefined.

4054 The behavior is undefined if the value specified by the *thr* argument to *thrd_join()* refers to
4055 the calling thread.

4056 [CX]It is unspecified whether a thread that has exited but remains unjoined counts against
4057 {PTHREAD_THREADS_MAX}.

4058 If *thr* refers to a thread that was created using *pthread_create()* or via a SIGEV_THREAD
4059 notification and the thread terminates, or has already terminated, by returning from its start
4060 routine, the behavior of *thrd_join()* is undefined. If *thr* refers to a thread that terminates, or
4061 has already terminated, by calling *pthread_exit()* or by being cancelled, the behavior of
4062 *thrd_join()* is undefined.

4063 The *thrd_join()* function shall not be affected if the calling thread executes a signal handler
4064 during the call.[/CX]

4065 **RETURN VALUE**

4066 The *thrd_join()* function shall return *thrd_success* on success or *thrd_error* if the
4067 request could not be honored.

4068 [CX]It is implementation-defined whether *thrd_join()* detects deadlock situations; if it does
4069 detect them, it shall return *thrd_error* when one is detected.[/CX]

4070 **ERRORS**

4071 See RETURN VALUE.

4072 **EXAMPLES**

4073 None.

4074 **APPLICATION USAGE**

4075 None.

4076 **RATIONALE**

4077 The *thrd_join()* function provides a simple mechanism allowing an application to wait for a
4078 thread to terminate. After the thread terminates, the application may then choose to clean up
4079 resources that were used by the thread. For instance, after *thrd_join()* returns, any
4080 application-provided stack storage could be reclaimed.

4081 The *thrd_join()* or *thrd_detach()* function should eventually be called for every thread that is
4082 created using *thrd_create()* so that storage associated with the thread may be reclaimed.

4083 The *thrd_join()* function cannot be used to obtain the exit status of a thread that was created
4084 using *pthread_create()* or via a SIGEV_THREAD notification and which terminates by
4085 returning from its start routine, or of a thread that terminates by calling *pthread_exit()*,
4086 because such threads have a **void *** exit status, instead of the **int** that *thrd_join()* returns via
4087 its *res* argument.

4088 The *thrd_join()* function cannot be used to obtain the exit status of a thread that terminates
4089 by being cancelled because it has no way to indicate that a thread was cancelled. (The
4090 *pthread_join()* function does this by returning a reserved **void *** exit status; it is not possible
4091 to reserve an **int** value for this purpose without introducing a conflict with the ISO C
4092 standard.) The standard developers considered adding a *thrd_cancelled* enumeration
4093 constant that *thrd_join()* would return in this case. However, this return would be
4094 unexpected in code that is written to conform to the ISO C standard, and it would also not
4095 solve the problem that threads which use only ISO C <**threads.h**> interfaces (such as ones
4096 created by third party libraries written to conform to the ISO C standard) have no way to
4097 handle being cancelled, as the ISO C standard does not provide cancellation cleanup
4098 handlers.

4099 The *thrd_join()* function is not affected by signal handlers for the reasons stated in [xref to
4100 XRAT B.2.3].

4101 **FUTURE DIRECTIONS**

4102 None.

4103 **SEE ALSO**

4104 *pthread_create*, *pthread_exit*, *pthread_join*, *thrd_create*, *thrd_exit*

4105 XBD Section 4.12.2, <**threads.h**>

4106 **CHANGE HISTORY**

4107 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4108 **NAME**

4109 `thrd_sleep` — suspend execution for an interval

4110 **SYNOPSIS**

4111 `#include <threads.h>`

4112 `int thrd_sleep(const struct timespec *duration,`
4113 `struct timespec *remaining);`

4114 **DESCRIPTION**

4115 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4116 Any conflict between the requirements described here and the ISO C standard is
4117 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4118 The `thrd_sleep()` function shall suspend execution of the calling thread until either the
4119 interval specified by `duration` has elapsed or a signal is delivered to the calling thread whose
4120 action is to invoke a signal-catching function or to terminate the process. If interrupted by a
4121 signal and the `remaining` argument is not null, the amount of time remaining (the requested
4122 interval minus the time actually slept) shall be stored in the interval it points to. The
4123 `duration` and `remaining` arguments can point to the same object.

4124 The suspension time may be longer than requested because the interval is rounded up to an
4125 integer multiple of the sleep resolution or because of the scheduling of other activity by the
4126 system. But, except for the case of being interrupted by a signal, the suspension time shall
4127 not be less than that specified, as measured by the system clock `TIME_UTC`.

4128 **RETURN VALUE**

4129 The `thrd_sleep()` function shall return zero if the requested time has elapsed, `-1` if it has
4130 been interrupted by a signal, or a negative value (which may also be `-1`) if it fails for any
4131 other reason. [CX]If it returns a negative value, it shall set `errno` to indicate the error.[/CX]

4132 **ERRORS**

4133 [CX]The `thrd_sleep()` function shall fail if:

4134 [EINTR]

4135 The `thrd_sleep()` function was interrupted by a signal.

4136 [EINVAL]

4137 The `duration` argument specified a nanosecond value less than zero or greater than or
4138 equal to 1000 million.[/CX]

4139 **EXAMPLES**

4140 None.

4141 **APPLICATION USAGE**

4142 Since the return value may be `-1` for errors other than [EINTR], applications should examine

4143 *errno* to distinguish [EINTR] from other errors (and thus determine whether the unslept time
4144 is available in the interval pointed to by *remaining*).

4145 **RATIONALE**

4146 The *thrd_sleep()* function is identical to the *nanosleep()* function except that the return value
4147 may be any negative value when it fails with an error other than [EINTR].

4148 **FUTURE DIRECTIONS**

4149 None.

4150 **SEE ALSO**

4151 *nanosleep*

4152 XBD <**threads.h**>, <**time.h**>

4153 **CHANGE HISTORY**

4154 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4155 **NAME**

4156 *thrd_yield* — yield the processor

4157 **SYNOPSIS**

4158 `#include <threads.h>`

4159 `void thrd_yield(void);`

4160 **DESCRIPTION**

4161 [CX] The functionality described on this reference page is aligned with the ISO C standard.

4162 Any conflict between the requirements described here and the ISO C standard is

4163 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4164 [CX]The *thrd_yield()* function shall force the running thread to relinquish the processor until
4165 it again becomes the head of its thread list.[/CX]

4166 **RETURN VALUE**

4167 This function shall not return a value.

4168 **ERRORS**

4169 No errors are defined.

4170 **EXAMPLES**

4171 None.

4172 **APPLICATION USAGE**

4173 See the APPLICATION USAGE section for *sched_yield()*.

4174 **RATIONALE**

4175 The *thrd_yield()* function is identical to the *sched_yield()* function except that it does not
4176 return a value.

4177 **FUTURE DIRECTIONS**

4178 None.

4179 **SEE ALSO**
4180 *sched_yield*

4181 XBD <**threads.h**>

4182 **CHANGE HISTORY**
4183 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4184 Ref 7.27.2.5
4185 On page 2161 line 69278 insert a new `timespec_get()` section:

4186 **NAME**
4187 `timespec_get` — get time

4188 **SYNOPSIS**
4189 `#include <time.h>`

4190 `int timespec_get(struct timespec *ts, int base);`

4191 **DESCRIPTION**
4192 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4193 Any conflict between the requirements described here and the ISO C standard is
4194 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4195 The `timespec_get()` function shall set the interval pointed to by `ts` to hold the current
4196 calendar time based on the specified time base.

4197 [CX]If `base` is `TIME_UTC`, the members of `ts` shall be set to the same values as would be
4198 set by a call to `clock_gettime(CLOCK_REALTIME, ts)`. If the number of seconds will not
4199 fit in an object of type **time_t**, the function shall return zero.[/CX]

4200 **RETURN VALUE**
4201 If the `timespec_get()` function is successful it shall return the non-zero value `base`; otherwise,
4202 it shall return zero.

4203 **ERRORS**
4204 See DESCRIPTION.

4205 **EXAMPLES**
4206 None.

4207 **APPLICATION USAGE**
4208 None.

4209 **RATIONALE**
4210 None.

4211 **FUTURE DIRECTIONS**
4212 None.

4213 **SEE ALSO**

4214 *clock_getres, time*

4215 XBD <**time.h**>

4216 **CHANGE HISTORY**

4217 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4218 Ref 7.21.4.4 para 4, 7.1.4 para 5

4219 On page 2164 line 69377 section *tmpnam()*, change:

4220 [CX]The *tmpnam()* function need not be thread-safe if called with a NULL parameter.[/CX]

4221 to:

4222 If called with a null pointer argument, the *tmpnam()* function need not be thread-safe;
4223 however, such calls shall avoid data races with calls to *tmpnam()* with a non-null argument
4224 and with calls to all other functions.

4225 Ref 7.30.3.2.1 para 4

4226 On page 2171 line 69568 section *towctrans()*, change:

4227 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped
4228 value of *wc* using the mapping described by *desc*. Otherwise, they shall return *wc*
4229 unchanged.

4230 to:

4231 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped
4232 value of *wc* using the mapping described by *desc*, or the value of *wc* unchanged if *desc* is
4233 zero. [CX]Otherwise, they shall return *wc* unchanged.[/CX]

4234 Ref F.10.6.8 para 2

4235 On page 2177 line 69716 section *trunc()*, add a new paragraph:

4236 [MX]These functions may raise the inexact floating-point exception for finite non-integer
4237 arguments.[/MX]

4238 Ref F.10.6.8 para 1,2

4239 On page 2177 line 69719 section *trunc()*, change:

4240 [MX]The result shall have the same sign as *x*.[/MX]

4241 to:

4242 [MX]The returned value shall be exact, shall be independent of the current rounding
4243 direction mode, and shall have the same sign as *x*.[/MX]

4244 Ref F.10.6.8 para 2

4245 On page 2177 line 69730 section *trunc()*, delete from APPLICATION USAGE:

4246 These functions may raise the inexact floating-point exception if the result differs in value

4247 from the argument.

4248 Ref 7.26.6

4249 On page 2182 line 69835 insert the following new `tss_*`() sections:

4250 **NAME**

4251 `tss_create` — thread-specific data key creation

4252 **SYNOPSIS**

4253 `#include <threads.h>`

4254 `int tss_create(tss_t *key, tss_dtor_t dtor);`

4255 **DESCRIPTION**

4256 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4257 Any conflict between the requirements described here and the ISO C standard is
4258 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4259 The `tss_create()` function shall create a thread-specific storage pointer with destructor `dtor`,
4260 which can be null.

4261 A null pointer value shall be associated with the newly created key in all existing threads.
4262 Upon subsequent thread creation, the value associated with all keys shall be initialized to a
4263 null pointer value in the new thread.

4264 Destructors associated with thread-specific storage shall not be invoked at process
4265 termination.

4266 The behavior is undefined if the `tss_create()` function is called from within a destructor.

4267 [CX]The `tss_create()` function shall not be affected if the calling thread executes a signal
4268 handler during the call.[/CX]

4269 **RETURN VALUE**

4270 If the `tss_create()` function is successful, it shall set the thread-specific storage pointed to by
4271 `key` to a value that uniquely identifies the newly created pointer and shall return
4272 `thrd_success`; otherwise, `thrd_error` shall be returned and the thread-specific storage
4273 pointed to by `key` has an indeterminate value.

4274 **ERRORS**

4275 No errors are defined.

4276 **EXAMPLES**

4277 None.

4278 **APPLICATION USAGE**

4279 The `tss_create()` function performs no implicit synchronization. It is the responsibility of the
4280 programmer to ensure that it is called exactly once per key before use of the key.

4281 **RATIONALE**

4282 If the value associated with a key needs to be updated during the lifetime of the thread, it
4283 may be necessary to release the storage associated with the old value before the new value is
4284 bound. Although the `tss_set()` function could do this automatically, this feature is not needed

4285 often enough to justify the added complexity. Instead, the programmer is responsible for
4286 freeing the stale storage:

```
4287 old = tss_get(key);  
4288 new = allocate();  
4289 destructor(old);  
4290 tss_set(key, new);
```

4291 There is no notion of a destructor-safe function. If an application does not call *thrd_exit()* or
4292 *pthread_exit()* from a signal handler, or if it blocks any signal whose handler may call
4293 *thrd_exit()* or *pthread_exit()* while calling async-unsafe functions, all functions can be safely
4294 called from destructors.

4295 The *tss_create()* function is not affected by signal handlers for the reasons stated in [xref to
4296 XRAT B.2.3].

4297 **FUTURE DIRECTIONS**

4298 None.

4299 **SEE ALSO**

4300 *pthread_exit*, *pthread_key_create*, *thrd_exit*, *tss_delete*, *tss_get*

4301 XBD <**threads.h**>

4302 **CHANGE HISTORY**

4303 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4304 **NAME**

4305 *tss_delete* — thread-specific data key deletion

4306 **SYNOPSIS**

```
4307 #include <threads.h>
```

```
4308 void tss_delete(tss_t key);
```

4309 **DESCRIPTION**

4310 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4311 Any conflict between the requirements described here and the ISO C standard is
4312 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4313 The *tss_delete()* function shall release any resources used by the thread-specific storage
4314 identified by *key*. The thread-specific data values associated with *key* need not be null at the
4315 time *tss_delete()* is called. It is the responsibility of the application to free any application
4316 storage or perform any cleanup actions for data structures related to the deleted key or
4317 associated thread-specific data in any threads; this cleanup can be done either before or after
4318 *tss_delete()* is called.

4319 The application shall ensure that the *tss_delete()* function is only called with a value for *key*
4320 that was returned by a call to *tss_create()* before the thread commenced executing
4321 destructors.

4322 If *tss_delete()* is called while another thread is executing destructors, whether this will affect

4323 the number of invocations of the destructor associated with *key* on that thread is unspecified.

4324 The *tss_delete()* function shall be callable from within destructor functions. Calling
4325 *tss_delete()* shall not result in the invocation of any destructors. Any destructor function that
4326 was associated with *key* shall no longer be called upon thread exit.

4327 Any attempt to use *key* following the call to *tss_delete()* results in undefined behavior.

4328 [CX]The *tss_delete()* function shall not be affected if the calling thread executes a signal
4329 handler during the call.[/CX]

4330 RETURN VALUE

4331 This function shall not return a value.

4332 ERRORS

4333 No errors are defined.

4334 EXAMPLES

4335 None.

4336 APPLICATION USAGE

4337 None.

4338 RATIONALE

4339 A thread-specific data key deletion function has been included in order to allow the
4340 resources associated with an unused thread-specific data key to be freed. Unused thread-
4341 specific data keys can arise, among other scenarios, when a dynamically loaded module that
4342 allocated a key is unloaded.

4343 Conforming applications are responsible for performing any cleanup actions needed for data
4344 structures associated with the key to be deleted, including data referenced by thread-specific
4345 data values. No such cleanup is done by *tss_delete()*. In particular, destructor functions
4346 are not called. See the RATIONALE for *pthread_key_delete()* for the reasons for this
4347 division of responsibility.

4348 The *tss_delete()* function is not affected by signal handlers for the reasons stated in [xref to
4349 XRAT B.2.3].

4350 FUTURE DIRECTIONS

4351 None.

4352 SEE ALSO

4353 *pthread_key_create*, *tss_create*

4354 XBD <**threads.h**>

4355 CHANGE HISTORY

4356 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4357 NAME

4358 *tss_get*, *tss_set* — thread-specific data management

4359 **SYNOPSIS**

4360 `#include <threads.h>`

4361 `void *tss_get(tss_t key);`
4362 `int tss_set(tss_t key, void *val);`

4363 **DESCRIPTION**

4364 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4365 Any conflict between the requirements described here and the ISO C standard is
4366 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4367 The *tss_get()* function shall return the value for the current thread held in the thread-specific
4368 storage identified by *key*.

4369 The *tss_set()* function shall set the value for the current thread held in the thread-specific
4370 storage identified by *key* to *val*. This action shall not invoke the destructor associated with
4371 the key on the value being replaced.

4372 The application shall ensure that the *tss_get()* and *tss_set()* functions are only called with a
4373 value for *key* that was returned by a call to *tss_create()* before the thread commenced
4374 executing destructors.

4375 The effect of calling *tss_get()* or *tss_set()* after *key* has been deleted with *tss_delete()* is
4376 undefined.

4377 [CX]Both *tss_get()* and *tss_set()* can be called from a thread-specific data destructor
4378 function. A call to *tss_get()* for the thread-specific data key being destroyed shall return a
4379 null pointer, unless the value is changed (after the destructor starts) by a call to *tss_set()*.
4380 Calling *tss_set()* from a thread-specific data destructor function may result either in lost
4381 storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction)
4382 or in an infinite loop.

4383 These functions shall not be affected if the calling thread executes a signal handler during
4384 the call.[/CX]

4385 **RETURN VALUE**

4386 The *tss_get()* function shall return the value for the current thread. If no thread-specific data
4387 value is associated with *key*, then a null pointer shall be returned.

4388 The *tss_set()* function shall return *thrd_success* on success or *thrd_error* if the request
4389 could not be honored.

4390 **ERRORS**

4391 No errors are defined.

4392 **EXAMPLES**

4393 None.

4394 **APPLICATION USAGE**

4395 None.

4396 **RATIONALE**

4397 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
4398 B.2.3].

4399 **FUTURE DIRECTIONS**

4400 None.

4401 **SEE ALSO**

4402 *pthread_getspecific, tss_create*

4403 XBD <threads.h>

4404 **CHANGE HISTORY**

4405 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4406 Ref 7.31.11 para 2

4407 On page 2193 line 70145 section `ungetc()`, change FUTURE DIRECTIONS from:

4408 None.

4409 to:

4410 The ISO C standard states that the use of `ungetc()` on a binary stream where the file position
4411 indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction
4412 between binary and text streams, so this applies to all streams. This feature may be removed
4413 in a future version of this standard.

4414 Ref 7.29.6.3 para 1, 7.1.4 para 5

4415 On page 2242 line 71441 section `wcrtomb()`, change:

4416 [CX]The `wcrtomb()` function need not be thread-safe if called with a NULL *ps*
4417 argument.[/CX]

4418 to:

4419 If called with a null *ps* argument, the `wcrtomb()` function need not be thread-safe; however,
4420 such calls shall avoid data races with calls to `wcrtomb()` with a non-null argument and with
4421 calls to all other functions.

4422 Ref 7.29.6.4 para 1, 7.1.4 para 5

4423 On page 2266 line 72111 section `wcsrtombs()`, change:

4424 [CX]The `wcsnrtombs()` and `wcsrtombs()` functions need not be thread-safe if called with a
4425 NULL *ps* argument.[/CX]

4426 to:

4427 [CX]If called with a null *ps* argument, the `wcsnrtombs()` function need not be thread-safe;
4428 however, such calls shall avoid data races with calls to `wcsnrtombs()` with a non-null
4429 argument and with calls to all other functions.[/CX]

4430 If called with a null *ps* argument, the `wcsrtombs()` function need not be thread-safe;

4431 however, such calls shall avoid data races with calls to `wcsrtombs()` with a non-null
4432 argument and with calls to all other functions.

4433 Ref 7.22.7 para 1, 7.1.4 para 5
4434 On page 2292 line 72879 section `wctomb()`, change:

4435 [CX]The `wctomb()` function need not be thread-safe.[/CX]

4436 to:

4437 The `wctomb()` function need not be thread-safe; however, it shall avoid data races with all
4438 other functions.

4439 Changes to XCU

4440 Ref 7.22.2
4441 On page 2333 line 74167 section 1.1.2.2 Mathematical Functions, change:

4442 Section 7.20.2, Pseudo-Random Sequence Generation Functions

4443 to:

4444 Section 7.22.2, Pseudo-Random Sequence Generation Functions

4445 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)
4446 On page 2542 line 82220 section c99, rename the c99 page to c17.

4447 Ref 7.26
4448 On page 2545 line 82375 section c99 (now c17), change:

4449 ... , `<spawn.h>`, `<sys/socket.h>`, ...

4450 to:

4451 ... , `<spawn.h>`, `<sys/socket.h>`, `<threads.h>`, ...

4452 Ref 7.26
4453 On page 2545 line 82382 section c99 (now c17), change:

4454 This option shall make available all interfaces referenced in `<pthread.h>` and `pthread_kill()`
4455 and `pthread_sigmask()` referenced in `<signal.h>`.

4456 to:

4457 This option shall make available all interfaces referenced in `<pthread.h>` and `<threads.h>`,
4458 and also `pthread_kill()` and `pthread_sigmask()` referenced in `<signal.h>`.

4459 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)
4460 On page 2552-2553 line 82641-82677 section c99 (now c17), change CHANGE HISTORY to:

4461 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4462 **Changes to XRAT**

4463 Ref G.1 para 1

4464 On page 3483 line 117680 section A.1.7.1 Codes, add a new tagged paragraph:

4465 MXC This margin code is used to denote functionality related to the IEC 60559 Complex
4466 Floating-Point option.

4467 Ref (none)

4468 On page 3489 line 117909 section A.3 Definitions (Byte), change:

4469 alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types are now defined.

4470 to:

4471 alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types were first defined.

4472 Ref 5.1.2.4, 7.17.3

4473 On page 3515 line 118946 section A.4.12 Memory Synchronization, change:

4474 **A.4.12 Memory Synchronization**

4475 to:

4476 **A.4.12 Memory Ordering and Synchronization**

4477 A.4.12.1 *Memory Ordering*

4478 There is no additional rationale provided for this section.

4479 A.4.12.2 *Memory Synchronization*

4480 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4481 On page 3556 line 120684 section A.12.2 Utility Syntax Guidelines, change:

4482 Thus, they had to devise a new name, *c89* (now superseded by *c99*), rather than ...

4483 to:

4484 Thus, they had to devise a new name, *c89* (subsequently superseded by *c99* and now by
4485 *c17*), rather than ...

4486 Ref K.3.1.1

4487 On page 3567 line 121053 section B.2.2.1 POSIX.1 Symbols, add a new unnumbered subsection:

4488 **The `__STDC_WANT_LIB_EXT1__` Feature Test Macro**

4489 The ISO C standard specifies the feature test macro `__STDC_WANT_LIB_EXT1__` as the
4490 announcement mechanism for the application that it requires functionality from Annex K. It

4491 specifies that the symbols specified in Annex K (if supported) are made visible when
4492 `__STDC_WANT_LIB_EXT1__` is 1 and are not made visible when it is 0, but leaves it
4493 unspecified whether they are made visible when `__STDC_WANT_LIB_EXT1__` is
4494 undefined. POSIX.1 requires that they are not made visible when the macro is undefined
4495 (except for those symbols that are already explicitly allowed to be visible through the
4496 definition of `_POSIX_C_SOURCE` or `_XOPEN_SOURCE`, or both).

4497 POSIX.1 does not include the interfaces specified in Annex K of the ISO C standard, but
4498 allows the symbols to be made visible in headers when requested by the application in order
4499 that applications can use symbols from Annex K and symbols from POSIX.1 in the same
4500 translation unit.

4501 Ref 6.10.3.4

4502 On page 3570 line 121176 section B.2.2.2 The Name Space, change:

4503 as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C
4504 standard

4505 to:

4506 as described for macros that expand to their own name as in Section 6.10.3.4 of the ISO C
4507 standard

4508 Ref 7.5 para 2

4509 On page 3571 line 121228-121243 section B.2.3 Error Numbers, change:

4510 The ISO C standard requires that *errno* be an assignable lvalue. Originally, ...
4511 [...]
4512 ... using the return value for a mixed purpose was judged to be of limited use and
4513 error prone.

4514 to:

4515 The original ISO C standard just required that *errno* be an modifiable lvalue. Since the
4516 introduction of threads in 2011, the ISO C standard has instead required that *errno* be a
4517 macro which expands to a modifiable lvalue that has thread local storage duration.

4518 Ref 7.26

4519 On page 3575 line 121390 section B.2.3 Error Numbers, change:

4520 In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a
4521 special case since it will never occur.

4522 to:

4523 In particular, applications calling blocking interfaces need not handle any possible [EINTR]
4524 return as a special case since it will never occur. In the case of threads functions in
4525 `<threads.h>`, the requirement is stated in terms of the call not being affected if the calling
4526 thread executes a signal handler during the call, since these functions return errors in a
4527 different way and cannot distinguish an [EINTR] condition from other error conditions.

4528 Ref (none)

4529 On page 3733 line 128128 section C.2.6.4 Arithmetic Expansion, change:

4530 Although the ISO/IEC 9899: 1999 standard now requires support for ...

4531 to:

4532 Although the ISO C standard requires support for ...

4533 Ref 7.17

4534 On page 3789 line 129986 section E.1 Subprofiling Option Groups, change:

4535 by collecting sets of related functions

4536 to:

4537 by collecting sets of related functions and generic functions

4538 Ref 7.22.3.1, 7.27.2.5, 7.22.4

4539 On page 3789, 3792 line 130022-130032, 130112-130114 section E.1 Subprofiling Option Groups,

4540 add new functions (in sorted order) to the existing groups as indicated:

4541 POSIX_C_LANG_SUPPORT

4542 *aligned_alloc(), timespec_get()*

4543 POSIX_MULTI_PROCESS

4544 *at_quick_exit(), quick_exit()*

4545 Ref 7.17

4546 On page 3789 line 129991 section E.1 Subprofiling Option Groups, add:

4547 POSIX_C_LANG_ATOMICS: ISO C Atomic Operations

4548 *atomic_compare_exchange_strong(), atomic_compare_exchange_strong_explicit(),*

4549 *atomic_compare_exchange_weak(), atomic_compare_exchange_weak_explicit(),*

4550 *atomic_exchange(), atomic_exchange_explicit(), atomic_fetch_add(),*

4551 *atomic_fetch_add_explicit(), atomic_fetch_and(), atomic_fetch_and_explicit(),*

4552 *atomic_fetch_or(), atomic_fetch_or_explicit(), atomic_fetch_sub(),*

4553 *atomic_fetch_sub_explicit(), atomic_fetch_xor(), atomic_fetch_xor_explicit(),*

4554 *atomic_flag_clear(), atomic_flag_clear_explicit(), atomic_flag_test_and_set(),*

4555 *atomic_flag_test_and_set_explicit(), atomic_init(), atomic_is_lock_free(),*

4556 *atomic_load(), atomic_load_explicit(), atomic_signal_fence(),*

4557 *atomic_thread_fence(), atomic_store(), atomic_store_explicit(), kill_dependency()*

4558 Ref 7.26

4559 On page 3790 line 1300349 section E.1 Subprofiling Option Groups, add:

4560 POSIX_C_LANG_THREADS: ISO C Threads

4561 *call_once(), cnd_broadcast(), cnd_signal(), cnd_destroy(), cnd_init(),*

4562 *cnd_timedwait(), cnd_wait(), mtx_destroy(), mtx_init(), mtx_lock(), mtx_timedlock(),*

4563 *mtx_trylock(), mtx_unlock(), thrd_create(), thrd_current(), thrd_detach(),*

4564 *thrd_equal(), thrd_exit(), thrd_join(), thrd_sleep(), thrd_yield(), tss_create(),*

4565 *tss_delete(), tss_get(), tss_set()*

4566 POSIX_C_LANG_UCHAR: ISO C Unicode Utilities

