

1 **TODO**

2 Check for overlaps with Mantis bugs: 374 and 1218 (once resolved; NB 374 may also affect
3 aligned_alloc()), and any that get tagged tc3 or issue8 after 2020-10-29

4 **Introduction**

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

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

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

12 **Global Change**

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

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

17 **Changes to XBD**

18 Ref G.1 para 1

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

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

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

23 Ref (none)

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

25 the ISO/IEC 9899: 1999 standard

26 to:

27 the ISO C standard

28 Ref 6.2.8

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

30 See also the ISO C standard, Section B3.

31 to:

32 See also the ISO C standard, Section 6.2.8.

33 Ref 5.1.2.4

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

35 **3.31 Atomic Operation**

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

40 Ref 7.26.3

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

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

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

49 Ref 5.1.2.4

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

51 **3.125 Data Race**

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

55 Ref 5.1.2.4

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

57 **3.215 Lock-Free Operation**

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

60 Ref 7.26.5.1

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

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

64 to:

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

67 Ref 7.26.4

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

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

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

75 Ref 7.26.5.5

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

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

79 to:

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

82 Ref 7.26.5.1

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

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

86 to:

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

89 Ref 5.1.2.4

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

91 **3.382 Synchronization Operation**

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

93 Ref 7.26.5.1

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

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

97 to:

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

- 100 • The initial thread.

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

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

110 Ref 5.1.2.4

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

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

114 to:

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

117 Ref 5.1.2.4

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

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

122 Ref 7.26.6

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

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

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

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

131 to:

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

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

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

142 Ref 5.1.2.4, 7.17.3

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

144 **4.12 Memory Synchronization**

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

151 to:

152 **4.12 Memory Ordering and Synchronization**

153 **4.12.1 Memory Ordering**

154 *4.12.1.1 Data Races*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

219 **Note:** The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced
220 before”, “synchronizes with”, and “dependency-ordered before” relationships, with two

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

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

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

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

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

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

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

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

249
250 The value of an atomic object *M*, as determined by evaluation *B*, shall be the value stored by
251 some side effect *A* that modifies *M*, where *B* does not happen before *A*.

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

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

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

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

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

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

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

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

278 4.12.1.2 Memory Order and Consistency

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

283 For `memory_order_relaxed`, no operation orders memory.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

336 4.12.2 Memory Synchronization

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

343 Ref 7.26.3, 7.26.4

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

| | | | |
|-----|------------------------|------------------------|----------------------|
| 346 | <i>cnd_broadcast()</i> | <i>mtx_lock()</i> | <i>thrd_create()</i> |
| 347 | <i>cnd_signal()</i> | <i>mtx_timedlock()</i> | <i>thrd_join()</i> |
| 348 | <i>cnd_timedwait()</i> | <i>mtx_trylock()</i> | |
| 349 | <i>cnd_wait()</i> | <i>mtx_unlock()</i> | |

350 Ref 7.26.2.1, 7.26.4

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

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

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

360 to:

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

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

371 Ref 7.12.1 para 7

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

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

375 to:

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

381 Ref 7.12.1 para 3

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

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

385 to:

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

391 Ref 7.12.1 para 4

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

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

395 add:

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

399 Ref 7.29.1 para 5

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

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

408 Ref 7.3.1 para 2

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

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

415 Ref G.6 para 1

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

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

419 add:

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

423 Ref 7.3.9.3

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

425 The following shall be defined as macros.

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

429 Ref 7.3.1 para 2

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

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

436 Ref 7.3.9.3

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

438 Ref 7.5 para 2

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

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

443 to:

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

446 Ref (none)

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

448 the ISO/IEC 9899: 1999 standard

449 to:

450 the ISO C standard

451 Ref 5.2.4.2.2 para 11

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

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

-1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

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

460 Ref 5.2.4.2.2 para 12

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

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

465 [math stuff]

466 FLT_DECIMAL_DIG 6

467 DBL_DECIMAL_DIG 10

468 LDBL_DECIMAL_DIG 10

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

471 Ref 5.2.4.2.2 para 14

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

473 Minimum positive floating-point number.

474 FLT_TRUE_MIN 1E-37

475 DBL_TRUE_MIN 1E-37

476 LDBL_TRUE_MIN 1E-37

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

480 Ref (none)

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

482 the ISO/IEC 9899: 1999 standard

483 to:

484 the ISO C standard

485 Ref 7.22.4.3

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

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

488 to:

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

491 Ref 5.2.4.2.1 para 2

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

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

498 to:

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

503 Ref (none)

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

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

506 to:

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

508 Ref 7.26.5.5

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

510 void pthread_exit(void *);

511 to:

512 _Noreturn void pthread_exit(void *);

513 Ref 7.13.2.1 para 1

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

515 void longjmp(jmp_buf, int);

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

517 to:

518 _Noreturn void longjmp(jmp_buf, int);

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

520 Ref 7.15

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

522 **NAME**

523 stdalign.h — alignment macros

524 **SYNOPSIS**

525 #include <stdalign.h>

526 **DESCRIPTION**

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

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

531 alignas Expands to **_Alignas**

532 alignof Expands to **_Alignof**

533 __alignas_is_defined

534 Expands to the integer constant 1

535 __alignof_is_defined

536 Expands to the integer constant 1

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

539 **APPLICATION USAGE**

540 None.

541 **RATIONALE**

542 None.

543 **FUTURE DIRECTIONS**

544 None.

545 **SEE ALSO**

546 None.

547 **CHANGE HISTORY**

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

549 Ref 7.17, 7.31.8 para 2

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

551 **NAME**

552 stdatomic.h — atomics

553 **SYNOPSIS**

554 #include <stdatomic.h>

555 **DESCRIPTION**

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

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

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

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

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

| Atomic type name | Direct type |
|------------------------------|-----------------------------------|
| atomic_bool | _Atomic_Bool |
| atomic_char | _Atomic char |
| atomic_schar | _Atomic signed char |
| atomic_uchar | _Atomic unsigned char |
| atomic_short | _Atomic short |
| atomic_ushort | _Atomic unsigned short |
| atomic_int | _Atomic int |
| atomic_uint | _Atomic unsigned int |
| atomic_long | _Atomic long |
| atomic_ulong | _Atomic unsigned long |
| atomic_llong | _Atomic long long |
| atomic_ullong | _Atomic unsigned long long |
| atomic_char16_t | _Atomic char16_t |
| atomic_char32_t | _Atomic char32_t |
| atomic_wchar_t | _Atomic wchar_t |
| atomic_int_least8_t | _Atomic int_least8_t |
| atomic_uint_least8_t | _Atomic uint_least8_t |
| atomic_int_least16_t | _Atomic int_least16_t |
| atomic_uint_least16_t | _Atomic uint_least16_t |
| atomic_int_least32_t | _Atomic int_least32_t |
| atomic_uint_least32_t | _Atomic uint_least32_t |
| atomic_int_least64_t | _Atomic int_least64_t |
| atomic_uint_least64_t | _Atomic uint_least64_t |
| atomic_int_fast8_t | _Atomic int_fast8_t |
| atomic_uint_fast8_t | _Atomic uint_fast8_t |
| atomic_int_fast16_t | _Atomic int_fast16_t |
| atomic_uint_fast16_t | _Atomic uint_fast16_t |
| atomic_int_fast32_t | _Atomic int_fast32_t |
| atomic_uint_fast32_t | _Atomic uint_fast32_t |
| atomic_int_fast64_t | _Atomic int_fast64_t |
| atomic_uint_fast64_t | _Atomic uint_fast64_t |
| atomic_intptr_t | _Atomic intptr_t |

| | |
|-------------------------------|--------------------------------|
| <code>atomic_uintptr_t</code> | <code>_Atomic uintptr_t</code> |
| <code>atomic_size_t</code> | <code>_Atomic size_t</code> |
| <code>atomic_ptrdiff_t</code> | <code>_Atomic ptrdiff_t</code> |
| <code>atomic_intmax_t</code> | <code>_Atomic intmax_t</code> |
| <code>atomic_uintmax_t</code> | <code>_Atomic uintmax_t</code> |

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

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

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

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

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

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

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

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

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

```

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

```

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

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

```

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

```

649 APPLICATION USAGE

650 None.

651 RATIONALE

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

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

659 **FUTURE DIRECTIONS**

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

662 **SEE ALSO**

663 Section 4.12.1

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

668 **CHANGE HISTORY**

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

670 Ref 7.31.9

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

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

673 Ref 7.19 para 2

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

675 **`max_align_t`** Object type whose alignment is the greatest fundamental alignment.

676 Ref (none)

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

678 the ISO/IEC 9899: 1999 standard

679 to:

680 the ISO C standard

681 Ref 7.20.1.1 para 1

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

683 denotes a signed integer type

684 to:

685 denotes such a signed integer type

686 Ref 7.20.1.1 para 2

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

688 ... designates an unsigned integer type with width *N*. Thus, **`uint24_t`** denotes an unsigned

689 integer type ...

690 to:

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

693 Ref 7.21.1 para 2

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

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

697 to:

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

700 Ref 7.21.1 para 3

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

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

706 to:

707 None.

708 Ref 7.22.4.5 para 1

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

710 void _Exit(int);

711 to:

712 _Noreturn void _Exit(int);

713 Ref 7.22.4.1 para 1

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

715 void abort(void);

716 to:

717 _Noreturn void abort(void);

718 Ref 7.22.3.1, 7.22.4.3

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

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

722 Ref 7.22.4.4 para 1
723 On page 360 line 12282 section <stdlib.h>, change:

724 void exit(int);

725 to:

726 _Noreturn void exit(int);

727 Ref 7.22.4.7
728 On page 360 line 12309 section <stdlib.h>, add:

729 _Noreturn void quick_exit(int);

730 Ref 7.23
731 On page 363 line 12380 insert a new <stdnoreturn.h> section:

732 **NAME**

733 stdnoreturn.h — noreturn macro

734 **SYNOPSIS**

735 #include <stdnoreturn.h>

736 **DESCRIPTION**

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

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

742 **APPLICATION USAGE**

743 None.

744 **RATIONALE**

745 None.

746 **FUTURE DIRECTIONS**

747 None.

748 **SEE ALSO**

749 None.

750 **CHANGE HISTORY**

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

752 Ref G.7

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

754 [MXC]Type-generic macros that accept complex arguments shall also accept imaginary
755 arguments. If an argument is imaginary, the macro shall expand to an expression whose type
756 is real, imaginary, or complex, as appropriate for the particular function: if the argument is

757 imaginary, then the types of *cos()*, *cosh()*, *fabs()*, *carg()*, *cimag()*, and *creal()* shall be real;
758 the types of *sin()*, *tan()*, *sinh()*, *tanh()*, *asin()*, *atan()*, *asinh()*, and *atanh()* shall be imaginary;
759 and the types of the others shall be complex.

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

| | | | |
|-----|------------------|---|-------------------|
| 763 | <i>cos(iy)</i> | = | <i>cosh(y)</i> |
| 764 | <i>sin(iy)</i> | = | <i>i sinh(y)</i> |
| 765 | <i>tan(iy)</i> | = | <i>i tanh(y)</i> |
| 766 | <i>cosh(iy)</i> | = | <i>cos(y)</i> |
| 767 | <i>sinh(iy)</i> | = | <i>i sin(y)</i> |
| 768 | <i>tanh(iy)</i> | = | <i>i tan(y)</i> |
| 769 | <i>asin(iy)</i> | = | <i>i asinh(y)</i> |
| 770 | <i>atan(iy)</i> | = | <i>i atanh(y)</i> |
| 771 | <i>asinh(iy)</i> | = | <i>i asin(y)</i> |
| 772 | <i>atanh(iy)</i> | = | <i>i atan(y)</i> |
| 773 | [/MXC] | | |

774 Ref (none)
775 On page 423 line 14404 section <tgmath.h>, change:

776 the ISO/IEC 9899: 1999 standard

777 to:

778 the ISO C standard

779 Ref 7.26
780 On page 424 line 14425 insert a new <threads.h> section:

781 **NAME**
782 threads.h — ISO C threads

783 **SYNOPSIS**
784 #include <threads.h>

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

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

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

794 `thread_local` Expands to `_Thread_local`.

795 ONCE_FLAG_INIT Expands to a value that can be used to initialize an object of
796 type **once_flag**.

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

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

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

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

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

```

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

```

844 int tss_set(tss_t, void *);

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

847 **APPLICATION USAGE**

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

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

859 **RATIONALE**

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

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

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

882 **FUTURE DIRECTIONS**

883 None.

884 **SEE ALSO**

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

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

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

890 CHANGE HISTORY

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

892 Ref 7.27.1 para 4

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

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

| | | | |
|-----|---------------------|----------------------|--------------|
| 896 | <code>time_t</code> | <code>tv_sec</code> | Seconds. |
| 897 | <code>long</code> | <code>tv_nsec</code> | Nanoseconds. |

898 and change the members to:

| | | | |
|-----|---------------------|----------------------|-------------------------------|
| 899 | <code>time_t</code> | <code>tv_sec</code> | Whole seconds. |
| 900 | <code>long</code> | <code>tv_nsec</code> | Nanoseconds [0, 999 999 999]. |

901 Ref 7.27.1 para 2

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

| | | |
|-----|-----------------------|--|
| 903 | <code>TIME_UTC</code> | An integer constant greater than 0 that designates the UTC time base 904 in calls to <i>timespec_get()</i> . The value shall be suitable for use in #if 905 preprocessing directives. |
|-----|-----------------------|--|

906 Ref 7.27.2.5

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

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

909 Ref 7.28

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

911 NAME

912 `uchar.h` — Unicode character handling

913 SYNOPSIS

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

915 DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

936 **APPLICATION USAGE**

937 None.

938 **RATIONALE**

939 None.

940 **FUTURE DIRECTIONS**

941 None.

942 **SEE ALSO**

943 <**stddef.h**>, <**stdint.h**>, <**wchar.h**>

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

945 **CHANGE HISTORY**

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

947 Ref 7.22.4.5 para 1

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

949 void _exit(int);

950 to:

951 _Noreturn void _exit(int);

952 Ref 7.29.1 para 2

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

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

955 to:

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

957 **Changes to XSH**

958 Ref 7.1.4 paras 5, 6

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

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

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

970 Ref K.3.1.1

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

972 2.2.1.3 *The `__STDC_WANT_LIB_EXT1__` Feature Test Macro*

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

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

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

985 Ref 7.31.8 para 1

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

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

987 Ref 7.31.15 para 1

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

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

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

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

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

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

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

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

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

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

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

1002 **Note:** The above two tables only include those symbols from Annex K of the ISO C standard that
 1003 are not already allowed to be visible by entries in earlier tables in this section.

1004 Ref 7.1.3 para 1
 1005 On page 478 line 16438 section 2.2.2, change:

1006 With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that
 1007 begin with an <underscore> and either an uppercase letter or another <underscore> are
 1008 always reserved for any use by the implementation.

1009 to:

1010 With the exception of identifiers beginning with the prefix `_POSIX_` and those identifiers
1011 which are lexically identical to keywords defined by the ISO C standard (for example
1012 `_Bool`), all identifiers that begin with an `<underscore>` and either an uppercase letter or
1013 another `<underscore>` are always reserved for any use by the implementation.

1014 Ref 7.1.3 para 1

1015 On page 478 line 16448 section 2.2.2, change:

1016 that have external linkage are always reserved

1017 to:

1018 that have external linkage and `errno` are always reserved

1019 Ref 7.1.3 para 1

1020 On page 479 line 16453 section 2.2.2, add the following in the appropriate place in the list:

| | | |
|------|--|------------------------------|
| 1021 | <code>aligned_alloc</code> | <code>c32rtomb</code> |
| 1022 | <code>at_quick_exit</code> | <code>call_once</code> |
| 1023 | <code>atomic_compare_exchange_strong</code> | <code>cnd_broadcast</code> |
| 1024 | <code>atomic_compare_exchange_strong_explicit</code> | <code>cnd_destroy</code> |
| 1025 | <code>atomic_compare_exchange_weak</code> | <code>cnd_init</code> |
| 1026 | <code>atomic_compare_exchange_weak_explicit</code> | <code>cnd_signal</code> |
| 1027 | <code>atomic_exchange</code> | <code>cnd_timedwait</code> |
| 1028 | <code>atomic_exchange_explicit</code> | <code>cnd_wait</code> |
| 1029 | <code>atomic_fetch_add</code> | <code>kill_dependency</code> |
| 1030 | <code>atomic_fetch_add_explicit</code> | <code>mbrtoc16</code> |
| 1031 | <code>atomic_fetch_and</code> | <code>mbrtoc32</code> |
| 1032 | <code>atomic_fetch_and_explicit</code> | <code>mtx_destroy</code> |
| 1033 | <code>atomic_fetch_or</code> | <code>mtx_init</code> |
| 1034 | <code>atomic_fetch_or_explicit</code> | <code>mtx_lock</code> |
| 1035 | <code>atomic_fetch_sub</code> | <code>mtx_timedlock</code> |
| 1036 | <code>atomic_fetch_sub_explicit</code> | <code>mtx_trylock</code> |
| 1037 | <code>atomic_fetch_xor</code> | <code>mtx_unlock</code> |
| 1038 | <code>atomic_fetch_xor_explicit</code> | <code>quick_exit</code> |
| 1039 | <code>atomic_flag_clear</code> | <code>thr_create</code> |
| 1040 | <code>atomic_flag_clear_explicit</code> | <code>thr_current</code> |
| 1041 | <code>atomic_flag_test_and_set</code> | <code>thr_detach</code> |
| 1042 | <code>atomic_flag_test_and_set_explicit</code> | <code>thr_equal</code> |
| 1043 | <code>atomic_init</code> | <code>thr_exit</code> |
| 1044 | <code>atomic_is_lock_free</code> | <code>thr_join</code> |
| 1045 | <code>atomic_load</code> | <code>thr_sleep</code> |
| 1046 | <code>atomic_load_explicit</code> | <code>thr_yield</code> |
| 1047 | <code>atomic_signal_fence</code> | <code>timespec_get</code> |
| 1048 | <code>atomic_store</code> | <code>tss_create</code> |
| 1049 | <code>atomic_store_explicit</code> | <code>tss_delete</code> |
| 1050 | <code>atomic_thread_fence</code> | <code>tss_get</code> |
| 1051 | <code>c16rtomb</code> | <code>tss_set</code> |

1052 Ref 7.1.2 para 4

1053 On page 480 line 16551 section 2.2.2, change:

1054 Prior to the inclusion of a header, the application shall not define any macros with names
1055 lexically identical to symbols defined by that header.

1056 to:

1057 Prior to the inclusion of a header, or when any macro defined in the header is expanded, the
1058 application shall not define any macros with names lexically identical to symbols defined by
1059 that header.

1060 Ref 7.26.5.1

1061 On page 490 line 16980 section 2.4.2 Realtime Signal Generation and Delivery, change:

1062 The function shall be executed in an environment as if it were the *start_routine* for a newly
1063 created thread with thread attributes specified by *sigev_notify_attributes*.

1064 to:

1065 The function shall be executed in a newly created thread as if it were the *start_routine* for a
1066 call to *pthread_create()* with the thread attributes specified by *sigev_notify_attributes*.

1067 Ref 7.14.1.1 para 5

1068 On page 493 line 17088 section 2.4.3 Signal Actions, change:

1069 with static storage duration

1070 to:

1071 with static or thread storage duration that is not a lock-free atomic object

1072 Ref 7.14.1.1 para 5

1073 On page 493 line 17090 section 2.4.3 Signal Actions, after applying bug 711 change:

1074 other than one of the functions and macros listed in the following table

1075 to:

1076 other than one of the functions and macros specified below as being async-signal-safe

1077 Ref 7.14.1.1 para 5

1078 On page 494 line 17133 section 2.4.3 Signal Actions, add *quick_exit()* to the table of async-signal-
1079 safe functions.

1080 Ref 7.14.1.1 para 5

1081 On page 494 line 17147 section 2.4.3 Signal Actions, change:

1082 Any function or function-like macro not in the above table may be unsafe with respect to
1083 signals.

1084 to:

1085 In addition, the functions in `<stdatomic.h>` other than `atomic_init()` shall be async-signal-
1086 safe when the atomic arguments are lock-free, and the `atomic_is_lock_free()` function shall
1087 be async-signal-safe when called with an atomic argument.

1088 All other functions (including generic functions) and function-like macros may be unsafe
1089 with respect to signals.

1090 Ref 7.21.2 para 7,8

1091 On page 496 line 17228 section 2.5 Standard I/O Streams, add a new paragraph:

1092 Each stream shall have an associated lock that is used to prevent data races when multiple
1093 threads of execution access a stream, and to restrict the interleaving of stream operations
1094 performed by multiple threads. Only one thread can hold this lock at a time. The lock shall
1095 be reentrant: a single thread can hold the lock multiple times at a given time. All functions
1096 that read, write, position, or query the position of a stream, [CX]except those with names
1097 ending `_unlocked`[/CX], shall lock the stream [CX] as if by a call to `flockfile()`[/CX] before
1098 accessing it and release the lock [CX] as if by a call to `funlockfile()`[/CX] when the access is
1099 complete.

1100 Ref (none)

1101 On page 498 line 17312 section 2.5.2 Stream Orientation and Encoding Rules, change:

1102 For conformance to the ISO/IEC 9899: 1999 standard, the definition of a stream includes an
1103 “orientation”.

1104 to:

1105 The definition of a stream includes an “orientation”.

1106 Ref 7.26.5.8

1107 On page 508 line 17720 section 2.8.4 Process Scheduling, change:

1108 When a running thread issues the `sched_yield()` function

1109 to:

1110 When a running thread issues the `sched_yield()` or `thr_yield()` function

1111 Ref 7.17.2.2 para 3, 7.22.2.2 para 3

1112 On page 513 line 17907,17916 section 2.9.1 Thread-Safety, add `atomic_init()` and `srand()` to the list
1113 of functions that need not be thread-safe.

1114 Ref 7.12.8.3, 7.22.4.8

1115 On page 513 line 17907-17927 section 2.9.1 Thread-Safety, delete the following from the list of
1116 functions that need not be thread-safe:

1117 `lgamma()`, `lgammaf()`, `lgammal()`, `system()`

1118 [Note to reviewers: deletion of `mblen\(\)`, `mbtowc\(\)`, and `wctomb\(\)` from this list is the subject of](#)
1119 [Mantis bug 708.](#)

1120 Ref 7.28.1 para 1

1121 On page 513 line 17928 section 2.9.1 Thread-Safety, change:

1122 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument.
1123 The *mbrlen()*, *mbrtowc()*, *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and
1124 *wcsrtombs()* functions need not be thread-safe if passed a NULL *ps* argument.

1125 to:

1126 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a null pointer
1127 argument. The *c16rtomb()*, *c32rtomb()*, *mbrlen()*, *mbrtoc16()*, *mbrtoc32()*, *mbrtowc()*,
1128 *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and *wcsrtombs()* functions need not
1129 be thread-safe if passed a null *ps* argument. The *lgamma()*, *lgammaf()*, and *lgammal()*
1130 functions shall be thread-safe [XSI]except that they need not avoid data races when storing a
1131 value in the *siggam* variable[/XSI].

1132 Ref 7.1.4 para 5

1133 On page 513 line 17934 section 2.9.1 Thread-Safety, change:

1134 Implementations shall provide internal synchronization as necessary in order to satisfy this
1135 requirement.

1136 to:

1137 Some functions that are not required to be thread-safe are nevertheless required to avoid data
1138 races with either all or some other functions, as specified on their individual reference pages.

1139 Implementations shall provide internal synchronization as necessary in order to satisfy
1140 thread-safety requirements.

1141 Ref 7.26.5

1142 On page 513 line 17944 section 2.9.2 Thread IDs, change:

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

1146 to:

1147 The lifetime of a thread ID ends after the thread terminates if it was created using
1148 *pthread_create()* with the *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or
1149 if *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()* has been called for that
1150 thread.

1151 Ref 7.26.5

1152 On page 514 line 17950 section 2.9.2 Thread IDs, change:

1153 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1154 *pthread_detach()* or *pthread_join()*.

1155 to:

1156 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1157 *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()*.

1158 Ref 7.26.4

1159 On page 514 line 17956 section 2.9.3 Thread Mutexes, change:

1160 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

1161 to:

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

1164 Ref 7.26.3, 7.26.4

1165 On page 514 line 17972 section 2.9.3 Thread Mutexes, add two new paragraphs and lists:

1166 A thread shall become the owner of a mutex, *m*, of type **mtx_t** when one of the following
1167 occurs:

- 1168 • It calls *mtx_lock()* with *m* as the *mtx* argument and the call returns *thrd_success*.
- 1169 • It calls *mtx_trylock()* with *m* as the *mtx* argument and the call returns
1170 *thrd_success*.
- 1171 • It calls *mtx_timedlock()* with *m* as the *mtx* argument and the call returns
1172 *thrd_success*.
- 1173 • It calls *cond_wait()* with *m* as the *mtx* argument and the call returns *thrd_success*.
- 1174 • It calls *cond_timedwait()* with *m* as the *mtx* argument and the call returns
1175 *thrd_success* or *thrd_timedout*.

1176 The thread shall remain the owner of *m* until one of the following occurs:

- 1177 • It executes *mtx_unlock()* with *m* as the *mtx* argument.
- 1178 • It blocks in a call to *cond_wait()* with *m* as the *mtx* argument.
- 1179 • It blocks in a call to *cond_timedwait()* with *m* as the *mtx* argument.

1180 Ref 7.26.4

1181 On page 514 line 17980 section 2.9.3 Thread Mutexes, change:

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

1184 to:

1185 Robust mutexes provide a means to enable the implementation to notify other threads in the
1186 event of a process terminating while one of its threads holds a lock on a mutex of type
1187 **pthread_mutex_t**.

1188 Ref 7.26.5

1189 On page 517 line 18085 section 2.9.5 Thread Cancellation, change:

1190 The thread cancellation mechanism allows a thread to terminate the execution of any other
1191 thread in the process in a controlled manner.

1192 to:

1193 The thread cancellation mechanism allows a thread to terminate the execution of any thread
1194 in the process, except for threads created using *thrd_create()*, in a controlled manner.

1195 Ref 7.26.3, 7.26.5.6

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

1198 *cnd_timedwait()*, *cnd_wait()*, *thrd_join()*, *thrd_sleep()*

1199 Ref 7.26.5

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

1201 Each thread maintains a list of cancellation cleanup handlers.

1202 to:

1203 Each thread that was not created using *thrd_create()* maintains a list of cancellation cleanup
1204 handlers.

1205 Ref 7.26.6.1

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

1207 as described for *pthread_key_create()*

1208 to:

1209 as described for *pthread_key_create()* and *tss_create()*

1210 Ref 7.26

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

1213 For ISO C functions declared in **<threads.h>**, the above requirements shall apply as if
1214 condition variables of type **cnd_t** and mutexes of type **mtx_t** have a process-shared attribute
1215 that is set to **PTHREAD_PROCESS_PRIVATE**.

1216 Ref 7.26.3

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

1218 **pthread_cond_t**

1219 to

1220 **pthread_cond_t, cnd_t**

1221 Ref 7.26.6, 7.26.4

1222 On page 547 line 19281 section 2.12.1 Defined Types, change:

1223 **pthread_key_t**

1224 **pthread_mutex_t**

1225 to

1226 **pthread_key_t, tss_t**
1227 **pthread_mutex_t, mtx_t**

1228 Ref 7.26.2.1

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

1230 **pthread_once_t**

1231 to

1232 **pthread_once_t, once_flag**

1233 Ref 7.26.5

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

1235 **pthread_t**

1236 to

1237 **pthread_t, thrd_t**

1238 Ref 7.3.9.3

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

1240 **NAME**

1241 CMPLX — make a complex value

1242 **SYNOPSIS**

1243 #include <complex.h>

1244 double complex CMPLX(double x, double y);

1245 float complex CMPLXF(float x, float y);

1246 long double complex CMPLXL(long double x, long double y);

1247 **DESCRIPTION**

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

1251 The CMPLX macros shall expand to an expression of the specified complex type, with the
1252 real part having the (converted) value of x and the imaginary part having the (converted)
1253 value of y . The resulting expression shall be suitable for use as an initializer for an object
1254 with static or thread storage duration, provided both arguments are likewise suitable.

1255 **RETURN VALUE**

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

1257 These macros shall behave as if the implementation supported imaginary types and the
1258 definitions were:

```
1259     #define CMLX(x, y) ((double complex)((double)(x) + \  
1260         _Imaginary_I * (double)(y)))  
1261     #define CMLXF(x, y) ((float complex)((float)(x) + \  
1262         _Imaginary_I * (float)(y)))  
1263     #define CMLXL(x, y) ((long double complex)((long double)(x) + \  
1264         _Imaginary_I * (long double)(y)))
```

1265 **ERRORS**

1266 No errors are defined.

1267 **EXAMPLES**

1268 None.

1269 **APPLICATION USAGE**

1270 None.

1271 **RATIONALE**

1272 None.

1273 **FUTURE DIRECTIONS**

1274 None.

1275 **SEE ALSO**

1276 XBD <complex.h>

1277 **CHANGE HISTORY**

1278 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1279 Ref 7.22.4.5 para 1

1280 On page 553 line 19384 section `_Exit()`, change:

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

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

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

1284 to:

```
1285     _Noreturn void _Exit(int status);
```

```
1286     #include <unistd.h>
```

```
1287     _Noreturn void _exit(int status);
```

1288 Ref 7.22.4.5 para 2

1289 On page 553 line 19396 section `_Exit()`, change:

1290 shall not call functions registered with `atexit()` nor any registered signal handlers

1291 to:

1292 shall not call functions registered with `atexit()` nor `at_quick_exit()`, nor any registered signal

1293 handlers

1294 Ref (none)

1295 On page 557 line 19562 section `_Exit()`, change:

1296 The ISO/IEC 9899: 1999 standard adds the `_Exit()` function

1297 to:

1298 The ISO/IEC 9899: 1999 standard added the `_Exit()` function

1299 Ref 7.22.4.3, 7.22.4.7

1300 On page 557 line 19568 section `_Exit()`, add `at_quick_exit` and `quick_exit` to the SEE ALSO section.

1301 Ref 7.22.4.1 para 1

1302 On page 565 line 19761 section `abort()`, change:

1303 `void abort(void);`

1304 to:

1305 `_Noreturn void abort(void);`

1306 Ref (none)

1307 On page 565 line 19785 section `abort()`, change:

1308 The ISO/IEC 9899: 1999 standard requires the `abort()` function to be async-signal-safe.

1309 to:

1310 The ISO/IEC 9899: 1999 standard required (and the current standard still requires) the

1311 `abort()` function to be async-signal-safe.

1312 Ref 7.22.3.1

1313 On page 597 line 20771 insert the following new `aligned_alloc()` section:

1314 **NAME**

1315 `aligned_alloc` — allocate memory with a specified alignment

1316 **SYNOPSIS**

1317 `#include <stdlib.h>`

1318 `void *aligned_alloc(size_t alignment, size_t size);`

1319 **DESCRIPTION**

1320 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1321 Any conflict between the requirements described here and the ISO C standard is

1322 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1323 The `aligned_alloc()` function shall allocate unused space for an object whose alignment is

1324 specified by `alignment`, whose size in bytes is specified by `size` and whose value is

1325 indeterminate.

1326 The order and contiguity of storage allocated by successive calls to *aligned_alloc()* is
1327 unspecified. Each such allocation shall yield a pointer to an object disjoint from any other
1328 object. The pointer returned shall point to the start (lowest byte address) of the allocated
1329 space. If the value of *alignment* is not a valid alignment supported by the implementation, a
1330 null pointer shall be returned. If the space cannot be allocated, a null pointer shall be
1331 returned. If the size of the space requested is 0, the behavior is implementation-defined:
1332 either a null pointer shall be returned to indicate an error, or the behavior shall be as if the
1333 size were some non-zero value, except that the behavior is undefined if the returned pointer
1334 is used to access an object.

1335 For purposes of determining the existence of a data race, *aligned_alloc()* shall behave as
1336 though it accessed only memory locations accessible through its arguments and not other
1337 static duration storage. The function may, however, visibly modify the storage that it
1338 allocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*,
1339 *[ADV]posix_memalign()*,*[/ADV]* and *realloc()* that allocate or deallocate a particular region
1340 of memory shall occur in a single total order (see *[xref to XBD 4.12.1]*), and each such
1341 deallocation call shall synchronize with the next allocation (if any) in this order.

1342 **RETURN VALUE**

1343 Upon successful completion with *size* not equal to 0, *aligned_alloc()* shall return a pointer to
1344 the allocated space. If *size* is 0, either:

- 1345 • A null pointer shall be returned *[CX]* and *errno* may be set to an implementation-
1346 defined value,*[/CX]* or
- 1347 • A pointer to the allocated space shall be returned. The application shall ensure that
1348 the pointer is not used to access an object.

1349 Otherwise, it shall return a null pointer *[CX]* and set *errno* to indicate the error $[/CX]$.

1350 **ERRORS**

1351 The *aligned_alloc()* function shall fail if:

1352 *[CX][EINVAL]* The value of *alignment* is not a valid alignment supported by the
1353 implementation.

1354 *[ENOMEM]* Insufficient storage space is available.*[/CX]*

1355 **EXAMPLES**

1356 None.

1357 **APPLICATION USAGE**

1358 None.

1359 **RATIONALE**

1360 None.

1361 **FUTURE DIRECTIONS**

1362 None.

1363 **SEE ALSO**

1364 *calloc, free, getrlimit, malloc, posix_memalign, realloc*

1365 XBD <stdlib.h>

1366 **CHANGE HISTORY**

1367 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1368 Ref 7.27.3, 7.1.4 para 5

1369 On page 600 line 20911 section *asctime()*, change:

1370 [CX]The *asctime()* function need not be thread-safe.[/CX]

1371 to:

1372 The *asctime()* function need not be thread-safe; however, *asctime()* shall avoid data races
1373 with all functions other than itself, *ctime()*, *gmtime()* and *localtime()*.

1374 Ref 7.22.4.3

1375 On page 618 line 21380 insert the following new *at_quick_exit()* section:

1376 **NAME**

1377 *at_quick_exit* — register a function to be called from *quick_exit()*

1378 **SYNOPSIS**

1379 `#include <stdlib.h>`

1380 `int at_quick_exit(void (*func)(void));`

1381 **DESCRIPTION**

1382 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1383 Any conflict between the requirements described here and the ISO C standard is
1384 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1385 The *at_quick_exit()* function shall register the function pointed to by *func*, to be called
1386 without arguments should *quick_exit()* be called. It is unspecified whether a call to the
1387 *at_quick_exit()* function that does not happen before the *quick_exit()* function is called will
1388 succeed.

1389 At least 32 functions can be registered with *at_quick_exit()*.

1390 [CX]After a successful call to any of the *exec* functions, any functions previously registered
1391 by *at_quick_exit()* shall no longer be registered.[/CX]

1392 **RETURN VALUE**

1393 Upon successful completion, *at_quick_exit()* shall return 0; otherwise, it shall return a non-
1394 zero value.

1395 **ERRORS**

1396 No errors are defined.

1397 **EXAMPLES**

1398 None.

1399 **APPLICATION USAGE**

1400 The *at_quick_exit()* function registrations are distinct from the *atexit()* registrations, so
1401 applications might need to call both registration functions with the same argument.

1402 The functions registered by a call to *at_quick_exit()* must return to ensure that all registered
1403 functions are called.

1404 The application should call *sysconf()* to obtain the value of {ATEXIT_MAX}, the number of
1405 functions that can be registered. There is no way for an application to tell how many
1406 functions have already been registered with *at_quick_exit()*.

1407 Since the behavior is undefined if the *quick_exit()* function is called more than once,
1408 portable applications calling *at_quick_exit()* must ensure that the *quick_exit()* function is not
1409 called when the functions registered by the *at_quick_exit()* function are called.

1410 If a function registered by the *at_quick_exit()* function is called and a portable application
1411 needs to stop further *quick_exit()* processing, it must call the *_exit()* function or the *_Exit()*
1412 function or one of the functions which cause abnormal process termination.

1413 **RATIONALE**

1414 None.

1415 **FUTURE DIRECTIONS**

1416 None.

1417 **SEE ALSO**

1418 *atexit, exec, exit, quick_exit, sysconf*

1419 XBD <stdlib.h>

1420 **CHANGE HISTORY**

1421 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1422 Ref 7.22.4.3

1423 On page 618 line 21381 section *atexit()*, change:

1424 *atexit* — register a function to run at process termination

1425 to:

1426 *atexit* — register a function to be called from *exit()* or after return from *main()*

1427 Ref 7.22.4.2 para 2, 7.22.4.3

1428 On page 618 line 21389 section *atexit()*, change:

1429 The *atexit()* function shall register the function pointed to by *func*, to be called without
1430 arguments at normal program termination. At normal program termination, all functions
1431 registered by the *atexit()* function shall be called, in the reverse order of their registration,
1432 except that a function is called after any previously registered functions that had already
1433 been called at the time it was registered. Normal termination occurs either by a call to *exit()*
1434 or a return from *main()*.

1435 to:

1436 The *atexit()* function shall register the function pointed to by *func*, to be called without
1437 arguments from *exit()*, or after return from the initial call to *main()*, or on the last thread
1438 termination. If the *exit()* function is called, it is unspecified whether a call to the *atexit()*
1439 function that does not happen before *exit()* is called will succeed.

1440 [Note to reviewers: the part about all registered functions being called in reverse order is duplicated](#)
1441 [on the *exit\(\)* page and is not needed here.](#)

1442 Ref 7.22.4.2 para 2

1443 On page 618 line 21405 section *atexit()*, insert a new first APPLICATION USAGE paragraph:

1444 The *atexit()* function registrations are distinct from the *at_quick_exit()* registrations, so
1445 applications might need to call both registration functions with the same argument.

1446 Ref 7.22.4.3

1447 On page 618 line 21410 section *atexit()*, change:

1448 Since the behavior is undefined if the *exit()* function is called more than once, portable
1449 applications calling *atexit()* must ensure that the *exit()* function is not called at normal
1450 process termination when all functions registered by the *atexit()* function are called.

1451 All functions registered by the *atexit()* function are called at normal process termination,
1452 which occurs by a call to the *exit()* function or a return from *main()* or on the last thread
1453 termination, when the behavior is as if the implementation called *exit()* with a zero argument
1454 at thread termination time.

1455 If, at normal process termination, a function registered by the *atexit()* function is called and a
1456 portable application needs to stop further *exit()* processing, it must call the *_exit()* function
1457 or the *_Exit()* function or one of the functions which cause abnormal process termination.

1458 to:

1459 Since the behavior is undefined if the *exit()* function is called more than once, portable
1460 applications calling *atexit()* must ensure that the *exit()* function is not called when the
1461 functions registered by the *atexit()* function are called.

1462 If a function registered by the *atexit()* function is called and a portable application needs to
1463 stop further *exit()* processing, it must call the *_exit()* function or the *_Exit()* function or one
1464 of the functions which cause abnormal process termination.

1465 Ref 7.22.4.3

1466 On page 619 line 21425 section *atexit()*, add *at_quick_exit* to the SEE ALSO section.

1467 Ref 7.16

1468 On page 624 line 21548 insert the following new *atomic_**() sections:

1469 **NAME**

1470 *atomic_compare_exchange_strong*, *atomic_compare_exchange_strong_explicit*,
1471 *atomic_compare_exchange_weak*, *atomic_compare_exchange_weak_explicit* — atomically

1472 compare and exchange the values of two objects

1473 SYNOPSIS

```
1474 #include <stdatomic.h>
1475 _Bool atomic_compare_exchange_strong(volatile A *object,
1476     C *expected, C desired);
1477 _Bool atomic_compare_exchange_strong_explicit(volatile A *object,
1478     C *expected, C desired, memory_order success,
1479     memory_order failure);
1480 _Bool atomic_compare_exchange_weak(volatile A *object,
1481     C *expected, C desired);
1482 _Bool atomic_compare_exchange_weak_explicit(volatile A *object,
1483     C *expected, C desired, memory_order success,
1484     memory_order failure);
```

1485 DESCRIPTION

1486 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1487 Any conflict between the requirements described here and the ISO C standard is
1488 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1489 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1490 `<stdatomic.h>` header nor support these generic functions.

1491 The `atomic_compare_exchange_strong_explicit()` generic function shall atomically compare
1492 the contents of the memory pointed to by *object* for equality with that pointed to by
1493 *expected*, and if true, shall replace the contents of the memory pointed to by *object*
1494 with *desired*, and if false, shall update the contents of the memory pointed to by *expected*
1495 with that pointed to by *object*. This operation shall be an atomic read-modify-write operation
1496 (see [xref to XBD 4.12.1]). If the comparison is true, memory shall be affected according to
1497 the value of *success*, and if the comparison is false, memory shall be affected according to
1498 the value of *failure*. The application shall ensure that *failure* is not
1499 `memory_order_release` nor `memory_order_acq_rel`, and shall ensure that *failure* is
1500 no stronger than *success*.

1501 The `atomic_compare_exchange_strong()` generic function shall be equivalent to
1502 `atomic_compare_exchange_strong_explicit()` called with *success* and *failure* both set to
1503 `memory_order_seq_cst`.

1504 The `atomic_compare_exchange_weak_explicit()` generic function shall be equivalent to
1505 `atomic_compare_exchange_strong_explicit()`, except that the compare-and-exchange
1506 operation may fail spuriously. That is, even when the contents of memory referred to by
1507 *expected* and *object* are equal, it may return zero and store back to *expected* the same
1508 memory contents that were originally there.

1509 The `atomic_compare_exchange_weak()` generic function shall be equivalent to
1510 `atomic_compare_exchange_weak_explicit()` called with *success* and *failure* both set to
1511 `memory_order_seq_cst`.

1512 RETURN VALUE

1513 These generic functions shall return the result of the comparison.

1514 ERRORS

1515 No errors are defined.

1516 **EXAMPLES**

1517 None.

1518 **APPLICATION USAGE**

1519 A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will
1520 be in a loop. For example:

```
1521       exp = atomic_load(&cur);  
1522       do {  
1523           des = function(exp);  
1524       } while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

1525 When a compare-and-exchange is in a loop, the weak version will yield better performance
1526 on some platforms. When a weak compare-and-exchange would require a loop and a strong
1527 one would not, the strong one is preferable.

1528 **RATIONALE**

1529 None.

1530 **FUTURE DIRECTIONS**

1531 None.

1532 **SEE ALSO**

1533 XBD Section 4.12.1, <stdatomic.h>

1534 **CHANGE HISTORY**

1535 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1536 **NAME**

1537 atomic_exchange, atomic_exchange_explicit — atomically exchange the value of an object

1538 **SYNOPSIS**

```
1539       #include <stdatomic.h>  
1540       C atomic_exchange(volatile A *object, C desired);  
1541       C atomic_exchange_explicit(volatile A *object,  
1542           C desired, memory_order order);
```

1543 **DESCRIPTION**

1544 [**CX**] The functionality described on this reference page is aligned with the ISO C standard.
1545 Any conflict between the requirements described here and the ISO C standard is
1546 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/**CX**]

1547 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1548 <stdatomic.h> header nor support these generic functions.

1549 The `atomic_exchange_explicit()` generic function shall atomically replace the value pointed
1550 to by *object* with *desired*. This operation shall be an atomic read-modify-write operation (see
1551 [xref to XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1552 The `atomic_exchange()` generic function shall be equivalent to `atomic_exchange_explicit()`
1553 called with *order* set to `memory_order_seq_cst`.

1554 **RETURN VALUE**

1555 These generic functions shall return the value pointed to by *object* immediately before the
1556 effects.

1557 **ERRORS**

1558 No errors are defined.

1559 **EXAMPLES**

1560 None.

1561 **APPLICATION USAGE**

1562 None.

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

1576 **SYNOPSIS**

```
1577 #include <stdatomic.h>
1578 C   atomic_fetch_add(volatile A *object, M operand);
1579 C   atomic_fetch_add_explicit(volatile A *object, M operand,
1580                               memory_order order);
1581 C   atomic_fetch_and(volatile A *object, M operand);
1582 C   atomic_fetch_and_explicit(volatile A *object, M operand,
1583                               memory_order order);
1584 C   atomic_fetch_or(volatile A *object, M operand);
1585 C   atomic_fetch_or_explicit(volatile A *object, M operand,
1586                               memory_order order);
1587 C   atomic_fetch_sub(volatile A *object, M operand);
1588 C   atomic_fetch_sub_explicit(volatile A *object, M operand,
1589                               memory_order order);
1590 C   atomic_fetch_xor(volatile A *object, M operand);
1591 C   atomic_fetch_xor_explicit(volatile A *object, M operand,
1592                               memory_order order);
```

1593 **DESCRIPTION**

1594 [**CX**] The functionality described on this reference page is aligned with the ISO C standard.
1595 Any conflict between the requirements described here and the ISO C standard is

1596 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1597 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1598 `<stdatomic.h>` header nor support these generic functions.

1599 The `atomic_fetch_add_explicit()` generic function shall atomically replace the value pointed
1600 to by *object* with the result of adding *operand* to this value. This operation shall be an
1601 atomic read-modify-write operation (see [xref to XBD 4.12.1]). Memory shall be affected
1602 according to the value of *order*.

1603 The `atomic_fetch_add()` generic function shall be equivalent to `atomic_fetch_add_explicit()`
1604 called with *order* set to `memory_order_seq_cst`.

1605 The other `atomic_fetch_*`() generic functions shall be equivalent to
1606 `atomic_fetch_add_explicit()` if their name ends with *explicit*, or to `atomic_fetch_add()` if it
1607 does not, respectively, except that they perform the computation indicated in their name,
1608 instead of addition:

1609 *sub* subtraction
1610 *or* bitwise inclusive OR
1611 *xor* bitwise exclusive OR
1612 *and* bitwise AND

1613 For addition and subtraction, the application shall ensure that *A* is an atomic integer type or
1614 an atomic pointer type and is not **atomic_bool**. For the other operations, the application
1615 shall ensure that *A* is an atomic integer type and is not **atomic_bool**.

1616 For signed integer types, the computation shall silently wrap around on overflow; there are
1617 no undefined results. For pointer types, the result can be an undefined address, but the
1618 computations otherwise have no undefined behavior.

1619 RETURN VALUE

1620 These generic functions shall return the value pointed to by *object* immediately before the
1621 effects.

1622 ERRORS

1623 No errors are defined.

1624 EXAMPLES

1625 None.

1626 APPLICATION USAGE

1627 The operation of these generic functions is nearly equivalent to the operation of the
1628 corresponding compound assignment operators `+=`, `-=`, etc. The only differences are that the
1629 compound assignment operators are not guaranteed to operate atomically, and the value
1630 yielded by a compound assignment operator is the updated value of the object, whereas the
1631 value returned by these generic functions is the previous value of the atomic object.

1632 RATIONALE

1633 None.

1634 FUTURE DIRECTIONS

1635 None.

1636 **SEE ALSO**

1637 XBD Section 4.12.1, <**stdatomic.h**>

1638 **CHANGE HISTORY**

1639 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1640 **NAME**

1641 `atomic_flag_clear`, `atomic_flag_clear_explicit` — clear an atomic flag

1642 **SYNOPSIS**

```
1643 #include <stdatomic.h>
1644 void atomic_flag_clear(volatile atomic_flag *object);
1645 void atomic_flag_clear_explicit(
1646     volatile atomic_flag *object, memory_order order);
```

1647 **DESCRIPTION**

1648 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1649 Any conflict between the requirements described here and the ISO C standard is
1650 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1651 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1652 <**stdatomic.h**> header nor support these functions.

1653 The `atomic_flag_clear_explicit()` function shall atomically place the atomic flag pointed to
1654 by `object` into the clear state. Memory shall be affected according to the value of `order`,
1655 which the application shall ensure is not `memory_order_acquire` nor
1656 `memory_order_acq_rel`.

1657 The `atomic_flag_clear()` function shall be equivalent to `atomic_flag_clear_explicit()` called
1658 with `order` set to `memory_order_seq_cst`.

1659 **RETURN VALUE**

1660 These functions shall not return a value.

1661 **ERRORS**

1662 No errors are defined.

1663 **EXAMPLES**

1664 None.

1665 **APPLICATION USAGE**

1666 None.

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_test_and_set`, `atomic_flag_test_and_set_explicit` — test and set an atomic flag

1677 **SYNOPSIS**

```
1678 #include <stdatomic.h>
1679 _Bool atomic_flag_test_and_set(volatile atomic_flag *object);
1680 _Bool atomic_flag_test_and_set_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_test_and_set_explicit()` function shall atomically place the atomic flag
1689 pointed to by *object* into the set state and return the value corresponding to the immediately
1690 preceding state. This operation shall be an atomic read-modify-write operation (see [xref to
1691 XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1692 The `atomic_flag_test_and_set()` function shall be equivalent to
1693 `atomic_flag_test_and_set_explicit()` called with *order* set to `memory_order_seq_cst`.

1694 **RETURN VALUE**

1695 These functions shall return the value that corresponds to the state of the atomic flag
1696 immediately before the effects. The return value true shall correspond to the set state and the
1697 return value false shall correspond to the clear state.

1698 **ERRORS**

1699 No errors are defined.

1700 **EXAMPLES**

1701 None.

1702 **APPLICATION USAGE**

1703 None.

1704 **RATIONALE**

1705 None.

1706 **FUTURE DIRECTIONS**

1707 None.

1708 **SEE ALSO**

1709 XBD Section 4.12.1, <stdatomic.h>

1710 **CHANGE HISTORY**

1711 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1712 **NAME**

1713 `atomic_init` — initialize an atomic object

1714 **SYNOPSIS**

```
1715 #include <stdatomic.h>
1716 void atomic_init(volatile A *obj, C value);
```

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 this generic function.

1723 The `atomic_init()` generic function shall initialize the atomic object pointed to by `obj` to the
1724 value `value`, while also initializing any additional state that the implementation might need
1725 to carry for the atomic object.

1726 Although this function initializes an atomic object, it does not avoid data races; concurrent
1727 access to the variable being initialized, even via an atomic operation, constitutes a data race.

1728 **RETURN VALUE**

1729 The `atomic_init()` generic function shall not return a value.

1730 **ERRORS**

1731 No errors are defined.

1732 **EXAMPLES**

```
1733 atomic_int guide;
1734 atomic_init(&guide, 42);
```

1735 **APPLICATION USAGE**

1736 None.

1737 **RATIONALE**

1738 None.

1739 **FUTURE DIRECTIONS**

1740 None.

1741 **SEE ALSO**

1742 XBD `<stdatomic.h>`

1743 **CHANGE HISTORY**

1744 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1745 **NAME**

1746 `atomic_is_lock_free` — indicate whether or not atomic operations are lock-free

1747 **SYNOPSIS**

```
1748 #include <stdatomic.h>
1749 _Bool atomic_is_lock_free(const volatile A *obj);
```

1750 **DESCRIPTION**

1751 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1752 Any conflict between the requirements described here and the ISO C standard is
1753 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1754 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1755 `<stdatomic.h>` header nor support this generic function.

1756 The `atomic_is_lock_free()` generic function shall indicate whether or not atomic operations
1757 on objects of the type pointed to by `obj` are lock-free; `obj` can be a null pointer.

1758 **RETURN VALUE**

1759 The `atomic_is_lock_free()` generic function shall return a non-zero value if and only if
1760 atomic operations on objects of the type pointed to by `obj` are lock-free. During the lifetime
1761 of the calling process, the result of the lock-free query shall be consistent for all pointers of
1762 the same type.

1763 **ERRORS**

1764 No errors are defined.

1765 **EXAMPLES**

1766 None.

1767 **APPLICATION USAGE**

1768 None.

1769 **RATIONALE**

1770 Operations that are lock-free should also be address-free. That is, atomic operations on the
1771 same memory location via two different addresses will communicate atomically. The
1772 implementation should not depend on any per-process state. This restriction enables
1773 communication via memory mapped into a process more than once and memory shared
1774 between two processes.

1775 **FUTURE DIRECTIONS**

1776 None.

1777 **SEE ALSO**

1778 XBD `<stdatomic.h>`

1779 **CHANGE HISTORY**

1780 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1781 **NAME**

1782 `atomic_load`, `atomic_load_explicit` — atomically obtain the value of an object

1783 **SYNOPSIS**

```
1784     #include <stdatomic.h>
1785     C atomic_load(const volatile A *object);
1786     C atomic_load_explicit(const volatile A *object,
1787         memory_order order);
```

1788 **DESCRIPTION**

1789 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1790 Any conflict between the requirements described here and the ISO C standard is
1791 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1792 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1793 `<stdatomic.h>` header nor support these generic functions.

1794 The `atomic_load_explicit()` generic function shall atomically obtain the value pointed to by
1795 `object`. Memory shall be affected according to the value of `order`, which the application shall
1796 ensure is not `memory_order_release` nor `memory_order_acq_rel`.

1797 The `atomic_load()` generic function shall be equivalent to `atomic_load_explicit()` called with
1798 `order` set to `memory_order_seq_cst`.

1799 **RETURN VALUE**

1800 These generic functions shall return the value pointed to by `object`.

1801 **ERRORS**

1802 No errors are defined.

1803 **EXAMPLES**

1804 None.

1805 **APPLICATION USAGE**

1806 None.

1807 **RATIONALE**

1808 None.

1809 **FUTURE DIRECTIONS**

1810 None.

1811 **SEE ALSO**

1812 XBD Section 4.12.1, `<stdatomic.h>`

1813 **CHANGE HISTORY**

1814 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1815 **NAME**

1816 `atomic_signal_fence`, `atomic_thread_fence` — fence operations

1817 **SYNOPSIS**

```
1818     #include <stdatomic.h>
1819     void atomic_signal_fence(memory_order order);
1820     void atomic_thread_fence(memory_order order);
```

1821 **DESCRIPTION**

1822 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1823 Any conflict between the requirements described here and the ISO C standard is
1824 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1825 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1826 `<stdatomic.h>` header nor support these functions.

1827 The `atomic_signal_fence()` and `atomic_thread_fence()` functions provide synchronization
1828 primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A
1829 fence with acquire semantics is called an *acquire fence*; a fence with release semantics is
1830 called a *release fence*.

1831 A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X*
1832 and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X*
1833 modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* or a value written
1834 by any side effect in the hypothetical release sequence *X* would head if it were a release
1835 operation.

1836 A release fence *A* synchronizes with an atomic operation *B* that performs an acquire
1837 operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is
1838 sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by
1839 any side effect in the hypothetical release sequence *X* would head if it were a release
1840 operation.

1841 An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with
1842 an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced
1843 before *B* and reads the value written by *A* or a value written by any side effect in the release
1844 sequence headed by *A*.

1845 Depending on the value of *order*, the operation performed by `atomic_thread_fence()` shall:

- 1846 • have no effects, if *order* is equal to `memory_order_relaxed`;
- 1847 • be an acquire fence, if *order* is equal to `memory_order_acquire` or
1848 `memory_order_consume`;
- 1849 • be a release fence, if *order* is equal to `memory_order_release`;
- 1850 • be both an acquire fence and a release fence, if *order* is equal to
1851 `memory_order_acq_rel`;
- 1852 • be a sequentially consistent acquire and release fence, if *order* is equal to
1853 `memory_order_seq_cst`.

1854 The `atomic_signal_fence()` function shall be equivalent to `atomic_thread_fence()`, except
1855 that the resulting ordering constraints shall be established only between a thread and a signal
1856 handler executed in the same thread.

1857 **RETURN VALUE**

1858 These functions shall not return a value.

1859 **ERRORS**

1860 No errors are defined.

1861 **EXAMPLES**

1862 None.

1863 **APPLICATION USAGE**

1864 The *atomic_signal_fence()* function can be used to specify the order in which actions
1865 performed by the thread become visible to the signal handler. Implementation reorderings of
1866 loads and stores are inhibited in the same way as with *atomic_thread_fence()*, but the
1867 hardware fence instructions that *atomic_thread_fence()* would have inserted are not
1868 emitted.

1869 **RATIONALE**

1870 None.

1871 **FUTURE DIRECTIONS**

1872 None.

1873 **SEE ALSO**

1874 XBD Section 4.12.1, <**stdatomic.h**>

1875 **CHANGE HISTORY**

1876 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1877 **NAME**

1878 *atomic_store*, *atomic_store_explicit* — atomically store a value in an object

1879 **SYNOPSIS**

```
1880 #include <stdatomic.h>  
1881 void atomic_store(volatile A *object, C desired);  
1882 void atomic_store_explicit(volatile A *object, C desired,  
1883 memory_order order);
```

1884 **DESCRIPTION**

1885 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1886 Any conflict between the requirements described here and the ISO C standard is
1887 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1888 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1889 `<stdatomic.h>` header nor support these generic functions.

1890 The *atomic_store_explicit()* generic function shall atomically replace the value pointed to by
1891 *object* with the value of *desired*. Memory shall be affected according to the value of *order*,
1892 which the application shall ensure is not `memory_order_acquire`,
1893 `memory_order_consume`, nor `memory_order_acq_rel`.

1894 The *atomic_store()* generic function shall be equivalent to *atomic_store_explicit()* called
1895 with *order* set to `memory_order_seq_cst`.

1896 **RETURN VALUE**

1897 These generic functions shall not return a value.

1898 **ERRORS**

1899 No errors are defined.

1900 **EXAMPLES**

1901 None.

1902 **APPLICATION USAGE**

1903 None.

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 Ref 7.28.1, 7.1.4 para 5

1913 On page 633 line 21891 insert a new `c16rtomb()` section:

1914 **NAME**

1915 `c16rtomb`, `c32rtomb` — convert a Unicode character code to a character (restartable)

1916 **SYNOPSIS**

1917

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

1918

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

1919

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

1920

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

1921

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

1922 **DESCRIPTION**

1923 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1924 Any conflict between the requirements described here and the ISO C standard is

1925 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1926 If *s* is a null pointer, the `c16rtomb()` function shall be equivalent to the call:

1927

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

1928 where *buf* is an internal buffer.

1929 If *s* is not a null pointer, the `c16rtomb()` function shall determine the number of bytes needed
1930 to represent the character that corresponds to the wide character given by *c16* (including any
1931 shift sequences), and store the resulting bytes in the array whose first element is pointed to
1932 by *s*. At most {`MB_CUR_MAX`} bytes shall be stored. If *c16* is a null wide character, a null
1933 byte shall be stored, preceded by any shift sequence needed to restore the initial shift state;

1934 the resulting state described shall be the initial conversion state.

1935 If *ps* is a null pointer, the *c16rtomb()* function shall use its own internal **mbstate_t** object,
1936 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
1937 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current
1938 conversion state of the associated character sequence.

1939 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

1940 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

1941 The *c32rtomb()* function shall behave the same way as *c16rtomb()* except that the second
1942 parameter shall be an object of type **char32_t** instead of **char16_t**. References to *c16* in the
1943 above description shall apply as if they were *c32* when they are being read as describing
1944 *c32rtomb()*.

1945 If called with a null *ps* argument, the *c16rtomb()* function need not be thread-safe; however,
1946 such calls shall avoid data races with calls to *c16rtomb()* with a non-null argument and with
1947 calls to all other functions.

1948 If called with a null *ps* argument, the *c32rtomb()* function need not be thread-safe; however,
1949 such calls shall avoid data races with calls to *c32rtomb()* with a non-null argument and with
1950 calls to all other functions.

1951 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
1952 calls *c16rtomb()* or *c32rtomb()* with a null pointer for *ps*.

1953 **RETURN VALUE**

1954 These functions shall return the number of bytes stored in the array object (including any
1955 shift sequences). When *c16* or *c32* is not a valid wide character, an encoding error shall
1956 occur. In this case, the function shall store the value of the macro [EILSEQ] in *errno* and
1957 shall return (**size_t**)-1; the conversion state is unspecified.

1958 **ERRORS**

1959 These function shall fail if:

1960 [EILSEQ] An invalid wide-character code is detected.

1961 These functions may fail if:

1962 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

1963 **EXAMPLES**

1964 None.

1965 **APPLICATION USAGE**

1966 None.

1967 **RATIONALE**

1968 None.

1969 **FUTURE DIRECTIONS**

1970 None.

- 1971 **SEE ALSO**
- 1972 *mbrtoc16*
- 1973 XBD <**uchar.h**>
- 1974 **CHANGE HISTORY**
- 1975 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.
- 1976 Ref G.6 para 6, F.10.4.3, F.10.4.2, F.10 para 11
- 1977 On page 633 line 21905 section *cabs()*, add:
- 1978 [MXC]*cabs(x + iy)*, *cabs(y + ix)*, and *cabs(x - iy)* shall return exactly the same value.
- 1979 If z is $\pm 0 \pm i0$, $+0$ shall be returned.
- 1980 If the real or imaginary part of z is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned, even if the other part is NaN.
- 1981 If the real or imaginary part of z is NaN and the other part is not $\pm\text{Inf}$, NaN shall be returned.
- 1982 [/MXC]
- 1983 Ref G.6.1.1
- 1984 On page 634 line 21935 section *cacos()*, add:
- 1985 [MXC]*cacos(conj(z))*, *cacosf(conjf(z))* and *cacosl(conjl(z))* shall return exactly the same
- 1986 value as *conj(cacos(z))*, *conjf(cacosf(z))* and *conjl(cacosl(z))*, respectively, including for the
- 1987 special values of z below.
- 1988 If z is $\pm 0 + i0$, $\pi/2 - i0$ shall be returned.
- 1989 If z is $\pm 0 + i\text{NaN}$, $\pi/2 + i\text{NaN}$ shall be returned.
- 1990 If z is $x + i\text{Inf}$ where x is finite, $\pi/2 - i\text{Inf}$ shall be returned.
- 1991 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
- 1992 floating-point exception may be raised.
- 1993 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $\pi - i\text{Inf}$ shall be returned.
- 1994 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 - i\text{Inf}$ shall be returned.
- 1995 If z is $-\text{Inf} + i\text{Inf}$, $3\pi/4 - i\text{Inf}$ shall be returned.
- 1996 If z is $+\text{Inf} + i\text{Inf}$, $\pi/4 - i\text{Inf}$ shall be returned.
- 1997 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
- 1998 is unspecified.
- 1999 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
- 2000 point exception may be raised.
- 2001 If z is $\text{NaN} + i\text{Inf}$, $\text{NaN} - i\text{Inf}$ shall be returned.

- 2002 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} - i\text{NaN}$ shall be returned.[/MXC]
- 2003 Ref G.6.2.1
- 2004 On page 635 line 21966 section `cacosh()`, add:
- 2005 [MXC]`cacosh(conj(z))`, `cacoshf(conjf(z))` and `cacoshl(conjl(z))` shall return exactly the same
2006 value as `conj(cacosh(z))`, `conjf(cacoshf(z))` and `conjl(cacoshl(z))`, respectively, including for
2007 the special values of z below.
- 2008 If z is $\pm 0 + i0$, $+0 + i\pi/2$ shall be returned.
- 2009 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.
- 2010 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
2011 unspecified.
- 2012 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
2013 floating-point exception may be raised.
- 2014 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.
- 2015 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.
- 2016 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.
- 2017 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.
- 2018 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2019 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2020 point exception may be raised.
- 2021 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2022 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2023 Ref 7.26.2.1
- 2024 On page 637 line 21989 insert the following new `call_once()` section:
- 2025 **NAME**
- 2026 `call_once` — dynamic package initialization
- 2027 **SYNOPSIS**
- 2028 `#include <threads.h>`
- 2029 `void call_once(once_flag *flag, void (*init_routine)(void));`
2030 `once_flag flag = ONCE_FLAG_INIT;`
- 2031 **DESCRIPTION**
- 2032 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2033 Any conflict between the requirements described here and the ISO C standard is
2034 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2035 The *call_once()* function shall use the **once_flag** pointed to by *flag* to ensure that
2036 *init_routine* is called exactly once, the first time the *call_once()* function is called with that
2037 value of *flag*. Completion of an effective call to the *call_once()* function shall synchronize
2038 with all subsequent calls to the *call_once()* function with the same value of *flag*.

2039 [CX]The *call_once()* function is not a cancellation point. However, if *init_routine* is a
2040 cancellation point and is canceled, the effect on *flag* shall be as if *call_once()* was never
2041 called.

2042 If the call to *init_routine* is terminated by a call to *longjmp()* or *siglongjmp()*, the behavior is
2043 undefined.

2044 The behavior of *call_once()* is undefined if *flag* has automatic storage duration or is not
2045 initialized by `ONCE_FLAG_INIT`.

2046 The *call_once()* function shall not be affected if the calling thread executes a signal handler
2047 during the call.[/CX]

2048 **RETURN VALUE**

2049 The *call_once()* function shall not return a value.

2050 **ERRORS**

2051 No errors are defined.

2052 **EXAMPLES**

2053 None.

2054 **APPLICATION USAGE**

2055 If *init_routine* recursively calls *call_once()* with the same *flag*, the recursive call will not call
2056 the specified *init_routine*, and thus the specified *init_routine* will not complete, and thus the
2057 recursive call to *call_once()* will not return. Use of *longjmp()* or *siglongjmp()* within an
2058 *init_routine* to jump to a point outside of *init_routine* prevents *init_routine* from returning.

2059 **RATIONALE**

2060 For dynamic library initialization in a multi-threaded process, if an initialization flag is used
2061 the flag needs to be protected against modification by multiple threads simultaneously
2062 calling into the library. This can be done by using a statically-initialized mutex. However,
2063 the better solution is to use *call_once()* or *pthread_once()* which are designed for exactly
2064 this purpose, for example:

```
2065 #include <threads.h>  
2066 static once_flag random_is_initialized = ONCE_FLAG_INIT;  
2067 extern void initialize_random(void);
```

```
2068 int random_function()  
2069 {  
2070     call_once(&random_is_initialized, initialize_random);  
2071     ...  
2072     /* Operations performed after initialization. */  
2073 }
```

2074 The *call_once()* function is not affected by signal handlers for the reasons stated in [xref to
2075 XRAT B.2.3].

2076 **FUTURE DIRECTIONS**

2077 None.

2078 **SEE ALSO**

2079 *pthread_once*

2080 XBD Section 4.12.2, <**threads.h**>

2081 **CHANGE HISTORY**

2082 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2083 Ref 7.22.3 para 1

2084 On page 637 line 22002 section *calloc()*, change:

2085 a pointer to any type of object

2086 to:

2087 a pointer to any type of object with a fundamental alignment requirement

2088 Ref 7.22.3 para 1

2089 On page 637 line 22007 section *calloc()*, change:

2090 either a null pointer shall be returned, or ...

2091 to:

2092 either a null pointer shall be returned to indicate an error, or ...

2093 Ref 7.22.3 para 2

2094 On page 637 line 22008 section *calloc()*, add a new paragraph:

2095 For purposes of determining the existence of a data race, *calloc()* shall behave as though it
2096 accessed only memory locations accessible through its arguments and not other static
2097 duration storage. The function may, however, visibly modify the storage that it allocates.
2098 Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] and
2099 *realloc()* that allocate or deallocate a particular region of memory shall occur in a single total
2100 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
2101 next allocation (if any) in this order.

2102 Ref 7.22.3.1

2103 On page 637 line 22029 section *calloc()*, add *aligned_alloc* to the SEE ALSO section.

2104 Ref G.6 para 6, F.10.1.4, F.10 para 11

2105 On page 639 line 22055 section *carg()*, add:

2106 [MXC]If z is $-0 \pm i0$, $\pm\pi$ shall be returned.

- 2107 If z is $+0 \pm i0$, ± 0 shall be returned.
- 2108 If z is $x \pm i0$ where x is negative, $\pm\pi$ shall be returned.
- 2109 If z is $x \pm i0$ where x is positive, ± 0 shall be returned.
- 2110 If z is $\pm 0 + iy$ where y is negative, $-\pi/2$ shall be returned.
- 2111 If z is $\pm 0 + iy$ where y is positive, $\pi/2$ shall be returned.
- 2112 If z is $-\text{Inf} \pm iy$ where y is positive and finite, $\pm\pi$ shall be returned.
- 2113 If z is $+\text{Inf} \pm iy$ where y is positive and finite, ± 0 shall be returned.
- 2114 If z is $x \pm i\text{Inf}$ where x is finite, $\pm\pi/2$ shall be returned.
- 2115 If z is $-\text{Inf} \pm i\text{Inf}$, $\pm 3\pi/4$ shall be returned.
- 2116 If z is $+\text{Inf} \pm i\text{Inf}$, $\pm\pi/4$ shall be returned.
- 2117 If the real or imaginary part of z is NaN, NaN shall be returned.[/MXC]
- 2118 Ref G.6 para 7, G.6.2.2
- 2119 On page 640 line 22086 section `casin()`, add:
- 2120 [MXC]`casin(conj(iz))`, `casinf(conjf(iz))` and `casinl(conjl(iz))` shall return exactly the same
- 2121 value as `conj(casin(iz))`, `conjf(casinf(iz))` and `conjl(casinl(iz))`, respectively, and `casin(-iz)`,
- 2122 `casinf(-iz)` and `casinl(-iz)` shall return exactly the same value as `-casin(iz)`, `-casinf(iz)` and
- 2123 `-casinl(iz)`, respectively, including for the special values of iz below.
- 2124 If iz is $+0 + i0$, $-i (0 + i0)$ shall be returned.
- 2125 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i (+\text{Inf} + i\pi/2)$ shall be returned.
- 2126 If iz is $x + i\text{NaN}$ where x is finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the invalid
- 2127 floating-point exception may be raised.
- 2128 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i (+\text{Inf} + i0)$ shall be returned.
- 2129 If iz is $+\text{Inf} + i\text{Inf}$, $-i (+\text{Inf} + i\pi/4)$ shall be returned.
- 2130 If iz is $+\text{Inf} + i\text{NaN}$, $-i (+\text{Inf} + i\text{NaN})$ shall be returned.
- 2131 If iz is $\text{NaN} + i0$, $-i (\text{NaN} + i0)$ shall be returned.
- 2132 If iz is $\text{NaN} + iy$ where y is non-zero and finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
- 2133 invalid floating-point exception may be raised.
- 2134 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
- 2135 result is unspecified.
- 2136 If iz is $\text{NaN} + i\text{NaN}$, $-i (\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2137 Ref G.6 para 7
2138 On page 640 line 22094 section `casin()`, change RATIONALE from:

2139 None.

2140 to:

2141 The MXC special cases for `casin()` are derived from those for `casinh()` by applying the
2142 formula $\text{casin}(z) = -i \text{casinh}(iz)$.

2143 Ref G.6.2.2
2144 On page 641 line 22118 section `casinh()`, add:

2145 [MXC]`casinh(conj(z))`, `casinhf(conjf(z))` and `casinhl(conjl(z))` shall return exactly the same
2146 value as `conj(casinh(z))`, `conjf(casinhf(z))` and `conjl(casinhl(z))`, respectively, and `casinh(-z)`,
2147 `casinhf(-z)` and `casinhl(-z)` shall return exactly the same value as $-\text{casinh}(z)$, $-\text{casinhf}(z)$
2148 and $-\text{casinhl}(z)$, respectively, including for the special values of z below.

2149 If z is $+0 + i0$, $0 + i0$ shall be returned.

2150 If z is $x + i\text{Inf}$ where x is positive-signed and finite, $+\text{Inf} + i\pi/2$ shall be returned.

2151 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2152 point exception may be raised.

2153 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2154 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

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

2156 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2157 If z is $\text{NaN} + iy$ where y is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2158 floating-point exception may be raised.

2159 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
2160 unspecified.

2161 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2162 Ref G.6 para 7, G.6.2.3
2163 On page 643 line 22157 section `catan`, add:

2164 [MXC]`catan(conj(iz))`, `catanf(conjf(iz))` and `catanl(conjl(iz))` shall return exactly the same
2165 value as `conj(catan(iz))`, `conjf(catanf(iz))` and `conjl(catanl(iz))`, respectively, and `catan(-iz)`,
2166 `catanf(-iz)` and `catanl(-iz)` shall return exactly the same value as $-\text{catan}(iz)$, $-\text{catanf}(iz)$ and
2167 $-\text{catanl}(iz)$, respectively, including for the special values of iz below.

2168 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.

- 2169 If iz is $+0 + iNaN$, $-i (+0 + iNaN)$ shall be returned.
- 2170 If iz is $+1 + i0$, $-i (+Inf + i0)$ shall be returned and the divide-by-zero floating-point
2171 exception shall be raised.
- 2172 If iz is $x + iInf$ where x is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2173 If iz is $x + iNaN$ where x is non-zero and finite, $-i (NaN + iNaN)$ shall be returned and the
2174 invalid floating-point exception may be raised.
- 2175 If iz is $+Inf + iy$ where y is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2176 If iz is $+Inf + iInf$, $-i (+0 + i\pi/2)$ shall be returned.
- 2177 If iz is $+Inf + iNaN$, $-i (+0 + iNaN)$ shall be returned.
- 2178 If iz is $NaN + iy$ where y is finite, $-i (NaN + iNaN)$ shall be returned and the invalid
2179 floating-point exception may be raised.
- 2180 If iz is $NaN + iInf$, $-i (\pm 0 + i\pi/2)$ shall be returned; the sign of the imaginary part of the
2181 result is unspecified.
- 2182 If iz is $NaN + iNaN$, $-i (NaN + iNaN)$ shall be returned.[/MXC]
- 2183 Ref G.6 para 7
- 2184 On page 643 line 22165 section `catan()`, change RATIONALE from:
- 2185 None.
- 2186 to:
- 2187 The MXC special cases for `catan()` are derived from those for `catanh()` by applying the
2188 formula $catan(z) = -i catanh(iz)$.
- 2189 Ref G.6.2.3
- 2190 On page 644 line 22189 section `catanh`, add:
- 2191 [MXC]`catanh(conj(z))`, `catanhf(conjf(z))` and `catanhl(conjl(z))` shall return exactly the same
2192 value as `conj(catanh(z))`, `conjf(catanhf(z))` and `conjl(catanhl(z))`, respectively, and
2193 `catanh(-z)`, `catanhf(-z)` and `catanhl(-z)` shall return exactly the same value as $-catanh(z)$,
2194 $-catanhf(z)$ and $-catanhl(z)$, respectively, including for the special values of z below.
- 2195 If z is $+0 + i0$, $+0 + i0$ shall be returned.
- 2196 If z is $+0 + iNaN$, $+0 + iNaN$ shall be returned.
- 2197 If z is $+1 + i0$, $+Inf + i0$ shall be returned and the divide-by-zero floating-point exception
2198 shall be raised.
- 2199 If z is $x + iInf$ where x is positive-signed and finite, $+0 + i\pi/2$ shall be returned.
- 2200 If z is $x + iNaN$ where x is non-zero and finite, $NaN + iNaN$ shall be returned and the invalid

- 2201 floating-point exception may be raised.
- 2202 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\pi/2$ shall be returned.
- 2203 If z is $+\text{Inf} + i\text{Inf}$, $+0 + i\pi/2$ shall be returned.
- 2204 If z is $+\text{Inf} + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.
- 2205 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2206 point exception may be raised.
- 2207 If z is $\text{NaN} + i\text{Inf}$, $\pm 0 + i\pi/2$ shall be returned; the sign of the real part of the result is
2208 unspecified.
- 2209 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2210 Ref G.6 para 7, G.6.2.4
2211 On page 652 line 22426 section `ccos()`, add:
- 2212 [MXC]`ccos(conj(iz))`, `ccosf(conjf(iz))` and `ccosl(conjl(iz))` shall return exactly the same value
2213 as `conj(ccos(iz))`, `conjf(ccosf(iz))` and `conjl(ccosl(iz))`, respectively, and `ccos(-iz)`, `ccosf(-iz)`
2214 and `ccosl(-iz)` shall return exactly the same value as `ccos(iz)`, `ccosf(iz)` and `ccosl(iz)`,
2215 respectively, including for the special values of iz below.
- 2216 If iz is $+0 + i0$, $1 + i0$ shall be returned.
- 2217 If iz is $+0 + i\text{Inf}$, $\text{NaN} \pm i0$ shall be returned and the invalid floating-point exception shall be
2218 raised; the sign of the imaginary part of the result is unspecified.
- 2219 If iz is $+0 + i\text{NaN}$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2220 unspecified.
- 2221 If iz is $x + i\text{Inf}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2222 floating-point exception shall be raised.
- 2223 If iz is $x + i\text{NaN}$ where x is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the
2224 invalid floating-point exception may be raised.
- 2225 If iz is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.
- 2226 If iz is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.
- 2227 If iz is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2228 shall be raised; the sign of the real part of the result is unspecified.
- 2229 If iz is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2230 If iz is $\text{NaN} + i0$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2231 unspecified.
- 2232 If iz is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2233 invalid floating-point exception may be raised.

2234 If iz is NaN + i NaN, NaN + i NaN shall be returned.[/MXC]

2235 Ref G.6 para 7
2236 On page 652 line 22434 section `ccos()`, change RATIONALE from:

2237 None.

2238 to:

2239 The MXC special cases for `ccos()` are derived from those for `ccosh()` by applying the
2240 formula $ccos(z) = ccosh(iz)$.

2241 Ref G.6.2.4
2242 On page 653 line 22455 section `ccosh()`, add:

2243 [MXC]`ccosh(conj(z))`, `ccoshf(conjf(z))` and `ccoshl(conjl(z))` shall return exactly the same
2244 value as `conj(ccosh(z))`, `conjf(ccoshf(z))` and `conjl(ccoshl(z))`, respectively, and `ccosh(-z)`,
2245 `ccoshf(-z)` and `ccoshl(-z)` shall return exactly the same value as `ccosh(z)`, `ccoshf(z)` and
2246 `ccoshl(z)`, respectively, including for the special values of z below.

2247 If z is $+0 + i0$, $1 + i0$ shall be returned.

2248 If z is $+0 + i\text{Inf}$, NaN $\pm i0$ shall be returned and the invalid floating-point exception shall be
2249 raised; the sign of the imaginary part of the result is unspecified.

2250 If z is $+0 + i\text{NaN}$, NaN $\pm i0$ shall be returned; the sign of the imaginary part of the result is
2251 unspecified.

2252 If z is $x + i\text{Inf}$ where x is non-zero and finite, NaN + $i\text{NaN}$ shall be returned and the invalid
2253 floating-point exception shall be raised.

2254 If z is $x + i\text{NaN}$ where x is non-zero and finite, NaN + $i\text{NaN}$ shall be returned and the invalid
2255 floating-point exception may be raised.

2256 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2257 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2258 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2259 shall be raised; the sign of the real part of the result is unspecified.

2260 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2261 If z is NaN + $i0$, NaN $\pm i0$ shall be returned; the sign of the imaginary part of the result is
2262 unspecified.

2263 If z is NaN + iy where y is any non-zero number, NaN + $i\text{NaN}$ shall be returned and the
2264 invalid floating-point exception may be raised.

2265 If z is NaN + $i\text{NaN}$, NaN + $i\text{NaN}$ shall be returned.[/MXC]

2266 Ref F.10.6.1 para 4
2267 On page 655 line 22489 section `ceil()`, add a new paragraph:

2268 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2269 arguments.[/MX]

2270 Ref F.10.6.1 para 2
2271 On page 655 line 22491 section `ceil()`, change:

2272 [MX]The result shall have the same sign as x .[/MX]

2273 to:

2274 [MX]The returned value shall be independent of the current rounding direction mode and
2275 shall have the same sign as x .[/MX]

2276 Ref F.10.6.1 para 4
2277 On page 655 line 22504 section `ceil()`, delete from APPLICATION USAGE:

2278 These functions may raise the inexact floating-point exception if the result differs in value
2279 from the argument.

2280 Ref G.6.3.1
2281 On page 657 line 22539 section `cexp()`, add:

2282 [MXC]`cexp(conj(z))`, `cexpf(conjf(z))` and `cexpl(conjl(z))` shall return exactly the same value
2283 as `conj(cexp(z))`, `conjf(cexpf(z))` and `conjl(cexpl(z))`, respectively, including for the special
2284 values of z below.

2285 If z is $\pm 0 + i0$, $1 + i0$ shall be returned.

2286 If z is $x + i\text{Inf}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-point
2287 exception shall be raised.

2288 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2289 point exception may be raised.

2290 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2291 If z is $-\text{Inf} + iy$ where y is finite, $+0 (\cos(y) + i\sin(y))$ shall be returned.

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

2293 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
2294 result are unspecified.

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

2297 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
2298 result are unspecified.

2299 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
2300 unspecified.

2301 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2302 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2303 invalid floating-point exception may be raised.

2304 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2305 Ref 7.26.5.7
2306 On page 679 line 23268 section `clock_getres()`, change:

2307 including the `nanosleep()` function

2308 to:

2309 including the `nanosleep()` and `thrd_sleep()` functions

2310 Ref G.6.3.2
2311 On page 687 line 23495 section `clog()`, add:

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

2315 If z is $-0 + i0$, $-\text{Inf} + i\pi$ shall be returned and the divide-by-zero floating-point exception
2316 shall be raised.

2317 If z is $+0 + i0$, $-\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2318 shall be raised.

2319 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2320 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2321 point exception may be raised.

2322 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2323 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2324 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

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

2326 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2327 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2328 point exception may be raised.

2329 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2330 If z is NaN + i NaN, NaN + i NaN shall be returned.[/MXC]

2331 Ref 7.26.3

2332 On page 698 line 23854 insert the following new `cond_*`() sections:

2333 [Note to reviewers: changes to `cond_broadcast` and `cond_signal` may be needed depending on the](#)
2334 [outcome of Mantis bug 609.](#)

2335 NAME

2336 `cond_broadcast`, `cond_signal` — broadcast or signal a condition

2337 SYNOPSIS

2338 `#include <threads.h>`

2339 `int cond_broadcast(cond_t *cond);`

2340 `int cond_signal(cond_t *cond);`

2341 DESCRIPTION

2342 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2343 Any conflict between the requirements described here and the ISO C standard is
2344 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2345 The `cond_broadcast()` function shall unblock all of the threads that are blocked on the
2346 condition variable pointed to by `cond` at the time of the call.

2347 The `cond_signal()` function shall unblock one of the threads that are blocked on the condition
2348 variable pointed to by `cond` at the time of the call (if any threads are blocked on `cond`).

2349 If no threads are blocked on the condition variable pointed to by `cond` at the time of the call,
2350 these functions shall have no effect and shall return `thrd_success`.

2351 [CX]If more than one thread is blocked on a condition variable, the scheduling policy shall
2352 determine the order in which threads are unblocked. When each thread unblocked as a result
2353 of a `cond_broadcast()` or `cond_signal()` returns from its call to `cond_wait()` or `cond_timedwait()`,
2354 the thread shall own the mutex with which it called `cond_wait()` or `cond_timedwait()`. The
2355 thread(s) that are unblocked shall contend for the mutex according to the scheduling policy
2356 (if applicable), and as if each had called `mtx_lock()`.

2357 The `cond_broadcast()` and `cond_signal()` functions can be called by a thread whether or not it
2358 currently owns the mutex that threads calling `cond_wait()` or `cond_timedwait()` have associated
2359 with the condition variable during their waits; however, if predictable scheduling behavior is
2360 required, then that mutex shall be locked by the thread calling `cond_broadcast()` or
2361 `cond_signal()`.

2362 These functions shall not be affected if the calling thread executes a signal handler during
2363 the call.[/CX]

2364 The behavior is undefined if the value specified by the `cond` argument to `cond_broadcast()` or
2365 `cond_signal()` does not refer to an initialized condition variable.

2366 RETURN VALUE

2367 These functions shall return `thrd_success` on success, or `thrd_error` if the request
2368 could not be honored.

2369 **ERRORS**

2370 No errors are defined.

2371 **EXAMPLES**

2372 None.

2373 **APPLICATION USAGE**

2374 See the APPLICATION USAGE section for *pthread_cond_broadcast()*, substituting
2375 *cond_broadcast()* for *pthread_cond_broadcast()* and *cond_signal()* for *pthread_cond_signal()*.

2376 **RATIONALE**

2377 As for *pthread_cond_broadcast()* and *pthread_cond_signal()*, spurious wakeups may occur
2378 with *cond_broadcast()* and *cond_signal()*, necessitating that applications code a predicate-
2379 testing-loop around the condition wait. (See the RATIONALE section for
2380 *pthread_cond_broadcast()*.)

2381 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2382 B.2.3].

2383 **FUTURE DIRECTIONS**

2384 None.

2385 **SEE ALSO**

2386 *cond_destroy*, *cond_timedwait*, *pthread_cond_broadcast*

2387 XBD Section 4.12.2, <**threads.h**>

2388 **CHANGE HISTORY**

2389 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2390 **NAME**

2391 *cond_destroy*, *cond_init* — destroy and initialize condition variables

2392 **SYNOPSIS**

2393 `#include <threads.h>`

2394 `void cond_destroy(cond_t *cond);`

2395 `int cond_init(cond_t *cond);`

2396 **DESCRIPTION**

2397 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2398 Any conflict between the requirements described here and the ISO C standard is
2399 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2400 The *cond_destroy()* function shall release all resources used by the condition variable pointed
2401 to by *cond*. It shall be safe to destroy an initialized condition variable upon which no threads
2402 are currently blocked. Attempting to destroy a condition variable upon which other threads
2403 are currently blocked results in undefined behavior. A destroyed condition variable object
2404 can be reinitialized using *cond_init()*; the results of otherwise referencing the object after it
2405 has been destroyed are undefined. The behavior is undefined if the value specified by the
2406 *cond* argument to *cond_destroy()* does not refer to an initialized condition variable.

2407 The *cond_init()* function shall initialize a condition variable. If it succeeds it shall set the
2408 variable pointed to by *cond* to a value that uniquely identifies the newly initialized condition
2409 variable. Attempting to initialize an already initialized condition variable results in
2410 undefined behavior. A thread that calls *cond_wait()* on a newly initialized condition variable
2411 shall block.

2412 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
2413 further requirements.

2414 These functions shall not be affected if the calling thread executes a signal handler during
2415 the call.[/CX]

2416 **RETURN VALUE**

2417 The *cond_destroy()* function shall not return a value.

2418 The *cond_init()* function shall return *thrd_success* on success, or *thrd_nomem* if no
2419 memory could be allocated for the newly created condition, or *thrd_error* if the request
2420 could not be honored.

2421 **ERRORS**

2422 See RETURN VALUE.

2423 **EXAMPLES**

2424 None.

2425 **APPLICATION USAGE**

2426 None.

2427 **RATIONALE**

2428 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2429 B.2.3].

2430 **FUTURE DIRECTIONS**

2431 None.

2432 **SEE ALSO**

2433 *cond_broadcast*, *cond_timedwait*

2434 XBD <**threads.h**>

2435 **CHANGE HISTORY**

2436 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2437 **NAME**

2438 *cond_timedwait*, *cond_wait* — wait on a condition

2439 **SYNOPSIS**

```
2440 #include <threads.h>
2441 int cond_timedwait(cond_t * restrict cond, mtx_t * restrict mtx,
2442                  const struct timespec * restrict ts);
2443 int cond_wait(cond_t *cond, mtx_t *mtx);
```

2444 **DESCRIPTION**

2445 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2446 Any conflict between the requirements described here and the ISO C standard is
2447 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2448 The *cnd_timedwait()* function shall atomically unlock the mutex pointed to by *mtx* and block
2449 until the condition variable pointed to by *cond* is signaled by a call to *cnd_signal()* or to
2450 *cnd_broadcast()*, or until after the TIME_UTC-based calendar time pointed to by *ts*, or until
2451 it is unblocked due to an unspecified reason.

2452 The *cnd_wait()* function shall atomically unlock the mutex pointed to by *mtx* and block until
2453 the condition variable pointed to by *cond* is signaled by a call to *cnd_signal()* or to
2454 *cnd_broadcast()*, or until it is unblocked due to an unspecified reason.

2455 [CX]Atomically here means "atomically with respect to access by another thread to the
2456 mutex and then the condition variable". That is, if another thread is able to acquire the mutex
2457 after the about-to-block thread has released it, then a subsequent call to *cnd_broadcast()* or
2458 *cnd_signal()* in that thread shall behave as if it were issued after the about-to-block thread
2459 has blocked.[/CX]

2460 When the calling thread becomes unblocked, these functions shall lock the mutex pointed to
2461 by *mtx* before they return. The application shall ensure that the mutex pointed to by *mtx* is
2462 locked by the calling thread before it calls these functions.

2463 When using condition variables there is always a Boolean predicate involving shared
2464 variables associated with each condition wait that is true if the thread should proceed.
2465 Spurious wakeups from the *cnd_timedwait()* and *cnd_wait()* functions may occur. Since the
2466 return from *cnd_timedwait()* or *cnd_wait()* does not imply anything about the value of this
2467 predicate, the predicate should be re-evaluated upon such return.

2468 When a thread waits on a condition variable, having specified a particular mutex to either
2469 the *cnd_timedwait()* or the *cnd_wait()* operation, a dynamic binding is formed between that
2470 mutex and condition variable that remains in effect as long as at least one thread is blocked
2471 on the condition variable. During this time, the effect of an attempt by any thread to wait on
2472 that condition variable using a different mutex is undefined. Once all waiting threads have
2473 been unblocked (as by the *cnd_broadcast()* operation), the next wait operation on
2474 that condition variable shall form a new dynamic binding with the mutex specified by that
2475 wait operation. Even though the dynamic binding between condition variable and mutex
2476 might be removed or replaced between the time a thread is unblocked from a wait on the
2477 condition variable and the time that it returns to the caller or begins cancellation cleanup, the
2478 unblocked thread shall always re-acquire the mutex specified in the condition wait operation
2479 call from which it is returning.

2480 [CX]A condition wait (whether timed or not) is a cancellation point. When the cancelability
2481 type of a thread is set to PTHREAD_CANCEL_DEFERRED, a side-effect of acting upon a
2482 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired
2483 before calling the first cancellation cleanup handler. The effect is as if the thread were
2484 unblocked, allowed to execute up to the point of returning from the call to *cnd_timedwait()*
2485 or *cnd_wait()*, but at that point notices the cancellation request and instead of returning to
2486 the caller of *cnd_timedwait()* or *cnd_wait()*, starts the thread cancellation activities, which
2487 includes calling cancellation cleanup handlers.

2488 A thread that has been unblocked because it has been canceled while blocked in a call to
2489 *cond_timedwait()* or *cond_wait()* shall not consume any condition signal that may be directed
2490 concurrently at the condition variable if there are other threads blocked on the condition
2491 variable.[/CX]

2492 When *cond_timedwait()* times out, it shall nonetheless release and re-acquire the mutex
2493 referenced by *mutex*, and may consume a condition signal directed concurrently at the
2494 condition variable.

2495 [CX]These functions shall not be affected if the calling thread executes a signal handler
2496 during the call, except that if a signal is delivered to a thread waiting for a condition
2497 variable, upon return from the signal handler either the thread shall resume waiting for the
2498 condition variable as if it was not interrupted, or it shall return *thrd_success* due to
2499 spurious wakeup.[/CX]

2500 The behavior is undefined if the value specified by the *cond* or *mtx* argument to these
2501 functions does not refer to an initialized condition variable or an initialized mutex object,
2502 respectively.

2503 RETURN VALUE

2504 The *cond_timedwait()* function shall return *thrd_success* upon success, or
2505 *thrd_timedout* if the time specified in the call was reached without acquiring the
2506 requested resource, or *thrd_error* if the request could not be honored.

2507 The *cond_wait()* function shall return *thrd_success* upon success or *thrd_error* if the
2508 request could not be honored.

2509 ERRORS

2510 See RETURN VALUE.

2511 EXAMPLES

2512 None.

2513 APPLICATION USAGE

2514 None.

2515 RATIONALE

2516 These functions are not affected by signal handlers (except as stated in the DESCRIPTION)
2517 for the reasons stated in [xref to XRAT B.2.3].

2518 FUTURE DIRECTIONS

2519 None.

2520 SEE ALSO

2521 *cond_broadcast*, *cond_destroy*, *timespec_get*

2522 XBD Section 4.12.2, <**threads.h**>

2523 CHANGE HISTORY

2524 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2525 Ref F.10.8.1 para 2
2526 On page 705 line 24155 section `copysign()`, add a new paragraph:

2527 [MX]The returned value shall be exact and shall be independent of the current rounding
2528 direction mode.[/MX]

2529 Ref G.6.4.1 para 1
2530 On page 711 line 24308 section `cpow()`, add a new paragraph:

2531 [MXC]These functions shall raise floating-point exceptions if appropriate for the calculation
2532 of the parts of the result, and may also raise spurious floating-point exceptions.[/MXC]

2533 Ref G.6.4.1 footnote 386
2534 On page 711 line 24318 section `cpow()`, change RATIONALE from:

2535 None.

2536 to:

2537 Permitting spurious floating-point exceptions allows `cpow(z, c)` to be implemented as `cexp(c`
2538 `clog(z))` without precluding implementations that treat special cases more carefully.

2539 Ref G.6 para 7, G.6.2.5
2540 On page 718 line 24545 section `csin()`, add:

2541 [MXC]`csin(conj(iz))`, `csinf(conjf(iz))` and `csinl(conjl(iz))` shall return exactly the same value
2542 as `conj(csin(iz))`, `conjf(csinf(iz))` and `conjl(csinl(iz))`, respectively, and `csin(-iz)`, `csinf(-iz)`
2543 and `csinl(-iz)` shall return exactly the same value as `-csin(iz)`, `-csinf(iz)` and `-csinl(iz)`,
2544 respectively, including for the special values of `iz` below.

2545 If `iz` is `+0 + i0`, `-i (+0 + i0)` shall be returned.

2546 If `iz` is `+0 + iInf`, `-i (±0 + iNaN)` shall be returned and the invalid floating-point exception
2547 shall be raised; the sign of the imaginary part of the result is unspecified.

2548 If `iz` is `+0 + iNaN`, `-i (±0 + iNaN)` shall be returned; the sign of the imaginary part of the
2549 result is unspecified.

2550 If `iz` is `x + iInf` where `x` is positive and finite, `-i (NaN + iNaN)` shall be returned and the
2551 invalid floating-point exception shall be raised.

2552 If `iz` is `x + iNaN` where `x` is non-zero and finite, `-i (NaN + iNaN)` shall be returned and the
2553 invalid floating-point exception may be raised.

2554 If `iz` is `+Inf + i0`, `-i (+Inf + i0)` shall be returned.

2555 If `iz` is `+Inf + iy` where `y` is positive and finite, `-iInf (cos(y) + isin(y))` shall be returned.

2556 If `iz` is `+Inf + iInf`, `-i (±Inf + iNaN)` shall be returned and the invalid floating-point exception
2557 shall be raised; the sign of the imaginary part of the result is unspecified.

2558 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
2559 result is unspecified.

2560 If iz is $\text{NaN} + i0$, $-i (\text{NaN} + i0)$ shall be returned.

2561 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
2562 invalid floating-point exception may be raised.

2563 If iz is $\text{NaN} + i\text{NaN}$, $-i (\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2564 Ref G.6 para 7
2565 On page 718 line 24553 section `csin()`, change RATIONALE from:

2566 None.

2567 to:

2568 The MXC special cases for `csin()` are derived from those for `csinh()` by applying the formula
2569 $csin(z) = -i csinh(iz)$.

2570 Ref G.6.2.5
2571 On page 719 line 24574 section `csinh()`, add:

2572 [MXC]`csinh(conj(z))`, `csinhf(conjf(z))` and `csinhl(conjl(z))` shall return exactly the same
2573 value as `conj(csinh(z))`, `conjf(csinhf(z))` and `conjl(csinhl(z))`, respectively, and `csinh(-z)`,
2574 `csinhf(-z)` and `csinhl(-z)` shall return exactly the same value as $-csinh(z)$, $-csinhf(z)$ and
2575 $-csinhl(z)$, respectively, including for the special values of z below.

2576 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2577 If z is $+0 + i\text{Inf}$, $\pm 0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2578 raised; the sign of the real part of the result is unspecified.

2579 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
2580 unspecified.

2581 If z is $x + i\text{Inf}$ where x is positive and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2582 floating-point exception shall be raised.

2583 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
2584 floating-point exception may be raised.

2585 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2586 If z is $+\text{Inf} + iy$ where y is positive and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2587 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2588 shall be raised; the sign of the real part of the result is unspecified.

2589 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
2590 unspecified.

2591 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2592 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2593 invalid floating-point exception may be raised.

2594 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2595 Ref G.6.4.2
2596 On page 721 line 24612 section `csqrt()`, add:

2597 [MXC]`csqrt(conj(z))`, `csqrtf(conjf(z))` and `csqrtl(conjl(z))` shall return exactly the same value
2598 as `conj(csqrt(z))`, `conjf(csqrtf(z))` and `conjl(csqrtl(z))`, respectively, including for the special
2599 values of z below.

2600 If z is $\pm 0 + i0$, $+0 + i0$ shall be returned.

2601 If the imaginary part of z is Inf , $+\text{Inf} + i\text{Inf}$, shall be returned.

2602 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2603 point exception may be raised.

2604 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\text{Inf}$ shall be returned.

2605 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2606 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
2607 is unspecified.

2608 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

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

2611 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2612 Ref G.6 para 7, G.6.2.6
2613 On page 722 line 24641 section `ctan()`, add:

2614 [MXC]`ctan(conj(iz))`, `ctanf(conjf(iz))` and `ctanl(conjl(iz))` shall return exactly the same value
2615 as `conj(ctan(iz))`, `conjf(ctanf(iz))` and `conjl(ctanl(iz))`, respectively, and `ctan(-iz)`, `ctanf(-iz)`
2616 and `ctanl(-iz)` shall return exactly the same value as `-ctan(iz)`, `-ctanf(iz)` and `-ctanl(iz)`,
2617 respectively, including for the special values of iz below.

2618 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.

2619 If iz is $0 + i\text{Inf}$, $-i (0 + i\text{NaN})$ shall be returned and the invalid floating-point exception shall
2620 be raised.

2621 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
2622 invalid floating-point exception shall be raised.

2623 If iz is $0 + i\text{NaN}$, $-i (0 + i\text{NaN})$ shall be returned.

- 2624 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
 2625 invalid floating-point exception may be raised.
- 2626 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i(1 + i0 \sin(2y))$ shall be returned.
- 2627 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
 2628 unspecified.
- 2629 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
 2630 unspecified.
- 2631 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.
- 2632 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
 2633 invalid floating-point exception may be raised.
- 2634 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2635 Ref G.6 para 7
 2636 On page 722 line 24649 section `ctan()`, change RATIONALE from:
- 2637 None.
- 2638 to:
- 2639 The MXC special cases for `ctan()` are derived from those for `ctanh()` by applying the
 2640 formula $ctan(z) = -i ctanh(iz)$.
- 2641 Ref G.6.2.6
 2642 On page 723 line 24670 section `ctanh()`, add:
- 2643 [MXC]`ctanh(conj(z))`, `ctanhf(conjf(z))` and `ctanhl(conjl(z))` shall return exactly the same
 2644 value as `conj(ctanh(z))`, `conjf(ctanhf(z))` and `conjl(ctanhl(z))`, respectively, and `ctanh(-z)`,
 2645 `ctanhf(-z)` and `ctanhl(-z)` shall return exactly the same value as $-ctanh(z)$, $-ctanhf(z)$ and
 2646 $-ctanhl(z)$, respectively, including for the special values of z below.
- 2647 If z is $+0 + i0$, $+0 + i0$ shall be returned.
- 2648 If z is $0 + i\text{Inf}$, $0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
 2649 raised.
- 2650 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
 2651 floating-point exception shall be raised.
- 2652 If z is $0 + i\text{NaN}$, $0 + i\text{NaN}$ shall be returned.
- 2653 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
 2654 floating-point exception may be raised.
- 2655 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $1 + i0 \sin(2y)$ shall be returned.

2656 If z is $+Inf + iInf$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2657 unspecified.

2658 If z is $+Inf + iNaN$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2659 unspecified.

2660 If z is $NaN + i0$, $NaN + i0$ shall be returned.

2661 If z is $NaN + iy$ where y is any non-zero number, $NaN + iNaN$ shall be returned and the
2662 invalid floating-point exception may be raised.

2663 If z is $NaN + iNaN$, $NaN + iNaN$ shall be returned.[/MXC]

2664 Ref 7.27.3, 7.1.4 para 5
2665 On page 727 line 24774 section `ctime()`, change:

2666 [CX]The `ctime()` function need not be thread-safe.[/CX]

2667 to:
2668 The `ctime()` function need not be thread-safe; however, `ctime()` shall avoid data races with all
2669 functions other than itself, `asctime()`, `gmtime()` and `localtime()`.

2670 Ref 7.5 para 2
2671 On page 781 line 26447 section `errno`, change:

2672 The lvalue `errno` is used by many functions to return error values.

2673 to:

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

2676 Ref 7.5 para 3
2677 On page 781 line 26449 section `errno`, change:

2678 The value of `errno` shall be defined only after a call to a function for which it is explicitly
2679 stated to be set and until it is changed by the next function call or if the application assigns it
2680 a value.

2681 to:

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

2686 Ref 7.5 para 2
2687 On page 781 line 26456 section `errno`, delete:

2688 It is unspecified whether `errno` is a macro or an identifier declared with external linkage.

2689 Ref 7.22.4.4 para 2

2690 On page 796 line 27057 section `exit()`, add a new (unshaded) paragraph:

2691 The `exit()` function shall cause normal process termination to occur. No functions registered
2692 by the `at_quick_exit()` function shall be called. If a process calls the `exit()` function more
2693 than once, or calls the `quick_exit()` function in addition to the `exit()` function, the behavior is
2694 undefined.

2695 Ref 7.22.4.4 para 2

2696 On page 796 line 27068 section `exit()`, delete:

2697 If `exit()` is called more than once, the behavior is undefined.

2698 Ref 7.22.4.3, 7.22.4.7

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

2700 Ref F.10.4.2 para 2

2701 On page 804 line 27323 section `fabs()`, add a new paragraph:

2702 [MX]The returned value shall be exact and shall be independent of the current rounding
2703 direction mode.[/MX]

2704 Ref 7.21.2 para 7,8

2705 On page 874 line 29483 section `flockfile()`, change:

2706 These functions shall provide for explicit application-level locking of stdio (**FILE ***)
2707 objects.

2708 to:

2709 These functions shall provide for explicit application-level locking of the locks associated
2710 with standard I/O streams (see [xref to 2.5]).

2711 Ref 7.21.2 para 7,8

2712 On page 874 line 29499 section `flockfile()`, delete:

2713 All functions that reference (**FILE ***) objects, except those with names ending in `_unlocked`,
2714 shall behave as if they use `flockfile()` and `funlockfile()` internally to obtain ownership of these
2715 (**FILE ***) objects.

2716 Ref F.10.6.2 para 3

2717 On page 876 line 29560 section `floor()`, add a new paragraph:

2718 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2719 arguments.[/MX]

2720 Ref F.10.6.2 para 2

2721 On page 876 line 29562 section `floor()`, change:

2722 [MX]The result shall have the same sign as `x`.[/MX]

2723 to:

2724 [MX]The returned value shall be independent of the current rounding direction mode and
2725 shall have the same sign as *x*.[/MX]

2726 Ref F.10.6.2 para 3
2727 On page 876 line 29576 section `floor()`, delete from APPLICATION USAGE:

2728 These functions may raise the inexact floating-point exception if the result differs in value
2729 from the argument.

2730 Ref F.10.9.2 para 2
2731 On page 880 line 29695 section `fmax()`, add a new paragraph:

2732 [MX]The returned value shall be exact and shall be independent of the current rounding
2733 direction mode.[/MX]

2734 Ref F.10.9.3 para 2
2735 On page 884 line 29844 section `fmin()`, add a new paragraph:

2736 [MX]The returned value shall be exact and shall be independent of the current rounding
2737 direction mode.[/MX]

2738 Ref F.10.7.1 para 2
2739 On page 885 line 29892 section `fmod()`, change:

2740 [MXX]If the correct value would cause underflow, and is representable, a range error may
2741 occur and the correct value shall be returned.[/MXX]

2742 to:

2743 [MX]When subnormal results are supported, the returned value shall be exact and shall be
2744 independent of the current rounding direction mode.[/MX]

2745 Ref 7.21.5.3 para 5
2746 On page 892 line 30117 section `fopen()`, change:

2747 [CX]The functionality described on this reference page is aligned with the ISO C standard.
2748 Any conflict between the requirements described here and the ISO C standard is
2749 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2750 to:

2751 [CX]Except for the “exclusive access” requirement (see below), the functionality described
2752 on this reference page is aligned with the ISO C standard. Any other conflict between the
2753 requirements described here and the ISO C standard is unintentional. This volume of
2754 POSIX.1-202x defers to the ISO C standard for all `fopen()` functionality except in relation to
2755 “exclusive access”.[/CX]

2756 Ref 7.21.5.3 para 5
2757 On page 892 line 30132 section `fopen()`, after applying bug 411, change:

2758 'x' If specified with a prefix beginning with 'w' [CX]or 'a'[/CX], then the function shall

2759 fail if the file already exists, [CX]as if by the O_EXCL flag to *open()*. If specified
2760 with a prefix beginning with 'r', this modifier shall have no effect.[/CX]

2761 to:

2762 'x' If specified with a prefix beginning with 'w' [CX]or 'a'[/CX], then the function shall
2763 fail if the file already exists or cannot be created; if the file does not exist and can be
2764 created, it shall be created with [CX]an implementation-defined form of[/CX]
2765 exclusive (also known as non-shared) access, [CX]if supported by the underlying file
2766 system, provided the resulting file permissions are the same as they would be without
2767 the 'x' modifier. If specified with a prefix beginning with 'r', this modifier shall have
2768 no effect.[/CX]

2769 **Note:** The ISO C standard requires exclusive access “to the extent that the underlying file
2770 system supports exclusive access”, but does not define what it means by this. Taken
2771 at face value—that systems must do whatever they are capable of, at the file system
2772 level, in order to exclude access by others—this would require POSIX.1 systems to
2773 set the file permissions in a way that prevents access by other users and groups.
2774 Consequently, this volume of POSIX.1-202x does not defer to the ISO C standard as
2775 regards the “exclusive access” requirement.

2776 [Note to reviewers: This “exclusive access” requirement may be clarified in C2x, in which case the](#)
2777 [above text may be changed to match the proposed C2x text.](#)

2778 Ref 7.21.5.3 para 3
2779 On page 892 line 30144 section *fopen()*, change:

2780 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and ...

2781 to:

2782 If the first character in *mode* is *w* or *a*, and ...

2783 Ref 7.21.5.3 para 3,5
2784 On page 892 line 30148 section *fopen()*, change:

2785 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously
2786 exist, the *fopen()* function shall create a file as if it called the *creat()* function with a value
2787 appropriate for the *path* argument interpreted from *pathname* and a value of S_IRUSR |
2788 S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH for the *mode* argument.

2789 to:

2790 If the first character in *mode* is *w* or *a*, and the file did not previously exist, the *fopen()*
2791 function shall create a file as if it called the *open()* function with a value appropriate for the
2792 *path* argument interpreted from *pathname*, a value for the *oflag* argument as specified below,
2793 and a value of S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH for
2794 the third argument.

2795 Ref 7.21.5.3 para 5
2796 On page 893 line 30158 section *fopen()*, change:

2797 The file descriptor ...

2798 to:

2799 If the first character in *mode* is *r*, or the suffix of *mode* does not include *x*, the file descriptor
2800 ...

2801 Ref (none; see bug 411)

2802 On page 893 line 30160 section `fopen()`, change the first column heading from:

2803 ***fopen()* Mode**

2804 to:

2805 ***fopen()* Mode Without Suffix**

2806 and add the following text after the table:

2807 with the addition of the `O_CLOEXEC` flag if the suffix of *mode* includes *e*.

2808 Ref 7.21.5.3 para 5

2809 On page 893 line 30166 section `fopen()`, add the following new paragraphs:

2810 [CX]If the first character in *mode* is *w* or *a*, the suffix of *mode* includes *x*, and the underlying
2811 file system does not support exclusive access, then the file descriptor associated with the
2812 opened stream shall be allocated and opened as if by a call to `open()` with the following
2813 flags:

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

2814 with the addition of the `O_CLOEXEC` flag if the suffix of *mode* includes *e*.

2815 If the first character in *mode* is *w* or *a*, the suffix of *mode* includes *x*, and the underlying file
2816 system supports exclusive access, then the file descriptor associated with the opened stream
2817 shall be allocated and opened as if by a call to `open()` with the above flags or with the above
2818 flags ORed with an implementation-defined file creation flag if necessary to enable
2819 exclusive access (see above).[CX]

2820 [Note to reviewers: The above change may need to be updated depending on whether WG14 clarify](#)
2821 [the “exclusive access” requirement.](#)

2822 Ref 7.21.5.3 para 5

2823 On page 895 line 30236 section `fopen()`, change APPLICATION USAGE from:

2824 None.

2825 to:

2826 If an application needs to create a file in a way that fails if the file already exists, and either
2827 requires that it does not have exclusive access to the file or does not need exclusive access, it
2828 should use *open()* with the *O_CREAT* and *O_EXCL* flags instead of using *fopen()* with an *x*
2829 in the *mode*. A stream can then be created, if needed, by calling *fdopen()* on the file
2830 descriptor returned by *open()*.

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

2833 Ref 7.21.5.3 para 5

2834 On page 895 line 30238 section *fopen()*, after applying bug 411, change:

2835 The *x* mode suffix character was added by C1x only for files opened with a mode string
2836 beginning with *w*.

2837 to:

2838 The *x* mode suffix character is specified by the ISO C standard only for files opened with a
2839 mode string beginning with *w*.

2840 and then add two new paragraphs after the one that starts with the above text:

2841 When the last character in *mode* is *x*, the ISO C standard requires that the file is created with
2842 exclusive access to the extent that the underlying system supports exclusive access.
2843 Although POSIX.1 does not specify any method of enabling exclusive access, it allows for
2844 the existence of an implementation-defined file creation flag that enables it. Note that it must
2845 be a file creation flag, not a file access mode flag (that is, one that is included in
2846 *O_ACCMODE*) or a file status flag, so that it does not affect the value returned by *fcntl()*
2847 with *F_GETFL*. On implementations that have such a flag, if support for it is file system
2848 dependent and exclusive access is requested when using *fopen()* to create a file on a file
2849 system that does not support it, the flag must not be used if it would cause *fopen()* to fail.

2850 Some implementations support mandatory file locking as a means of enabling exclusive
2851 access to a file. Locks are set in the normal way, but instead of only preventing others from
2852 setting conflicting locks they prevent others from accessing the contents of the locked part
2853 of the file in a way that conflicts with the lock. However, unless the implementation has a
2854 way of setting a whole-file write lock on file creation, this does not satisfy the requirement
2855 in the ISO C standard that the file is “created with exclusive access to the extent that the
2856 underlying system supports exclusive access”. (Having *fopen()* create the file and set a lock
2857 on the file as two separate operations is not the same, and it would introduce a race
2858 condition whereby another process could open the file and write to it (or set a lock) in
2859 between the two operations.) However, on all implementations that support mandatory file
2860 locking, its use is discouraged; therefore, it is recommended that implementations which
2861 support mandatory file locking do **not** add a means of creating a file with a whole-file
2862 exclusive lock set, so that *fopen()* is not required to enable mandatory file locking in order to
2863 conform to the ISO C standard. Note also that, since mandatory file locking is enabled via a
2864 file permissions change, the requirement that the '*x*' modifier does not alter the permissions
2865 means that this standard does not allow mandatory file locking to be enabled. An
2866 implementation that has a means of creating a file with a whole-file exclusive lock set would
2867 need to provide a way to change the behavior of *fopen()* depending on whether the calling
2868 process is executing in a POSIX.1 conforming environment or an ISO C conforming

2869 environment.

2870 [Note to reviewers: The above change may need to be updated depending on whether WG14 clarify](#)
2871 [the “exclusive access” requirement.](#)

2872 Ref 7.22.3.3 para 2

2873 On page 933 line 31673 section `free()`, change:

2874 Otherwise, if the argument does not match a pointer earlier returned by a function in
2875 POSIX.1-2017 that allocates memory as if by `malloc()`, or if the space has been deallocated
2876 by a call to `free()` or `realloc()`, the behavior is undefined.

2877 to:

2878 Otherwise, if the argument does not match a pointer earlier returned by `aligned_alloc()`,
2879 `calloc()`, `malloc()`, `[ADV]posix_memalign()`, `[/ADV] realloc()`, or a function in POSIX.1-
2880 20xx that allocates memory as if by `malloc()`, or if the space has been deallocated by a call
2881 to `free()` or `realloc()`, the behavior is undefined.

2882 Ref 7.22.3 para 2

2883 On page 933 line 31677 section `free()`, add a new paragraph:

2884 For purposes of determining the existence of a data race, `free()` shall behave as though it
2885 accessed only memory locations accessible through its argument and not other static
2886 duration storage. The function may, however, visibly modify the storage that it deallocates.
2887 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `[ADV]posix_memalign()`, `[/ADV]` and
2888 `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total
2889 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
2890 next allocation (if any) in this order.

2891 Ref 7.22.3.1

2892 On page 933 line 31691 section `free()`, add `aligned_alloc` to the SEE ALSO section.

2893 Ref 7.21.5.3 para 5

2894 On page 942 line 31988 section `freopen()`, change:

2895 [CX]The functionality described on this reference page is aligned with the ISO C standard.
2896 Any conflict between the requirements described here and the ISO C standard is
2897 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2898 to:

2899 [CX]Except for the “exclusive access” requirement (see [xref to `fopen()`]), the functionality
2900 described on this reference page is aligned with the ISO C standard. Any other conflict
2901 between the requirements described here and the ISO C standard is unintentional. This
2902 volume of POSIX.1-202x defers to the ISO C standard for all `freopen()` functionality except
2903 in relation to “exclusive access”.[/CX]

2904 Ref 7.21.5.3 para 3,5; 7.21.5.4 para 2

2905 On page 942 line 32010 section `freopen()`, replace the following text:

2906 shall be allocated and opened as if by a call to `open()` with the following flags:

2907 and the table that follows it, and the paragraph added by bug 411 after the table, with:

2908 shall be allocated and opened as if by a call to *open()* with the flags specified for *fopen()*
2909 with the same *mode* argument.

2910 Ref (none)
2911 On page 944 line 32094 section *freopen()*, change:

2912 It is possible that these side-effects are an unintended consequence of the way the feature is
2913 specified in the ISO/IEC 9899: 1999 standard, but unless or until the ISO C standard is
2914 changed, ...

2915 to:

2916 It is possible that these side-effects are an unintended consequence of the way the feature
2917 was specified in the ISO/IEC 9899: 1999 standard (and still is in the current standard), but
2918 unless or until the ISO C standard is changed, ...

2919 [Note to reviewers: if the APPLICATION USAGE and RATIONALE additions for *fopen\(\)* are
2920 retained, changes should be added here to make the equivalent sections for *freopen\(\)* refer to those
2921 for *fopen\(\)*.](#)

2922 Ref (none)
2923 On page 944 line 32102 section *freopen()*, after applying bug 411 change:

2924 The *x* mode suffix character was added by C1x only for files opened with a *mode* string
2925 beginning with *w*.

2926 to:

2927 The *x* mode suffix character is specified by the ISO C standard only for files opened with a
2928 mode string beginning with *w*.

2929 Ref 7.12.6.4 para 3
2930 On page 947 line 32161 section *frexp()*, change:

2931 The integer exponent shall be stored in the **int** object pointed to by *exp*.

2932 to:

2933 The integer exponent shall be stored in the **int** object pointed to by *exp*; if the integer
2934 exponent is outside the range of **int**, the results are unspecified.

2935 Ref F.10.3.4 para 3
2936 On page 947 line 32164 section *frexp()*, add a new paragraph:

2937 [MX]When the radix of the argument is a power of 2, the returned value shall be exact and
2938 shall be independent of the current rounding direction mode.[/MX]

2939 Ref 7.21.6.2 para 4
2940 On page 950 line 32239 section *fscanf()*, change:

2941 If a directive fails, as detailed below, the function shall return.

2942 to:

2943 When all directives have been executed, or if a directive fails (as detailed below), the
2944 function shall return.

2945 Ref 7.21.6.2 para 5
2946 On page 950 line 32242 section fscanf(), after applying bug 1163 change:

2947 A directive composed of one or more white-space bytes shall be executed by reading input
2948 until no more valid input can be read, or up to the first non-white-space byte , which remains
2949 unread.

2950 to:

2951 A directive composed of one or more white-space bytes shall be executed by reading input
2952 up to the first non-white-space byte, which shall remain unread, or until no more bytes can
2953 be read. The directive shall never fail.

2954 Ref (none)
2955 On page 955 line 32471 section fscanf(), change:

2956 This function is aligned with the ISO/IEC 9899: 1999 standard, and in doing so a few
2957 “obvious” things were not included. Specifically, the set of characters allowed in a scanset is
2958 limited to single-byte characters. In other similar places, multi-byte characters have been
2959 permitted, but for alignment with the ISO/IEC 9899: 1999 standard, it has not been done
2960 here.

2961 to:

2962 The set of characters allowed in a scanset is limited to single-byte characters. In other
2963 similar places, multi-byte characters have been permitted, but for alignment with the ISO C
2964 standard, it has not been done here.

2965 Ref 7.29.2.2 para 4
2966 On page 1004 line 34144 section fwscanf(), change:

2967 If a directive fails, as detailed below, the function shall return.

2968 to:

2969 When all directives have been executed, or if a directive fails (as detailed below), the
2970 function shall return.

2971 Ref 7.29.2.2 para 5
2972 On page 1004 line 34147 section fwscanf(), change:

2973 A directive composed of one or more white-space wide characters is executed by reading
2974 input until no more valid input can be read, or up to the first wide character which is not a
2975 white-space wide character, which remains unread.

2976 to:

2977 A directive composed of one or more white-space wide characters shall be executed by
2978 reading input up to the first wide character that is not a white-space wide character, which
2979 shall remain unread, or until no more wide characters can be read. The directive shall never
2980 fail.

2981 Ref 7.27.3, 7.1.4 para 5

2982 On page 1113 line 37680 section `gmtime()`, change:

2983 [CX]The `gmtime()` function need not be thread-safe.[/CX]

2984 to:

2985 The `gmtime()` function need not be thread-safe; however, `gmtime()` shall avoid data races
2986 with all functions other than itself, `asctime()`, `ctime()` and `localtime()`.

2987 Ref F.10.3.5 para 1

2988 On page 1133 line 38281 section `ilogb()`, add a new paragraph:

2989 [MX]When the correct result is representable in the range of the return type, the returned
2990 value shall be exact and shall be independent of the current rounding direction mode.[/MX]

2991 Ref F.10.3.5 para 3

2992 On page 1133 line 38282,38285,38288 section `ilogb()`, change:

2993 [XSI]On XSI-conformant systems, a domain error shall occur[/XSI]

2994 to:

2995 [XSI|MX]On XSI-conformant systems and on systems that support the IEC 60559 Floating-
2996 Point option, a domain error shall occur[/XSI|MX]

2997 Ref 7.12.6.5 para 2

2998 On page 1133 line 38291 section `ilogb()`, change:

2999 If the correct value is greater than `{INT_MAX}`, [MX]a domain error shall occur and[/MX]
3000 an unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error
3001 shall occur and `{INT_MAX}` shall be returned.[/XSI]

3002 If the correct value is less than `{INT_MIN}`, [MX]a domain error shall occur and[/MX] an
3003 unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error shall
3004 occur and `{INT_MIN}` shall be returned.[/XSI]

3005 to:

3006 If the correct value is greater than `{INT_MAX}` or less than `{INT_MIN}`, an unspecified
3007 value shall be returned. [XSI]On XSI-conformant systems, a domain error shall occur and
3008 `{INT_MAX}` or `{INT_MIN}`, respectively, shall be returned;[/XSI] [MX]if the IEC 60559
3009 Floating-Point option is supported, a domain error shall occur;[/MX] otherwise, a domain
3010 error or range error may occur.

3011 Ref F.10.3.5 para 3
3012 On page 1133 line 38300 section `ilogb()`, change:

3013 [XSI]The *x* argument is zero, NaN, or $\pm\text{Inf}$.[/XSI]

3014 to:

3015 [XSI|MX]The *x* argument is zero, NaN, or $\pm\text{Inf}$.[/XSI|MX]

3016 Ref F.10.11 para 1

3017 On page 1174 line 39604 section `isgreater()`,
3018 and page 1175 line 39642 section `isgreaterequal()`,
3019 and page 1177 line 39708 section `isless()`,
3020 and page 1178 line 39746 section `islessequal()`,
3021 and page 1179 line 39784 section `islessgreater()`, add a new paragraph:

3022 [MX]Relational operators and their corresponding comparison macros shall produce
3023 equivalent result values, even if argument values are represented in wider formats. Thus,
3024 comparison macro arguments represented in formats wider than their semantic types shall
3025 not be converted to the semantic types, unless the wide evaluation method converts operands
3026 of relational operators to their semantic types. The standard wide evaluation methods
3027 characterized by `FLT_EVAL_METHOD` equal to 1 or 2 (see [xref to <float.h>]) do not
3028 convert operands of relational operators to their semantic types. [/MX]

3029 (The editors may wish to merge the pages for the above interfaces to reduce duplication – they have
3030 duplicate APPLICATION USAGE as well.)

3031 Ref 7.30.2.2.1 para 4

3032 On page 1202 line 40411 section `iswctype()`, remove the CX shading from:

3033 If *charclass* is `(wctype_t)0`, these functions shall return 0.

3034 Ref 7.17.3.1

3035 On page 1229 line 41126 insert a new `kill_dependency()` section:

3036 **NAME**

3037 `kill_dependency` — terminate a dependency chain

3038 **SYNOPSIS**

3039 `#include <stdatomic.h>`
3040 `type kill_dependency(type y);`

3041 **DESCRIPTION**

3042 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3043 Any conflict between the requirements described here and the ISO C standard is
3044 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard. [/CX]

3045 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
3046 `<stdatomic.h>` header nor support this macro.

3047 The `kill_dependency()` macro shall terminate a dependency chain (see [xref to XBD 4.12.1
3048 Memory Ordering]). The argument shall not carry a dependency to the return value.

3049 **RETURN VALUE**

3050 The *kill_dependency()* macro shall return the value of *y*.

3051 **ERRORS**

3052 No errors are defined.

3053 **EXAMPLES**

3054 None.

3055 **APPLICATION USAGE**

3056 None.

3057 **RATIONALE**

3058 None.

3059 **FUTURE DIRECTIONS**

3060 None.

3061 **SEE ALSO**

3062 XBD Section 4.12.1, <**stdatomic.h**>

3063 **CHANGE HISTORY**

3064 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3065 Ref 7.12.8.3, 7.1.4 para 5

3066 On page 1241 line 41433 section *lgamma()*, change:

3067 [CX]These functions need not be thread-safe.[/CX]

3068 to:

3069 [XSI]If concurrent calls are made to these functions, the value of *signgam* is indeterminate.[/
3070 XSI]

3071 Ref 7.12.8.3, 7.1.4 para 5

3072 On page 1242 line 41464 section *lgamma()*, add a new paragraph to APPLICATION USAGE:

3073 If the value of *signgam* will be obtained after a call to *lgamma()*, *lgammaf()*, or *lgammal()*,
3074 in order to ensure that the value will not be altered by another call in a different thread,
3075 applications should either restrict calls to these functions to be from a single thread or use a
3076 lock such as a mutex or spin lock to protect a critical section starting before the function call
3077 and ending after the value of *signgam* has been obtained.

3078 Ref 7.12.8.3, 7.1.4 para 5

3079 On page 1242 line 41466 section *lgamma()*, change RATIONALE from:

3080 None.

3081 to:

3082 Earlier versions of this standard did not require *lgamma()*, *lgammaf()*, and *lgammal()* to be
3083 thread-safe because *signgam* was a global variable. They are now required to be thread-safe
3084 to align with the ISO C standard (which, since the introduction of threads in 2011, requires
3085 that they avoid data races), with the exception that they need not avoid data races when
3086 storing a value in the *signgam* variable. Since *signgam* is not specified by the ISO C
3087 standard, this exception is not a conflict with that standard.

3088 Ref 7.11.2.1, 7.1.4 para 5
3089 On page 1262 line 42124 section *localeconv()*, change:

3090 [CX]The *localeconv()* function need not be thread-safe.[/CX]

3091 to:

3092 The *localeconv()* function need not be thread-safe; however, *localeconv()* shall avoid data
3093 races with all other functions.

3094 Ref 7.27.3, 7.1.4 para 5
3095 On page 1265 line 42217 section *localtime()*, change:

3096 [CX]The *localtime()* function need not be thread-safe.[/CX]

3097 to:

3098 The *localtime()* function need not be thread-safe; however, *localtime()* shall avoid data races
3099 with all functions other than itself, *asctime()*, *ctime()* and *gmtime()*.

3100 Ref F.10.3.11 para 2
3101 On page 1280 line 42723 section *logb()*, add a new paragraph:

3102 [MX]The returned value shall be exact and shall be independent of the current rounding
3103 direction mode.[/MX]

3104 Ref 7.13.2.1 para 1
3105 On page 1283 line 42780 section *longjmp()*, change:

3106 `void longjmp(jmp_buf env, int val);`

3107 to:

3108 `_Noreturn void longjmp(jmp_buf env, int val);`

3109 Ref 7.13.2.1 para 2
3110 On page 1283 line 42804 section *longjmp()*, remove the CX shading from:

3111 The effect of a call to *longjmp()* where initialization of the **jmp_buf** structure was not
3112 performed in the calling thread is undefined.

3113 Ref 7.13.2.1 para 4
3114 On page 1283 line 42807 section *longjmp()*, change:

3115 After *longjmp()* is completed, program execution continues ...

3116 to:

3117 After *longjmp()* is completed, thread execution shall continue ...

3118 Ref 7.22.3 para 1

3119 On page 1295 line 43144 section *malloc()*, change:

3120 a pointer to any type of object

3121 to:

3122 a pointer to any type of object with a fundamental alignment requirement

3123 Ref 7.22.3 para 1

3124 On page 1295 line 43148 section *malloc()*, change:

3125 either a null pointer shall be returned, or ...

3126 to:

3127 either a null pointer shall be returned to indicate an error, or ...

3128 Ref 7.22.3 para 2

3129 On page 1295 line 43150 section *malloc()*, add a new paragraph:

3130 For purposes of determining the existence of a data race, *malloc()* shall behave as though it
3131 accessed only memory locations accessible through its argument and not other static
3132 duration storage. The function may, however, visibly modify the storage that it allocates.
3133 Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] and
3134 *realloc()* that allocate or deallocate a particular region of memory shall occur in a single total
3135 order (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the
3136 next allocation (if any) in this order.

3137 Ref 7.22.3.1

3138 On page 1295 line 43171 section *malloc()*, add *aligned_alloc* to the SEE ALSO section.

3139 Ref 7.22.7.1 para 2

3140 On page 1297 line 43194 section *mblen()*, change:

3141 `mbtowc((wchar_t *)0, s, n);`

3142 to:

3143 `mbtowc((wchar_t *)0, (const char *)0, 0);`

3144 `mbtowc((wchar_t *)0, s, n);`

3145 Ref 7.22.7 para 1

3146 On page 1297 line 43198 section *mblen()*, change:

3147 this function shall be placed into its initial state by a call for which

3148 to:

3149 this function shall be placed into its initial state at program startup and can be returned to
3150 that state by a call for which

3151 Ref 7.22.7 para 1, 7.1.4 para 5
3152 On page 1297 line 43206 section `mblen()`, change:

3153 [CX]The `mblen()` function need not be thread-safe.[/CX]

3154 to:

3155 The `mblen()` function need not be thread-safe; however, it shall avoid data races with all
3156 other functions.

3157 Ref 7.29.6.3 para 1, 7.1.4 para 5
3158 On page 1299 line 43254 section `mbrlen()`, change:

3159 [CX]The `mbrlen()` function need not be thread-safe if called with a NULL `ps`
3160 argument.[/CX]

3161 to:

3162 If called with a null `ps` argument, the `mbrlen()` function need not be thread-safe; however,
3163 such calls shall avoid data races with calls to `mbrlen()` with a non-null argument and with
3164 calls to all other functions.

3165 Ref 7.28.1, 7.1.4 para 5
3166 On page 1301 line 43296 insert a new `mbrtoc16()` section:

3167 **NAME**

3168 `mbrtoc16`, `mbrtoc32` — convert a character to a Unicode character code (restartable)

3169 **SYNOPSIS**

3170 `#include <uchar.h>`

3171 `size_t mbrtoc16(char16_t *restrict pc16, const char *restrict s,`
3172 `size_t n, mbstate_t *restrict ps);`

3173 `size_t mbrtoc32(char32_t *restrict pc32, const char *restrict s,`
3174 `size_t n, mbstate_t *restrict ps);`

3175 **DESCRIPTION**

3176 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3177 Any conflict between the requirements described here and the ISO C standard is
3178 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3179 If `s` is a null pointer, the `mbrtoc16()` function shall be equivalent to the call:

3180 `mbrtoc16(NULL, "", 1, ps)`

3181 In this case, the values of the parameters `pc16` and `n` are ignored.

3182 If `s` is not a null pointer, the `mbrtoc16()` function shall inspect at most `n` bytes beginning with
3183 the byte pointed to by `s` to determine the number of bytes needed to complete the next
3184 character (including any shift sequences). If the function determines that the next character

3185 is complete and valid, it shall determine the values of the corresponding wide characters and
3186 then, if *pc16* is not a null pointer, shall store the value of the first (or only) such character in
3187 the object pointed to by *pc16*. Subsequent calls shall store successive wide characters
3188 without consuming any additional input until all the characters have been stored. If the
3189 corresponding wide character is the null wide character, the resulting state described shall be
3190 the initial conversion state.

3191 If *ps* is a null pointer, the *mbrtoc16()* function shall use its own internal **mbstate_t** object,
3192 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
3193 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current
3194 conversion state of the associated character sequence.

3195 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

3196 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

3197 The *mbrtoc32()* function shall behave the same way as *mbrtoc16()* except that the first
3198 parameter shall point to an object of type **char32_t** instead of **char16_t**. References to *pc16*
3199 in the above description shall apply as if they were *pc32* when they are being read as
3200 describing *mbrtoc32()*.

3201 If called with a null *ps* argument, the *mbrtoc16()* function need not be thread-safe; however,
3202 such calls shall avoid data races with calls to *mbrtoc16()* with a non-null argument and with
3203 calls to all other functions.

3204 If called with a null *ps* argument, the *mbrtoc32()* function need not be thread-safe; however,
3205 such calls shall avoid data races with calls to *mbrtoc32()* with a non-null argument and with
3206 calls to all other functions.

3207 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
3208 calls *mbrtoc16()* or *mbrtoc32()* with a null pointer for *ps*.

3209 RETURN VALUE

3210 These functions shall return the first of the following that applies:

3211 0 If the next *n* or fewer bytes complete the character that corresponds to the null
3212 wide character (which is the value stored).

3213 between 1 and *n* inclusive

3214 If the next *n* or fewer bytes complete a valid character (which is the value
3215 stored); the value returned shall be the number of bytes that complete the
3216 character.

3217 (**size_t**)−3 If the next character resulting from a previous call has been stored, in which
3218 case no bytes from the input shall be consumed by the call.

3219 (**size_t**)−2 If the next *n* bytes contribute to an incomplete but potentially valid character,
3220 and all *n* bytes have been processed (no value is stored). When *n* has at least
3221 the value of the {**MB_CUR_MAX**} macro, this case can only occur if *s*
3222 points at a sequence of redundant shift sequences (for implementations with
3223 state-dependent encodings).

3224 (**size_t**)−1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
3225 contribute to a complete and valid character (no value is stored). In this case,

3226 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

3227 **ERRORS**

3228 These function shall fail if:

3229 [EILSEQ] An invalid character sequence is detected. [CX]In the POSIX locale
3230 an [EILSEQ] error cannot occur since all byte values are valid
3231 characters.[/CX]

3232 These functions may fail if:

3233 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

3234 **EXAMPLES**

3235 None.

3236 **APPLICATION USAGE**

3237 None.

3238 **RATIONALE**

3239 None.

3240 **FUTURE DIRECTIONS**

3241 None.

3242 **SEE ALSO**

3243 *c16rtomb*

3244 XBD <**uchar.h**>

3245 **CHANGE HISTORY**

3246 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3247 Ref 7.29.6.3 para 1, 7.1.4 para 5

3248 On page 1301 line 43322 section *mbrtowc()*, change:

3249 [CX]The *mbrtowc()* function need not be thread-safe if called with a NULL *ps*
3250 argument.[/CX]

3251 to:

3252 If called with a null *ps* argument, the *mbrtowc()* function need not be thread-safe; however,
3253 such calls shall avoid data races with calls to *mbrtowc()* with a non-null argument and with
3254 calls to all other functions.

3255 Ref 7.29.6.4 para 1, 7.1.4 para 5

3256 On page 1304 line 43451 section *mbsrtowcs()*, change:

3257 [CX]The *mbsnrtowcs()* and *mbsrtowcs()* functions need not be thread-safe if called with a
3258 NULL *ps* argument.[/CX]

3259 to:

3260 [CX]If called with a null *ps* argument, the *mbsnrrowcs()* function need not be thread-safe;
3261 however, such calls shall avoid data races with calls to *mbsnrrowcs()* with a non-null
3262 argument and with calls to all other functions.[/CX]

3263 If called with a null *ps* argument, the *mbsrtowcs()* function need not be thread-safe;
3264 however, such calls shall avoid data races with calls to *mbsrtowcs()* with a non-null
3265 argument and with calls to all other functions.

3266 Ref 7.22.7 para 1

3267 On page 1308 line 43557 section *mbtowc()*, change:

3268 this function is placed into its initial state by a call for which

3269 to:

3270 this function shall be placed into its initial state at program startup and can be returned to
3271 that state by a call for which

3272 Ref 7.22.7 para 1, 7.1.4 para 5

3273 On page 1308 line 43567 section *mbtowc()*, change:

3274 [CX]The *mbtowc()* function need not be thread-safe.[/CX]

3275 to:

3276 The *mbtowc()* function need not be thread-safe; however, it shall avoid data races with all
3277 other functions.

3278 Ref 7.24.5.1 para 2

3279 On page 1311 line 43642 section *memchr()*, change:

3280 Implementations shall behave as if they read the memory byte by byte from the beginning of
3281 the bytes pointed to by *s* and stop at the first occurrence of *c* (if it is found in the initial *n*
3282 bytes).

3283 to:

3284 The implementation shall behave as if it reads the bytes sequentially and stops as soon as a
3285 matching byte is found.

3286 Ref F.10.3.12 para 2

3287 On page 1346 line 44854 section *modf()*, add a new paragraph:

3288 [MX]The returned value shall be exact and shall be independent of the current rounding
3289 direction mode.[/MX]

3290 Ref 7.26.4

3291 On page 1384 line 46032 insert the following new *mtx_*()* sections:

3292 **NAME**

3293 `mtx_destroy`, `mtx_init` — destroy and initialize a mutex

3294 **SYNOPSIS**

3295 `#include <threads.h>`

3296 `void mtx_destroy(mtx_t *mtx);`
3297 `int mtx_init(mtx_t *mtx, int type);`

3298 **DESCRIPTION**

3299 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3300 Any conflict between the requirements described here and the ISO C standard is
3301 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3302 The `mtx_destroy()` function shall release any resources used by the mutex pointed to by `mtx`.
3303 A destroyed mutex object can be reinitialized using `mtx_init()`; the results of otherwise
3304 referencing the object after it has been destroyed are undefined. It shall be safe to destroy an
3305 initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that
3306 another thread is attempting to lock, or a mutex that is being used in a `cond_timedwait()` or
3307 `cond_wait()` call by another thread, results in undefined behavior. The behavior is undefined if
3308 the value specified by the `mtx` argument to `mtx_destroy()` does not refer to an initialized
3309 mutex.

3310 The `mtx_init()` function shall initialize a mutex object with properties indicated by `type`,
3311 whose valid values include:

3312 `mtx_plain` for a simple non-recursive mutex,

3313 `mtx_timed` for a non-recursive mutex that supports timeout,

3314 `mtx_plain | mtx_recursive` for a simple recursive mutex, or

3315 `mtx_timed | mtx_recursive` for a recursive mutex that supports timeout.

3316 If the `mtx_init()` function succeeds, it shall set the mutex pointed to by `mtx` to a value that
3317 uniquely identifies the newly initialized mutex. Upon successful initialization, the state of
3318 the mutex becomes initialized and unlocked. Attempting to initialize an already initialized
3319 mutex results in undefined behavior.

3320 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
3321 further requirements.

3322 These functions shall not be affected if the calling thread executes a signal handler during
3323 the call.[/CX]

3324 **RETURN VALUE**

3325 The `mtx_destroy()` function shall not return a value.

3326 The `mtx_init()` function shall return `thrd_success` on success or `thrd_error` if the
3327 request could not be honored.

3328 **ERRORS**

3329 No errors are defined.

3330 **EXAMPLES**

3331 None.

3332 **APPLICATION USAGE**

3333 A mutex can be destroyed immediately after it is unlocked. However, since attempting to
3334 destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that
3335 is being used in a *cond_timedwait()* or *cond_wait()* call by another thread results in undefined
3336 behavior, care must be taken to ensure that no other thread may be referencing the mutex.

3337 **RATIONALE**

3338 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3339 B.2.3].

3340 **FUTURE DIRECTIONS**

3341 None.

3342 **SEE ALSO**

3343 *mtx_lock*

3344 XBD <**threads.h**>

3345 **CHANGE HISTORY**

3346 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3347 **NAME**

3348 *mtx_lock*, *mtx_timedlock*, *mtx_trylock*, *mtx_unlock* — lock and unlock a mutex

3349 **SYNOPSIS**

3350 `#include <threads.h>`

```
3351 int mtx_lock(mtx_t *mtx);
3352 int mtx_timedlock(mtx_t * restrict mtx,
3353                  const struct timespec * restrict ts);
3354 int mtx_trylock(mtx_t *mtx);
3355 int mtx_unlock(mtx_t *mtx);
```

3356 **DESCRIPTION**

3357 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3358 Any conflict between the requirements described here and the ISO C standard is
3359 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3360 The *mtx_lock()* function shall block until it locks the mutex pointed to by *mtx*. If the mutex
3361 is non-recursive, the application shall ensure that it is not already locked by the calling
3362 thread.

3363 The *mtx_timedlock()* function shall block until it locks the mutex pointed to by *mtx* or until
3364 after the *TIME_UTC* -based calendar time pointed to by *ts*. The application shall ensure that
3365 the specified mutex supports timeout. [CX]Under no circumstance shall the function fail
3366 with a timeout if the mutex can be locked immediately. The validity of the *ts* parameter need
3367 not be checked if the mutex can be locked immediately.[/CX]

3368 The *mtx_trylock()* function shall endeavor to lock the mutex pointed to by *mtx*. If the mutex
3369 is already locked (by any thread, including the current thread), the function shall return
3370 without blocking. If the mutex is recursive and the mutex is currently owned by the calling
3371 thread, the mutex lock count (see below) shall be incremented by one and the *mtx_trylock()*
3372 function shall immediately return success.

3373 [CX]These functions shall not be affected if the calling thread executes a signal handler
3374 during the call; if a signal is delivered to a thread waiting for a mutex, upon return from the
3375 signal handler the thread shall resume waiting for the mutex as if it was not
3376 interrupted.[/CX]

3377 If a call to *mtx_lock()*, *mtx_timedlock()* or *mtx_trylock()* locks the mutex, prior calls to
3378 *mtx_unlock()* on the same mutex shall synchronize with this lock operation.

3379 The *mtx_unlock()* function shall unlock the mutex pointed to by *mtx* . The application shall
3380 ensure that the mutex pointed to by *mtx* is locked by the calling thread. [CX]If there are
3381 threads blocked on the mutex object referenced by *mtx* when *mtx_unlock()* is called,
3382 resulting in the mutex becoming available, the scheduling policy shall determine which
3383 thread shall acquire the mutex.[/CX]

3384 A recursive mutex shall maintain the concept of a lock count. When a thread successfully
3385 acquires a mutex for the first time, the lock count shall be set to one. Every time a thread
3386 relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks
3387 the mutex, the lock count shall be decremented by one. When the lock count reaches zero,
3388 the mutex shall become available for other threads to acquire.

3389 For purposes of determining the existence of a data race, mutex lock and unlock operations
3390 on mutexes of type **mtx_t** behave as atomic operations. All lock and unlock operations on a
3391 particular mutex occur in some particular total order.

3392 If *mtx* does not refer to an initialized mutex object, the behavior of these functions is
3393 undefined.

3394 RETURN VALUE

3395 The *mtx_lock()* and *mtx_unlock()* functions shall return *thrd_success* on success, or
3396 *thrd_error* if the request could not be honored.

3397 The *mtx_timedlock()* function shall return *thrd_success* on success, or *thrd_timedout*
3398 if the time specified was reached without acquiring the requested resource, or *thrd_error*
3399 if the request could not be honored.

3400 The *mtx_trylock()* function shall return *thrd_success* on success, or *thrd_busy* if the
3401 resource requested is already in use, or *thrd_error* if the request could not be honored.
3402 The *mtx_trylock()* function can spuriously fail to lock an unused resource, in which case it
3403 shall return *thrd_busy*.

3404 ERRORS

3405 See RETURN VALUE.

3406 EXAMPLES

3407 None.

3408 **APPLICATION USAGE**

3409 None.

3410 **RATIONALE**

3411 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3412 B.2.3].

3413 Since **<pthread.h>** has no equivalent of the `mtx_timed` mutex property, if the **<threads.h>**
3414 interfaces are implemented as a thin wrapper around **<pthread.h>** interfaces (meaning
3415 `mtx_t` and `pthread_mutex_t` are the same type), all mutexes support timeout and
3416 `mtx_timedlock()` will not fail for a mutex that was not initialized with `mtx_timed`.
3417 Alternatively, implementations can use a less thin wrapper where `mtx_t` contains additional
3418 properties that are not held in `pthread_mutex_t` in order to be able to return a failure
3419 indication from `mtx_timedlock()` calls where the mutex was not initialized with
3420 `mtx_timed`.

3421 **FUTURE DIRECTIONS**

3422 None.

3423 **SEE ALSO**

3424 `mtx_destroy`, `timespec_get`

3425 XBD Section 4.12.2, **<threads.h>**

3426 **CHANGE HISTORY**

3427 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3428 Ref F.10.8.2 para 2

3429 On page 1388 line 46143 section `nan()`, add a new paragraph:

3430 [MX]The returned value shall be exact and shall be independent of the current rounding
3431 direction mode.[/MX]

3432 Ref F.10.8.3 para 2, F.10.8.4 para 2

3433 On page 1395 line 46388 section `nextafter()`, add a new paragraph:

3434 [MX]Even though underflow or overflow can occur, the returned value shall be independent
3435 of the current rounding direction mode.[/MX]

3436 Ref 7.22.3 para 2

3437 On page 1448 line 48069 section `posix_memalign()`, add a new (unshaded) paragraph:

3438 For purposes of determining the existence of a data race, `posix_memalign()` shall behave as
3439 though it accessed only memory locations accessible through its arguments and not other
3440 static duration storage. The function may, however, visibly modify the storage that it
3441 allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `posix_memalign()`, and `realloc()`
3442 that allocate or deallocate a particular region of memory shall occur in a single total order
3443 (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the next
3444 allocation (if any) in this order.

3445 Ref 7.22.3.1
3446 On page 1449 line 48107 section `posix_memalign()`, add *aligned_alloc* to the SEE ALSO section.

3447 Ref F.10.4.4 para 1
3448 On page 1548 line 50724 section `pow()`, change:

3449 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 , a pole error shall
3450 occur and `pow()`, `powf()`, and `powl()` shall return `±HUGE_VAL`, `±HUGE_VALF`, and
3451 `±HUGE_VALL`, respectively if y is an odd integer, or `HUGE_VAL`, `HUGE_VALF`, and
3452 `HUGE_VALL`, respectively if y is not an odd integer.

3453 to:

3454 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 :

- 3455 • if y is an odd integer, a pole error shall occur and `pow()`, `powf()`, and `powl()` shall
3456 return `±HUGE_VAL`, `±HUGE_VALF`, and `±HUGE_VALL`, respectively;
- 3457 • if y is finite and is not an odd integer, a pole error shall occur and `pow()`, `powf()`, and
3458 `powl()` shall return `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively;
- 3459 • if y is `-Inf`, a pole error may occur and `pow()`, `powf()`, and `powl()` shall return
3460 `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

3461 Ref 7.26
3462 On page 1603 line 52244 section `pthread_cancel()`, add a new paragraph:

3463 If *thread* refers to a thread that was created using `thrd_create()`, the behavior is undefined.

3464 Ref 7.26.5.6
3465 On page 1603 line 52277 section `pthread_cancel()`, add a new RATIONALE paragraph:

3466 Use of `pthread_cancel()` to cancel a thread that was created using `thrd_create()` is undefined
3467 because `thrd_join()` has no way to indicate a thread was cancelled. The standard developers
3468 considered adding a `thrd_cancelled` enumeration constant that `thrd_join()` would return in
3469 this case. However, this return would be unexpected in code that is written to conform to the
3470 ISO C standard, and it would also not solve the problem that threads which use only ISO C
3471 `<threads.h>` interfaces (such as ones created by third party libraries written to conform to
3472 the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not
3473 provide cancellation cleanup handlers.

3474 Ref 7.26.5.5
3475 On page 1639 line 53422 section `pthread_exit()`, change:

3476 `void pthread_exit(void *value_ptr);`

3477 to:

3478 `_Noreturn void pthread_exit(void *value_ptr);`

3479 Ref 7.26.6
3480 On page 1639 line 53427 section `pthread_exit()`, change:

3481 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3482 specific data, appropriate destructor functions shall be called in an unspecified order.

3483 to:

3484 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3485 specific data (whether associated with key type **tss_t** or **pthread_key_t**), appropriate
3486 destructor functions shall be called in an unspecified order.

3487 Ref 7.26.5.5

3488 On page 1639 line 53432 section `pthread_exit()`, change:

3489 An implicit call to `pthread_exit()` is made when a thread other than the thread in which
3490 `main()` was first invoked returns from the start routine that was used to create it.

3491 to:

3492 An implicit call to `pthread_exit()` is made when a thread that was not created using
3493 `thrd_create()`, and is not the thread in which `main()` was first invoked, returns from the start
3494 routine that was used to create it.

3495 Ref 7.26.5.5

3496 On page 1639 line 53451 section `pthread_exit()`, change APPLICATION USAGE from:

3497 None.

3498 to:

3499 Calls to `pthread_exit()` should not be made from threads created using `thrd_create()`, as their
3500 exit status has a different type (**int** instead of **void ***). If `pthread_exit()` is called from the
3501 initial thread and it is not the last thread to terminate, other threads should not try to obtain
3502 its exit status using `thrd_join()`.

3503 Ref 7.26.5.5

3504 On page 1639 line 53453 section `pthread_exit()`, change:

3505 The normal mechanism by which a thread terminates is to return from the routine that was
3506 specified in the `pthread_create()` call that started it.

3507 to:

3508 The normal mechanism by which a thread that was started using `pthread_create()` terminates
3509 is to return from the routine that was specified in the `pthread_create()` call that started it.

3510 Ref 7.26.5.5, 7.26.6

3511 On page 1640 line 53470 section `pthread_exit()`, add `pthread_key_create`, `thrd_create`, `thrd_exit` and
3512 `tss_create` to the SEE ALSO section.

3513 Ref 7.26.5.5

3514 On page 1649 line 53748 section `pthread_join()`, add a new paragraph:

3515 If *thread* refers to a thread that was created using *thrd_create()* and the thread terminates, or
3516 has already terminated, by returning from its start routine, the behavior of *pthread_join()* is
3517 undefined. If *thread* refers to a thread that terminates, or has already terminated, by calling
3518 *thrd_exit()*, the behavior of *pthread_join()* is undefined.

3519 Ref 7.26.5.5

3520 On page 1651 line 53819 section *pthread_join()*, add a new RATIONALE paragraph:

3521 The *pthread_join()* function cannot be used to obtain the exit status of a thread that was
3522 created using *thrd_create()* and which terminates by returning from its start routine, or of a
3523 thread that terminates by calling *thrd_exit()*, because such threads have an **int** exit status,
3524 instead of the **void *** that *pthread_join()* returns via its *value_ptr* argument.

3525 Ref 7.22.4.7

3526 On page 1765 line 57040 insert the following new *quick_exit()* section:

3527 **NAME**

3528 *quick_exit* — terminate a process

3529 **SYNOPSIS**

3530 `#include <stdlib.h>`

3531 `_Noreturn void quick_exit(int status);`

3532 **DESCRIPTION**

3533 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3534 Any conflict between the requirements described here and the ISO C standard is
3535 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3536 The *quick_exit()* function shall cause normal process termination to occur. It shall not call
3537 functions registered with *atexit()* nor any registered signal handlers. If a process calls the
3538 *quick_exit()* function more than once, or calls the *exit()* function in addition to the
3539 *quick_exit()* function, the behavior is undefined. If a signal is raised while the *quick_exit()*
3540 function is executing, the behavior is undefined.

3541 The *quick_exit()* function shall first call all functions registered by *at_quick_exit()*, in the
3542 reverse order of their registration, except that a function is called after any previously
3543 registered functions that had already been called at the time it was registered. If, during the
3544 call to any such function, a call to the *longjmp()* [CX] or *siglongjmp()*[/CX] function is made
3545 that would terminate the call to the registered function, the behavior is undefined.

3546 If a function registered by a call to *at_quick_exit()* fails to return, the remaining registered
3547 functions shall not be called and the rest of the *quick_exit()* processing shall not be
3548 completed.

3549 Finally, the *quick_exit()* function shall terminate the process as if by a call to *_Exit(status)*.

3550 **RETURN VALUE**

3551 The *quick_exit()* function does not return.

3552 **ERRORS**

3553 No errors are defined.

3554 **EXAMPLES**

3555 None.

3556 **APPLICATION USAGE**

3557 None.

3558 **RATIONALE**

3559 None.

3560 **FUTURE DIRECTIONS**

3561 None.

3562 **SEE ALSO**

3563 *_Exit, at_quick_exit, atexit, exit*

3564 XBD <stdlib.h>

3565 **CHANGE HISTORY**

3566 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3567 Ref 7.22.2.1 para 3, 7.1.4 para 5

3568 On page 1767 line 57095 section *rand()*, change:

3569 [CX]The *rand()* function need not be thread-safe.[/CX]

3570 to:

3571 The *rand()* function need not be thread-safe; however, *rand()* shall avoid data races with all
3572 functions other than non-thread-safe pseudo-random sequence generation functions.

3573 Ref 7.22.2.2 para 3, 7.1.4 para 5

3574 On page 1767 line 57105 section *rand()*, add a new paragraph:

3575 The *srand()* function need not be thread-safe; however, *srand()* shall avoid data races with
3576 all functions other than non-thread-safe pseudo-random sequence generation functions.

3577 Ref 7.22.3 para 1,2; 7.22.3.5 para 2,3,4; 7.31.12 para 2

3578 On page 1788 line 57862-57892 section *realloc()*, replace the DESCRIPTION and RETURN
3579 VALUE sections with:

3580 **DESCRIPTION**

3581 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3582 Any conflict between the requirements described here and the ISO C standard is
3583 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3584 The *realloc()* function shall deallocate the old object pointed to by *ptr* and return a pointer to
3585 a new object that has the size specified by *size*. The contents of the new object shall be the
3586 same as that of the old object prior to deallocation, up to the lesser of the new and old sizes.
3587 Any bytes in the new object beyond the size of the old object have indeterminate values.

3588 If *ptr* is a null pointer, *realloc()* shall be equivalent to *malloc()* function for the specified
3589 size. Otherwise, if *ptr* does not match a pointer returned earlier by *aligned_alloc()*, *calloc()*,
3590 *malloc()*, [ADV]*posix_memalign()*,[/ADV] *realloc()*, or a function in POSIX.1-20xx that
3591 allocates memory as if by *malloc()*, or if the space has been deallocated by a call to *free()* or
3592 *realloc()*, the behavior is undefined.

3593 If *size* is non-zero and memory for the new object is not allocated, the old object shall not be
3594 deallocated. [OB]If *size* is zero and memory for the new object is not allocated, it is
3595 implementation-defined whether the old object is deallocated; if the old object is not
3596 deallocated, its value shall be unchanged.[/OB]

3597 The order and contiguity of storage allocated by successive calls to *realloc()* is unspecified.
3598 The pointer returned if the allocation succeeds shall be suitably aligned so that it may be
3599 assigned to a pointer to any type of object with a fundamental alignment requirement and
3600 then used to access such an object in the space allocated (until the space is explicitly freed or
3601 reallocated). Each such allocation shall yield a pointer to an object disjoint from any other
3602 object. The pointer returned shall point to the start (lowest byte address) of the allocated
3603 space. If the space cannot be allocated, a null pointer shall be returned. [OB]If the size of the
3604 space requested is 0, the behavior is implementation-defined: either a null pointer shall be
3605 returned to indicate an error, or the behavior shall be as if the size were some non-zero
3606 value, except that the behavior is undefined if the returned pointer is used to access an
3607 object.[/OB]

3608 For purposes of determining the existence of a data race, *realloc()* shall behave as though it
3609 accessed only memory locations accessible through its arguments and not other static
3610 duration storage. The function may, however, visibly modify the storage that it allocates or
3611 deallocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*,
3612 [ADV]*posix_memalign()*,[/ADV] and *realloc()* that allocate or deallocate a particular region
3613 of memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
3614 deallocation call shall synchronize with the next allocation (if any) in this order.

3615 **RETURN VALUE**

3616 The *realloc()* function shall return a pointer to the new object (which can have the same
3617 value as a pointer to the old object), or a null pointer if the new object has not been
3618 allocated.

3619 [OB]If *size* is zero, either:

- 3620 • A null pointer shall be returned [CX]and, if *ptr* is not a null pointer, *errno* shall be set
3621 to an implementation-defined value.[/CX]
- 3622 • A pointer to the allocated space shall be returned, and the memory object pointed to
3623 by *ptr* shall be freed. The application shall ensure that the pointer is not used to
3624 access an object.[/OB]

3625 If there is not enough available memory, *realloc()* shall return a null pointer [CX]and set
3626 *errno* to [ENOMEM][/CX].

3627 Ref 7.22.3.5 para 3,4

3628 On page 1789 line 57899 section *realloc()*, change:

3629 The description of *realloc()* has been modified from previous versions of this standard to
3630 align with the ISO/IEC 9899: 1999 standard. Previous versions explicitly permitted a call to

3631 *realloc(p, 0) to free the space pointed to by p and return a null pointer. While this behavior*
3632 *could be interpreted as permitted by this version of the standard, the C language committee*
3633 *have indicated that this interpretation is incorrect. Applications should assume that if*
3634 *realloc() returns a null pointer, the space pointed to by p has not been freed. Since this could*
3635 *lead to double-frees, implementations should also set errno if a null pointer actually*
3636 *indicates a failure, and applications should only free the space if errno was changed.*

3637 to:

3638 The ISO C standard makes it implementation-defined whether a call to *realloc(p, 0)* frees the
3639 space pointed to by *p* if it returns a null pointer because memory for the new object was not
3640 allocated. POSIX.1 instead requires that implementations set *errno* if a null pointer is
3641 returned and the space has not been freed, and POSIX applications should only free the
3642 space if *errno* was changed.

3643 Ref 7.31.12 para 2

3644 On page 1789 line 57909-57912 section *realloc()*, change FUTURE DIRECTIONS to:

3645 The ISO C standard states that invoking *realloc()* with a *size* argument equal to zero is an
3646 obsolescent feature. This feature may be removed in a future version of this standard.

3647 Ref 7.22.3.1

3648 On page 1789 line 57914 section *realloc()*, add *aligned_alloc* to the SEE ALSO section.

3649 Ref F.10.7.2 para 2

3650 On page 1809 line 58638 section *remainder()*, add a new paragraph:

3651 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3652 Ref F.10.7.3 para 2

3653 On page 1814 line 58758 section *remquo()*, add a new paragraph:

3654 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3655 Ref F.10.6.6 para 3

3656 On page 1828 line 59258 section *round()*, add a new paragraph:

3657 [MX]These functions may raise the inexact floating-point exception for finite non-integer
3658 arguments.[/MX]

3659 Ref F.10.6.6 para 3

3660 On page 1828 line 59272 section *round()*, delete from APPLICATION USAGE:

3661 These functions may raise the inexact floating-point exception if the result differs in value
3662 from the argument.

3663 Ref F.10.3.13 para 2

3664 On page 1829 line 59306 section *scalbln()*, add a new paragraph:

3665 [MX]If the calculation does not overflow or underflow, the returned value shall be exact and
3666 shall be independent of the current rounding direction mode.[/MX]

3667 Ref 7.11.1.1 para 5
3668 On page 1903 line 61520 section `setlocale()`, change:

3669 [CX]The `setlocale()` function need not be thread-safe.[/CX]

3670 to:

3671 The `setlocale()` function need not be thread-safe; however, it shall avoid data races with all
3672 function calls that do not affect and are not affected by the global locale.

3673 Ref 7.13.2.1 para 1
3674 On page 1970 line 63497 section `siglongjmp()`, change:

3675 void `siglongjmp(sigjmp_buf env, int val)`;

3676 to:

3677 _Noreturn void `siglongjmp(sigjmp_buf env, int val)`;

3678 Ref 7.13.2.1 para 4
3679 On page 1970 line 63504 section `siglongjmp()`, change:

3680 After `siglongjmp()` is completed, program execution shall continue ...

3681 to:

3682 After `siglongjmp()` is completed, thread execution shall continue ...

3683 Ref 7.14.1.1 para 5
3684 On page 1971 line 63564 section `signal()`, change:

3685 with static storage duration

3686 to:

3687 with static or thread storage duration that is not a lock-free atomic object

3688 Ref 7.14.1.1 para 7
3689 On page 1972 line 63573 section `signal()`, add a new paragraph:

3690 [CX]The `signal()` function is required to be thread-safe. (See [xref to 2.9.1 Thread-Safety].)
3691 [/CX]

3692 Ref 7.14.1.1 para 7
3693 On page 1972 line 63591 section `signal()`, change RATIONALE from:

3694 None.

3695 to:

3696 The ISO C standard says that the use of `signal()` in a multi-threaded program results in
3697 undefined behavior. However, POSIX.1 has required `signal()` to be thread-safe since before

3698 threads were added to the ISO C standard.

3699 Ref F.10.4.5 para 1
3700 On page 2009 line 64624 section `sqrt()`, add:

3701 [MX]The returned value shall be dependent on the current rounding direction mode.[/MX]

3702 Ref 7.24.6.2 para 3, 7.1.4 para 5
3703 On page 2035 line 65231 section `strerror()`, change:

3704 [CX]The `strerror()` function need not be thread-safe.[/CX]

3705 to:

3706 The `strerror()` function need not be thread-safe; however, `strerror()` shall avoid data races
3707 with all other functions.

3708 Ref 7.22.1.3 para 10
3709 On page 2073 line 66514 section `strtod()`, change:

3710 If the correct value is outside the range of representable values

3711 to:

3712 If the correct value would cause an overflow and default rounding is in effect

3713 Ref 7.24.5.8 para 6, 7.1.4 para 5
3714 On page 2078 line 66674 section `strtok()`, change:

3715 [CX]The `strtok()` function need not be thread-safe.[/CX]

3716 to:

3717 The `strtok()` function need not be thread-safe; however, `strtok()` shall avoid data races with
3718 all other functions.

3719 Ref 7.22.4.8, 7.1.4 para 5
3720 On page 2107 line 67579 section `system()`, change:

3721 The `system()` function need not be thread-safe.

3722 to:

3723 [CX]If concurrent calls to `system()` are made from multiple threads, it is unspecified
3724 whether:

- 3725 • each call saves and restores the dispositions of the SIGINT and SIGQUIT signals
3726 independently, or
- 3727 • in a set of concurrent calls the dispositions in effect after the last call returns are
3728 those that were in effect on entry to the first call.

3729 If a thread is cancelled while it is in a call to `system()`, it is unspecified whether the child
3730 process is terminated and waited for, or is left running.[/CX]

3731 Ref 7.22.4.8, 7.1.4 para 5
3732 On page 2108 line 67627 section `system()`, change:

3733 Using the `system()` function in more than one thread in a process or when the SIGCHLD
3734 signal is being manipulated by more than one thread in a process may produce unexpected
3735 results.

3736 to:

3737 Although `system()` is required to be thread-safe, it is recommended that concurrent calls
3738 from multiple threads are avoided, since `system()` is not required to coordinate the saving
3739 and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of
3740 overlapping calls, and therefore the signals might end up being set to ignored after the last
3741 call returns. Applications should also avoid cancelling a thread while it is in a call to
3742 `system()` as the child process may be left running in that event. In addition, if another thread
3743 alters the disposition of the SIGCHLD signal, a call to `signal()` may produce unexpected
3744 results.

3745 Ref 7.22.4.8, 7.1.4 para 5
3746 On page 2109 line 67675 section `system()`, delete:

3747 `#include <signal.h>`

3748 Ref 7.22.4.8, 7.1.4 para 5
3749 On page 2109 line 67692,67696,67712 section `system()`, change `sigprocmask` to
3750 `pthread_sigmask`.

3751 Ref 7.22.4.8, 7.1.4 para 5
3752 On page 2110 line 67718 section `system()`, change:

3753 Note also that the above example implementation is not thread-safe. Implementations can
3754 provide a thread-safe `system()` function, but doing so involves complications such as how to
3755 restore the signal dispositions for SIGINT and SIGQUIT correctly if there are overlapping
3756 calls, and how to deal with cancellation. The example above would not restore the signal
3757 dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-
3758 safe implementation since canceling a non-thread-safe function results in undefined
3759 behavior (see Section 2.9.5.2, on page 518). To avoid leaking a process ID, a thread-safe
3760 implementation would need to terminate the child process when acting on a cancellation.

3761 to:

3762 Earlier versions of this standard did not require `system()` to be thread-safe because it alters
3763 the process-wide disposition of the SIGINT and SIGQUIT signals. It is now required to be
3764 thread-safe to align with the ISO C standard, which (since the introduction of threads in
3765 2011) requires that it avoids data races. However, the function is not required to coordinate
3766 the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set
3767 of overlapping calls, and the above example does not do so. The example also does not
3768 terminate and wait for the child process if the calling thread is cancelled, and so would leak
3769 a process ID in that event.

3770 Ref 7.26.5
3771 On page 2148 line 68796 insert the following new `thrd_*` sections:

3772 **NAME**

3773 `thrd_create` — thread creation

3774 **SYNOPSIS**

3775 `#include <threads.h>`

3776 `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`

3777 **DESCRIPTION**

3778 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3779 Any conflict between the requirements described here and the ISO C standard is
3780 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3781 The `thrd_create()` function shall create a new thread executing `func(arg)`. If the `thrd_create()`
3782 function succeeds, it shall set the object pointed to by `thr` to the identifier of the newly
3783 created thread. (A thread's identifier might be reused for a different thread once the original
3784 thread has exited and either been detached or joined to another thread.) The completion of
3785 the `thrd_create()` function shall synchronize with the beginning of the execution of the new
3786 thread.

3787 [CX]The signal state of the new thread shall be initialized as follows:

- 3788 • The signal mask shall be inherited from the creating thread.
- 3789 • The set of signals pending for the new thread shall be empty.

3790 The thread-local current locale shall not be inherited from the creating thread.

3791 The floating-point environment shall be inherited from the creating thread.[/CX]

3792 [XSI] The alternate stack shall not be inherited from the creating thread.[/XSI]

3793 Returning from `func` shall have the same behavior as invoking `thrd_exit()` with the value
3794 returned from `func`.

3795 If `thrd_create()` fails, no new thread shall be created and the contents of the location
3796 referenced by `thr` are undefined.

3797 [CX]The `thrd_create()` function shall not be affected if the calling thread executes a signal
3798 handler during the call.[/CX]

3799 **RETURN VALUE**

3800 The `thrd_create()` function shall return `thrd_success` on success; or `thrd_nomem` if no
3801 memory could be allocated for the thread requested; or `thrd_error` if the request could not
3802 be honored, [CX]such as if the system-imposed limit on the total number of threads in a
3803 process {`PTHREAD_THREADS_MAX`} would be exceeded.[/CX]

3804 **ERRORS**

3805 See RETURN VALUE.

3806 **EXAMPLES**

3807 None.

3808 **APPLICATION USAGE**

3809 There is no requirement on the implementation that the ID of the created thread be available
3810 before the newly created thread starts executing. The calling thread can obtain the ID of the
3811 created thread through the *thr* argument of the *thrd_create()* function, and the newly created
3812 thread can obtain its ID by a call to *thrd_current()*.

3813 **RATIONALE**

3814 The *thrd_create()* function is not affected by signal handlers for the reasons stated in [xref to
3815 XRAT B.2.3].

3816 **FUTURE DIRECTIONS**

3817 None.

3818 **SEE ALSO**

3819 *pthread_create*, *thrd_current*, *thrd_detach*, *thrd_exit*, *thrd_join*

3820 XBD Section 4.12.2, <**threads.h**>

3821 **CHANGE HISTORY**

3822 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3823 **NAME**

3824 *thrd_current* — get the calling thread ID

3825 **SYNOPSIS**

3826 #include <threads.h>

3827 thrd_t thrd_current(void);

3828 **DESCRIPTION**

3829 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3830 Any conflict between the requirements described here and the ISO C standard is
3831 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3832 The *thrd_current()* function shall identify the thread that called it.

3833 **RETURN VALUE**

3834 The *thrd_current()* function shall return the thread ID of the thread that called it.

3835 The *thrd_current()* function shall always be successful. No return value is reserved to
3836 indicate an error.

3837 **ERRORS**

3838 No errors are defined.

3839 **EXAMPLES**

3840 None.

3841 **APPLICATION USAGE**

3842 None.

3843 **RATIONALE**

3844 None.

3845 **FUTURE DIRECTIONS**

3846 None.

3847 **SEE ALSO**

3848 *pthread_self*, *thrd_create*, *thrd_equal*

3849 XBD Section 4.12.2, <**threads.h**>

3850 **CHANGE HISTORY**

3851 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3852 **NAME**

3853 *thrd_detach* — detach a thread

3854 **SYNOPSIS**

3855 `#include <threads.h>`

3856 `int thrd_detach(thrd_t thr);`

3857 **DESCRIPTION**

3858 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3859 Any conflict between the requirements described here and the ISO C standard is
3860 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3861 The *thrd_detach*() function shall change the thread *thr* from joinable to detached, indicating
3862 to the implementation that any resources allocated to the thread can be reclaimed when that
3863 thread terminates. The application shall ensure that the thread identified by *thr* has not been
3864 previously detached or joined with another thread.

3865 [CX]The *thrd_detach*() function shall not be affected if the calling thread executes a signal
3866 handler during the call.[/CX]

3867 **RETURN VALUE**

3868 The *thrd_detach*() function shall return *thrd_success* on success or *thrd_error* if the
3869 request could not be honored.

3870 **ERRORS**

3871 No errors are defined.

3872 **EXAMPLES**

3873 None.

3874 **APPLICATION USAGE**

3875 None.

3876 **RATIONALE**

3877 The *thrd_detach*() function is not affected by signal handlers for the reasons stated in [xref
3878 to XRAT B.2.3].

3879 **FUTURE DIRECTIONS**

3880 None.

3881 **SEE ALSO**

3882 *pthread_detach*, *thrd_create*, *thrd_join*

3883 XBD <**threads.h**>

3884 **CHANGE HISTORY**

3885 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3886 **NAME**

3887 *thrd_equal* — compare thread IDs

3888 **SYNOPSIS**

3889 `#include <threads.h>`

3890 `int thrd_equal(thrd_t thr0, thrd_t thr1);`

3891 **DESCRIPTION**

3892 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3893 Any conflict between the requirements described here and the ISO C standard is
3894 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3895 The *thrd_equal()* function shall determine whether the thread identified by *thr0* refers to the
3896 thread identified by *thr1*.

3897 [CX]The *thrd_equal()* function shall not be affected if the calling thread executes a signal
3898 handler during the call.[/CX]

3899 **RETURN VALUE**

3900 The *thrd_equal()* function shall return a non-zero value if *thr0* and *thr1* are equal; otherwise,
3901 zero shall be returned.

3902 If either *thr0* or *thr1* is not a valid thread ID [CX]and is not equal to `PTHREAD_NULL`
3903 (which is defined in <**pthread.h**>)[/CX], the behavior is undefined.

3904 **ERRORS**

3905 No errors are defined.

3906 **EXAMPLES**

3907 None.

3908 **APPLICATION USAGE**

3909 None.

3910 **RATIONALE**

3911 See the RATIONALE section for *pthread_equal()*.

3912 The *thrd_equal()* function is not affected by signal handlers for the reasons stated in [xref to
3913 XRAT B.2.3].

3914 **FUTURE DIRECTIONS**

3915 None.

3916 **SEE ALSO**

3917 *pthread_equal*, *thrd_current*

3918 XBD <pthread.h>, <threads.h>

3919 **CHANGE HISTORY**

3920 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3921 **NAME**

3922 *thrd_exit* — thread termination

3923 **SYNOPSIS**

3924 `#include <threads.h>`

3925 `_Noreturn void thrd_exit(int res);`

3926 **DESCRIPTION**

3927 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3928 Any conflict between the requirements described here and the ISO C standard is
3929 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3930 For every thread-specific storage key [CX](regardless of whether it has type **tss_t** or
3931 **pthread_key_t**)[/CX] which was created with a non-null destructor and for which the value
3932 is non-null, *thrd_exit()* shall set the value associated with the key to a null pointer value and
3933 then invoke the destructor with its previous value. The order in which destructors are
3934 invoked is unspecified.

3935 If after this process there remain keys with both non-null destructors and values, the
3936 implementation shall repeat this process up to [CX]
3937 {PTHREAD_DESTRUCTOR_ITERATIONS}[/CX] times.

3938 Following this, the *thrd_exit()* function shall terminate execution of the calling thread and
3939 shall set its exit status to *res*. [CX]Thread termination shall not release any application
3940 visible process resources, including, but not limited to, mutexes and file descriptors, nor
3941 shall it perform any process-level cleanup actions, including, but not limited to, calling any
3942 *atexit()* routines that might exist.[/CX]

3943 An implicit call to *thrd_exit()* is made when a thread that was created using *thrd_create()*
3944 returns from the start routine that was used to create it (see [xref to *thrd_create()*]).

3945 [CX]The behavior of *thrd_exit()* is undefined if called from a destructor function that was
3946 invoked as a result of either an implicit or explicit call to *thrd_exit()*.[/CX]

3947 The process shall exit with an exit status of zero after the last thread has been terminated.
3948 The behavior shall be as if the implementation called *exit()* with a zero argument at thread
3949 termination time.

3950 **RETURN VALUE**

3951 This function shall not return a value.

3952 **ERRORS**

3953 No errors are defined.

3954 **EXAMPLES**

3955 None.

3956 **APPLICATION USAGE**

3957 Calls to *thrd_exit()* should not be made from threads created using *pthread_create()* or via a
3958 SIGEV_THREAD notification, as their exit status has a different type (**void *** instead of
3959 **int**). If *thrd_exit()* is called from the initial thread and it is not the last thread to terminate,
3960 other threads should not try to obtain its exit status using *pthread_join()*.

3961 **RATIONALE**

3962 The normal mechanism by which a thread that was started using *thrd_create()* terminates is
3963 to return from the function that was specified in the *thrd_create()* call that started it. The
3964 *thrd_exit()* function provides the capability for such a thread to terminate without requiring a
3965 return from the start routine of that thread, thereby providing a function analogous to *exit()*.

3966 Regardless of the method of thread termination, the destructors for any existing thread-
3967 specific data are executed.

3968 **FUTURE DIRECTIONS**

3969 None.

3970 **SEE ALSO**

3971 *exit*, *pthread_create*, *thrd_join*

3972 XBD <**threads.h**>

3973 **CHANGE HISTORY**

3974 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3975 **NAME**

3976 *thrd_join* — wait for thread termination

3977 **SYNOPSIS**

3978 `#include <threads.h>`

3979 `int thrd_join(thrd_t thr, int *res);`

3980 **DESCRIPTION**

3981 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3982 Any conflict between the requirements described here and the ISO C standard is
3983 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3984 The *thrd_join()* function shall join the thread identified by *thr* with the current thread by
3985 blocking until the other thread has terminated. If the parameter *res* is not a null pointer,
3986 *thrd_join()* shall store the thread's exit status in the integer pointed to by *res*. The
3987 termination of the other thread shall synchronize with the completion of the *thrd_join()*
3988 function. The application shall ensure that the thread identified by *thr* has not been

3989 previously detached or joined with another thread.

3990 The results of multiple simultaneous calls to *thrd_join()* specifying the same target thread
3991 are undefined.

3992 The behavior is undefined if the value specified by the *thr* argument to *thrd_join()* refers to
3993 the calling thread.

3994 [CX]It is unspecified whether a thread that has exited but remains unjoined counts against
3995 {PTHREAD_THREADS_MAX}.

3996 If *thr* refers to a thread that was created using *pthread_create()* or via a SIGEV_THREAD
3997 notification and the thread terminates, or has already terminated, by returning from its start
3998 routine, the behavior of *thrd_join()* is undefined. If *thr* refers to a thread that terminates, or
3999 has already terminated, by calling *pthread_exit()* or by being cancelled, the behavior of
4000 *thrd_join()* is undefined.

4001 The *thrd_join()* function shall not be affected if the calling thread executes a signal handler
4002 during the call.[/CX]

4003 **RETURN VALUE**

4004 The *thrd_join()* function shall return *thrd_success* on success or *thrd_error* if the
4005 request could not be honored.

4006 [CX]It is implementation-defined whether *thrd_join()* detects deadlock situations; if it does
4007 detect them, it shall return *thrd_error* when one is detected.[/CX]

4008 **ERRORS**

4009 See RETURN VALUE.

4010 **EXAMPLES**

4011 None.

4012 **APPLICATION USAGE**

4013 None.

4014 **RATIONALE**

4015 The *thrd_join()* function provides a simple mechanism allowing an application to wait for a
4016 thread to terminate. After the thread terminates, the application may then choose to clean up
4017 resources that were used by the thread. For instance, after *thrd_join()* returns, any
4018 application-provided stack storage could be reclaimed.

4019 The *thrd_join()* or *thrd_detach()* function should eventually be called for every thread that is
4020 created using *thrd_create()* so that storage associated with the thread may be reclaimed.

4021 The *thrd_join()* function cannot be used to obtain the exit status of a thread that was created
4022 using *pthread_create()* or via a SIGEV_THREAD notification and which terminates by
4023 returning from its start routine, or of a thread that terminates by calling *pthread_exit()*,
4024 because such threads have a **void *** exit status, instead of the **int** that *thrd_join()* returns via
4025 its *res* argument.

4026 The *thrd_join()* function cannot be used to obtain the exit status of a thread that terminates
4027 by being cancelled because it has no way to indicate that a thread was cancelled. (The
4028 *pthread_join()* function does this by returning a reserved **void *** exit status; it is not possible

4029 to reserve an **int** value for this purpose without introducing a conflict with the ISO C
4030 standard.) The standard developers considered adding a `thrd_cancelled` enumeration
4031 constant that `thrd_join()` would return in this case. However, this return would be
4032 unexpected in code that is written to conform to the ISO C standard, and it would also not
4033 solve the problem that threads which use only ISO C `<threads.h>` interfaces (such as ones
4034 created by third party libraries written to conform to the ISO C standard) have no way to
4035 handle being cancelled, as the ISO C standard does not provide cancellation cleanup
4036 handlers.

4037 The `thrd_join()` function is not affected by signal handlers for the reasons stated in [xref to
4038 XRAT B.2.3].

4039 **FUTURE DIRECTIONS**

4040 None.

4041 **SEE ALSO**

4042 `pthread_create`, `pthread_exit`, `pthread_join`, `thrd_create`, `thrd_exit`

4043 XBD Section 4.12.2, `<threads.h>`

4044 **CHANGE HISTORY**

4045 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4046 **NAME**

4047 `thrd_sleep` — suspend execution for an interval

4048 **SYNOPSIS**

4049 `#include <threads.h>`

```
4050 int thrd_sleep(const struct timespec *duration,  
4051               struct timespec *remaining);
```

4052 **DESCRIPTION**

4053 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4054 Any conflict between the requirements described here and the ISO C standard is
4055 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4056 The `thrd_sleep()` function shall suspend execution of the calling thread until either the
4057 interval specified by `duration` has elapsed or a signal is delivered to the calling thread whose
4058 action is to invoke a signal-catching function or to terminate the process. If interrupted by a
4059 signal and the `remaining` argument is not null, the amount of time remaining (the requested
4060 interval minus the time actually slept) shall be stored in the interval it points to. The
4061 `duration` and `remaining` arguments can point to the same object.

4062 The suspension time may be longer than requested because the interval is rounded up to an
4063 integer multiple of the sleep resolution or because of the scheduling of other activity by the
4064 system. But, except for the case of being interrupted by a signal, the suspension time shall
4065 not be less than that specified, as measured by the system clock `TIME_UTC`.

4066 **RETURN VALUE**

4067 The `thrd_sleep()` function shall return zero if the requested time has elapsed, `-1` if it has
4068 been interrupted by a signal, or a negative value (which may also be `-1`) if it fails for any
4069 other reason. [CX]If it returns a negative value, it shall set `errno` to indicate the error.[/CX]

4070 **ERRORS**

4071 [CX]The *thrd_sleep()* function shall fail if:

4072 [EINTR]

4073 The *thrd_sleep()* function was interrupted by a signal.

4074 [EINVAL]

4075 The *duration* argument specified a nanosecond value less than zero or greater than or
4076 equal to 1000 million.[/CX]

4077 **EXAMPLES**

4078 None.

4079 **APPLICATION USAGE**

4080 Since the return value may be -1 for errors other than [EINTR], applications should examine
4081 *errno* to distinguish [EINTR] from other errors (and thus determine whether the unslept time
4082 is available in the interval pointed to by *remaining*).

4083 **RATIONALE**

4084 The *thrd_sleep()* function is identical to the *nanosleep()* function except that the return value
4085 may be any negative value when it fails with an error other than [EINTR].

4086 **FUTURE DIRECTIONS**

4087 None.

4088 **SEE ALSO**

4089 *nanosleep*

4090 XBD <**threads.h**>, <**time.h**>

4091 **CHANGE HISTORY**

4092 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4093 **NAME**

4094 *thrd_yield* — yield the processor

4095 **SYNOPSIS**

4096 `#include <threads.h>`

4097 `void thrd_yield(void);`

4098 **DESCRIPTION**

4099 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4100 Any conflict between the requirements described here and the ISO C standard is
4101 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4102 [CX]The *thrd_yield()* function shall force the running thread to relinquish the processor until
4103 it again becomes the head of its thread list.[/CX]

4104 **RETURN VALUE**

4105 This function shall not return a value.

4106 **ERRORS**

4107 No errors are defined.

4108 **EXAMPLES**

4109 None.

4110 **APPLICATION USAGE**

4111 See the APPLICATION USAGE section for *sched_yield()*.

4112 **RATIONALE**

4113 The *thrd_yield()* function is identical to the *sched_yield()* function except that it does not
4114 return a value.

4115 **FUTURE DIRECTIONS**

4116 None.

4117 **SEE ALSO**

4118 *sched_yield*

4119 XBD <**threads.h**>

4120 **CHANGE HISTORY**

4121 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4122 Ref 7.27.2.5

4123 On page 2161 line 69278 insert a new *timespec_get()* section:

4124 **NAME**

4125 *timespec_get* — get time

4126 **SYNOPSIS**

4127 `#include <time.h>`

4128 `int timespec_get(struct timespec *ts, int base);`

4129 **DESCRIPTION**

4130 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4131 Any conflict between the requirements described here and the ISO C standard is
4132 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4133 The *timespec_get()* function shall set the interval pointed to by *ts* to hold the current
4134 calendar time based on the specified time base.

4135 [CX]If *base* is `TIME_UTC`, the members of *ts* shall be set to the same values as would be
4136 set by a call to *clock_gettime(CLOCK_REALTIME, ts)*. If the number of seconds will not
4137 fit in an object of type **time_t**, the function shall return zero.[/CX]

4138 **RETURN VALUE**

4139 If the *timespec_get()* function is successful it shall return the non-zero value *base*; otherwise,
4140 it shall return zero.

4141 **ERRORS**

4142 See DESCRIPTION.

4143 **EXAMPLES**

4144 None.

4145 **APPLICATION USAGE**

4146 None.

4147 **RATIONALE**

4148 None.

4149 **FUTURE DIRECTIONS**

4150 None.

4151 **SEE ALSO**

4152 *clock_getres, time*

4153 XBD <**time.h**>

4154 **CHANGE HISTORY**

4155 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4156 Ref 7.21.4.4 para 4, 7.1.4 para 5

4157 On page 2164 line 69377 section *tmpnam()*, change:

4158 [CX]The *tmpnam()* function need not be thread-safe if called with a NULL parameter.[/CX]

4159 to:

4160 If called with a null pointer argument, the *tmpnam()* function need not be thread-safe;

4161 however, such calls shall avoid data races with calls to *tmpnam()* with a non-null argument

4162 and with calls to all other functions.

4163 Ref 7.30.3.2.1 para 4

4164 On page 2171 line 69568 section *towctrans()*, change:

4165 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped

4166 value of *wc* using the mapping described by *desc*. Otherwise, they shall return *wc*

4167 unchanged.

4168 to:

4169 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped

4170 value of *wc* using the mapping described by *desc*, or the value of *wc* unchanged if *desc* is

4171 zero. [CX]Otherwise, they shall return *wc* unchanged.[/CX]

4172 Ref F.10.6.8 para 2

4173 On page 2177 line 69716 section *trunc()*, add a new paragraph:

4174 [MX]These functions may raise the inexact floating-point exception for finite non-integer

4175 arguments.[/MX]

4176 Ref F.10.6.8 para 1,2

4177 On page 2177 line 69719 section trunc(), change:

4178 [MX]The result shall have the same sign as x.[/MX]

4179 to:

4180 [MX]The returned value shall be exact, shall be independent of the current rounding
4181 direction mode, and shall have the same sign as x.[/MX]

4182 Ref F.10.6.8 para 2

4183 On page 2177 line 69730 section trunc(), delete from APPLICATION USAGE:

4184 These functions may raise the inexact floating-point exception if the result differs in value
4185 from the argument.

4186 Ref 7.26.6

4187 On page 2182 line 69835 insert the following new tss_*() sections:

4188 **NAME**

4189 tss_create — thread-specific data key creation

4190 **SYNOPSIS**

4191 #include <threads.h>

4192 int tss_create(tss_t *key, tss_dtor_t dtor);

4193 **DESCRIPTION**

4194 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4195 Any conflict between the requirements described here and the ISO C standard is
4196 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4197 The *tss_create()* function shall create a thread-specific storage pointer with destructor *dtor*,
4198 which can be null.

4199 A null pointer value shall be associated with the newly created key in all existing threads.
4200 Upon subsequent thread creation, the value associated with all keys shall be initialized to a
4201 null pointer value in the new thread.

4202 Destructors associated with thread-specific storage shall not be invoked at process
4203 termination.

4204 The behavior is undefined if the *tss_create()* function is called from within a destructor.

4205 [CX]The *tss_create()* function shall not be affected if the calling thread executes a signal
4206 handler during the call.[/CX]

4207 **RETURN VALUE**

4208 If the *tss_create()* function is successful, it shall set the thread-specific storage pointed to by
4209 *key* to a value that uniquely identifies the newly created pointer and shall return
4210 *thrd_success*; otherwise, *thrd_error* shall be returned and the thread-specific storage

4211 pointed to by *key* has an indeterminate value.

4212 **ERRORS**

4213 No errors are defined.

4214 **EXAMPLES**

4215 None.

4216 **APPLICATION USAGE**

4217 The *tss_create()* function performs no implicit synchronization. It is the responsibility of the
4218 programmer to ensure that it is called exactly once per key before use of the key.

4219 **RATIONALE**

4220 If the value associated with a key needs to be updated during the lifetime of the thread, it
4221 may be necessary to release the storage associated with the old value before the new value is
4222 bound. Although the *tss_set()* function could do this automatically, this feature is not needed
4223 often enough to justify the added complexity. Instead, the programmer is responsible for
4224 freeing the stale storage:

```
4225     old = tss_get(key);  
4226     new = allocate();  
4227     destructor(old);  
4228     tss_set(key, new);
```

4229 There is no notion of a destructor-safe function. If an application does not call *thrd_exit()* or
4230 *pthread_exit()* from a signal handler, or if it blocks any signal whose handler may call
4231 *thrd_exit()* or *pthread_exit()* while calling async-unsafe functions, all functions can be safely
4232 called from destructors.

4233 The *tss_create()* function is not affected by signal handlers for the reasons stated in [xref to
4234 XRAT B.2.3].

4235 **FUTURE DIRECTIONS**

4236 None.

4237 **SEE ALSO**

4238 *pthread_exit*, *pthread_key_create*, *thrd_exit*, *tss_delete*, *tss_get*

4239 XBD <**threads.h**>

4240 **CHANGE HISTORY**

4241 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4242 **NAME**

4243 *tss_delete* — thread-specific data key deletion

4244 **SYNOPSIS**

```
4245     #include <threads.h>
```

```
4246     void tss_delete(tss_t key);
```

4247 **DESCRIPTION**

4248 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4249 Any conflict between the requirements described here and the ISO C standard is
4250 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4251 The *tss_delete()* function shall release any resources used by the thread-specific storage
4252 identified by *key*. The thread-specific data values associated with *key* need not be null at the
4253 time *tss_delete()* is called. It is the responsibility of the application to free any application
4254 storage or perform any cleanup actions for data structures related to the deleted key or
4255 associated thread-specific data in any threads; this cleanup can be done either before or after
4256 *tss_delete()* is called.

4257 The application shall ensure that the *tss_delete()* function is only called with a value for *key*
4258 that was returned by a call to *tss_create()* before the thread commenced executing
4259 destructors.

4260 If *tss_delete()* is called while another thread is executing destructors, whether this will affect
4261 the number of invocations of the destructor associated with *key* on that thread is unspecified.

4262 The *tss_delete()* function shall be callable from within destructor functions. Calling
4263 *tss_delete()* shall not result in the invocation of any destructors. Any destructor function that
4264 was associated with *key* shall no longer be called upon thread exit.

4265 Any attempt to use *key* following the call to *tss_delete()* results in undefined behavior.

4266 [CX]The *tss_delete()* function shall not be affected if the calling thread executes a signal
4267 handler during the call.[/CX]

4268 **RETURN VALUE**

4269 This function shall not return a value.

4270 **ERRORS**

4271 No errors are defined.

4272 **EXAMPLES**

4273 None.

4274 **APPLICATION USAGE**

4275 None.

4276 **RATIONALE**

4277 A thread-specific data key deletion function has been included in order to allow the
4278 resources associated with an unused thread-specific data key to be freed. Unused thread-
4279 specific data keys can arise, among other scenarios, when a dynamically loaded module that
4280 allocated a key is unloaded.

4281 Conforming applications are responsible for performing any cleanup actions needed for data
4282 structures associated with the key to be deleted, including data referenced by thread-specific
4283 data values. No such cleanup is done by *tss_delete()*. In particular, destructor functions
4284 are not called. See the RATIONALE for *pthread_key_delete()* for the reasons for this
4285 division of responsibility.

4286 The *tss_delete()* function is not affected by signal handlers for the reasons stated in [xref to

4287 XRAT B.2.3].

4288 FUTURE DIRECTIONS

4289 None.

4290 SEE ALSO

4291 *pthread_key_create*, *tss_create*

4292 XBD <**threads.h**>

4293 CHANGE HISTORY

4294 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4295 NAME

4296 *tss_get*, *tss_set* — thread-specific data management

4297 SYNOPSIS

4298 `#include <threads.h>`

4299 `void *tss_get(tss_t key);`
4300 `int tss_set(tss_t key, void *val);`

4301 DESCRIPTION

4302 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4303 Any conflict between the requirements described here and the ISO C standard is
4304 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4305 The *tss_get()* function shall return the value for the current thread held in the thread-specific
4306 storage identified by *key*.

4307 The *tss_set()* function shall set the value for the current thread held in the thread-specific
4308 storage identified by *key* to *val*. This action shall not invoke the destructor associated with
4309 the key on the value being replaced.

4310 The application shall ensure that the *tss_get()* and *tss_set()* functions are only called with a
4311 value for *key* that was returned by a call to *tss_create()* before the thread commenced
4312 executing destructors.

4313 The effect of calling *tss_get()* or *tss_set()* after *key* has been deleted with *tss_delete()* is
4314 undefined.

4315 [CX]Both *tss_get()* and *tss_set()* can be called from a thread-specific data destructor
4316 function. A call to *tss_get()* for the thread-specific data key being destroyed shall return a
4317 null pointer, unless the value is changed (after the destructor starts) by a call to *tss_set()*.
4318 Calling *tss_set()* from a thread-specific data destructor function may result either in lost
4319 storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction)
4320 or in an infinite loop.

4321 These functions shall not be affected if the calling thread executes a signal handler during
4322 the call.[/CX]

4323 RETURN VALUE

4324 The *tss_get()* function shall return the value for the current thread. If no thread-specific data
4325 value is associated with *key*, then a null pointer shall be returned.

4326 The *tss_set()* function shall return *thrd_success* on success or *thrd_error* if the request
4327 could not be honored.

4328 **ERRORS**

4329 No errors are defined.

4330 **EXAMPLES**

4331 None.

4332 **APPLICATION USAGE**

4333 None.

4334 **RATIONALE**

4335 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
4336 B.2.3].

4337 **FUTURE DIRECTIONS**

4338 None.

4339 **SEE ALSO**

4340 *pthread_getspecific*, *tss_create*

4341 XBD <**threads.h**>

4342 **CHANGE HISTORY**

4343 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4344 Ref 7.31.11 para 2

4345 On page 2193 line 70145 section *ungetc()*, change FUTURE DIRECTIONS from:

4346 None.

4347 to:

4348 The ISO C standard states that the use of *ungetc()* on a binary stream where the file position
4349 indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction
4350 between binary and text streams, so this applies to all streams. This feature may be removed
4351 in a future version of this standard.

4352 Ref 7.29.6.3 para 1, 7.1.4 para 5

4353 On page 2242 line 71441 section *wcrtomb()*, change:

4354 [CX]The *wcrtomb()* function need not be thread-safe if called with a NULL *ps*
4355 argument.[/CX]

4356 to:

4357 If called with a null *ps* argument, the *wcrtomb()* function need not be thread-safe; however,

4358 such calls shall avoid data races with calls to *wcrtomb()* with a non-null argument and with
4359 calls to all other functions.

4360 Ref 7.29.6.4 para 1, 7.1.4 para 5
4361 On page 2266 line 72111 section *wcsrtombs()*, change:

4362 [CX]The *wcsnrtombs()* and *wcsrtombs()* functions need not be thread-safe if called with a
4363 NULL *ps* argument.[/CX]

4364 to:

4365 [CX]If called with a null *ps* argument, the *wcsnrtombs()* function need not be thread-safe;
4366 however, such calls shall avoid data races with calls to *wcsnrtombs()* with a non-null
4367 argument and with calls to all other functions.[/CX]

4368 If called with a null *ps* argument, the *wcsrtombs()* function need not be thread-safe;
4369 however, such calls shall avoid data races with calls to *wcsrtombs()* with a non-null
4370 argument and with calls to all other functions.

4371 Ref 7.22.7 para 1, 7.1.4 para 5
4372 On page 2292 line 72879 section *wctomb()*, change:

4373 [CX]The *wctomb()* function need not be thread-safe.[/CX]

4374 to:

4375 The *wctomb()* function need not be thread-safe; however, it shall avoid data races with all
4376 other functions.

4377 **Changes to XCU**

4378 Ref 7.22.2
4379 On page 2333 line 74167 section 1.1.2.2 Mathematical Functions, change:

4380 Section 7.20.2, Pseudo-Random Sequence Generation Functions

4381 to:

4382 Section 7.22.2, Pseudo-Random Sequence Generation Functions

4383 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)
4384 On page 2542 line 82220 section c99, rename the c99 page to c17.

4385 Ref 7.26
4386 On page 2545 line 82375 section c99 (now c17), change:

4387 ... , `<spawn.h>`, `<sys/socket.h>`, ...

4388 to:

4389 ... , <spawn.h>, <sys/socket.h>, <threads.h>, ...

4390 Ref 7.26

4391 On page 2545 line 82382 section c99 (now c17), change:

4392 This option shall make available all interfaces referenced in <pthread.h> and *pthread_kill()*
4393 and *pthread_sigmask()* referenced in <signal.h>.

4394 to:

4395 This option shall make available all interfaces referenced in <pthread.h> and <threads.h>,
4396 and also *pthread_kill()* and *pthread_sigmask()* referenced in <signal.h>.

4397 Ref 6.10.8.1 para 1 (__STDC_VERSION__)

4398 On page 2552-2553 line 82641-82677 section c99 (now c17), change CHANGE HISTORY to:

4399 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4400 Changes to XRAT

4401 Ref G.1 para 1

4402 On page 3483 line 117680 section A.1.7.1 Codes, add a new tagged paragraph:

4403 MXC This margin code is used to denote functionality related to the IEC 60559 Complex
4404 Floating-Point option.

4405 Ref (none)

4406 On page 3489 line 117909 section A.3 Definitions (Byte), change:

4407 alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types are now defined.

4408 to:

4409 alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types were first defined.

4410 Ref 5.1.2.4, 7.17.3

4411 On page 3515 line 118946 section A.4.12 Memory Synchronization, change:

4412 **A.4.12 Memory Synchronization**

4413 to:

4414 **A.4.12 Memory Ordering and Synchronization**

4415 *A.4.12.1 Memory Ordering*

4416 There is no additional rationale provided for this section.

4417 *A.4.12.2 Memory Synchronization*

4418 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4419 On page 3556 line 120684 section A.12.2 Utility Syntax Guidelines, change:

4420 Thus, they had to devise a new name, *c89* (now superseded by *c99*), rather than ...

4421 to:

4422 Thus, they had to devise a new name, *c89* (subsequently superseded by *c99* and now by
4423 *c17*), rather than ...

4424 Ref K.3.1.1

4425 On page 3567 line 121053 section B.2.2.1 POSIX.1 Symbols, add a new unnumbered subsection:

4426 **The `__STDC_WANT_LIB_EXT1__` Feature Test Macro**

4427 The ISO C standard specifies the feature test macro `__STDC_WANT_LIB_EXT1__` as the
4428 announcement mechanism for the application that it requires functionality from Annex K. It
4429 specifies that the symbols specified in Annex K (if supported) are made visible when
4430 `__STDC_WANT_LIB_EXT1__` is 1 and are not made visible when it is 0, but leaves it
4431 unspecified whether they are made visible when `__STDC_WANT_LIB_EXT1__` is
4432 undefined. POSIX.1 requires that they are not made visible when the macro is undefined
4433 (except for those symbols that are already explicitly allowed to be visible through the
4434 definition of `_POSIX_C_SOURCE` or `_XOPEN_SOURCE`, or both).

4435 POSIX.1 does not include the interfaces specified in Annex K of the ISO C standard, but
4436 allows the symbols to be made visible in headers when requested by the application in order
4437 that applications can use symbols from Annex K and symbols from POSIX.1 in the same
4438 translation unit.

4439 Ref 6.10.3.4

4440 On page 3570 line 121176 section B.2.2.2 The Name Space, change:

4441 as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C
4442 standard

4443 to:

4444 as described for macros that expand to their own name as in Section 6.10.3.4 of the ISO C
4445 standard

4446 Ref 7.5 para 2

4447 On page 3571 line 121228-121243 section B.2.3 Error Numbers, change:

4448 The ISO C standard requires that *errno* be an assignable lvalue. Originally, ...
4449 [...]

4450 ... using the return value for a mixed purpose was judged to be of limited use and
4451 error prone.

4452 to:

4453 The original ISO C standard just required that *errno* be a modifiable lvalue. Since the
4454 introduction of threads in 2011, the ISO C standard has instead required that *errno* be a
4455 macro which expands to a modifiable lvalue that has thread local storage duration.

4456 Ref 7.26
4457 On page 3575 line 121390 section B.2.3 Error Numbers, change:

4458 In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a
4459 special case since it will never occur.

4460 to:

4461 In particular, applications calling blocking interfaces need not handle any possible [EINTR]
4462 return as a special case since it will never occur. In the case of threads functions in
4463 `<threads.h>`, the requirement is stated in terms of the call not being affected if the calling
4464 thread executes a signal handler during the call, since these functions return errors in a
4465 different way and cannot distinguish an [EINTR] condition from other error conditions.

4466 Ref (none)
4467 On page 3733 line 128128 section C.2.6.4 Arithmetic Expansion, change:

4468 Although the ISO/IEC 9899: 1999 standard now requires support for ...

4469 to:

4470 Although the ISO C standard requires support for ...

4471 Ref 7.17
4472 On page 3789 line 129986 section E.1 Subprofiling Option Groups, change:

4473 by collecting sets of related functions

4474 to:

4475 by collecting sets of related functions and generic functions

4476 Ref 7.22.3.1, 7.27.2.5, 7.22.4
4477 On page 3789, 3792 line 130022-130032, 130112-130114 section E.1 Subprofiling Option Groups,
4478 add new functions (in sorted order) to the existing groups as indicated:

4479 POSIX_C_LANG_SUPPORT
4480 *aligned_alloc(), timespec_get()*

4481 POSIX_MULTI_PROCESS
4482 *at_quick_exit(), quick_exit()*

4483 Ref 7.17
4484 On page 3789 line 129991 section E.1 Subprofiling Option Groups, add:

4485 POSIX_C_LANG_ATOMICS: ISO C Atomic Operations
4486 *atomic_compare_exchange_strong(), atomic_compare_exchange_strong_explicit(),*
4487 *atomic_compare_exchange_weak(), atomic_compare_exchange_weak_explicit(),*
4488 *atomic_exchange(), atomic_exchange_explicit(), atomic_fetch_add(),*
4489 *atomic_fetch_add_explicit(), atomic_fetch_and(), atomic_fetch_and_explicit(),*
4490 *atomic_fetch_or(), atomic_fetch_or_explicit(), atomic_fetch_sub(),*
4491 *atomic_fetch_sub_explicit(), atomic_fetch_xor(), atomic_fetch_xor_explicit(),*

4492 *atomic_flag_clear(), atomic_flag_clear_explicit(), atomic_flag_test_and_set(),*
4493 *atomic_flag_test_and_set_explicit(), atomic_init(), atomic_is_lock_free(),*
4494 *atomic_load(), atomic_load_explicit(), atomic_signal_fence(),*
4495 *atomic_thread_fence(), atomic_store(), atomic_store_explicit(), kill_dependency()*

4496 Ref 7.26

4497 On page 3790 line 1300349 section E.1 Subprofiling Option Groups, add:

4498 POSIX_C_LANG_THREADS: ISO C Threads

4499 *call_once(), cnd_broadcast(), cnd_signal(), cnd_destroy(), cnd_init(),*
4500 *cnd_timedwait(), cnd_wait(), mtx_destroy(), mtx_init(), mtx_lock(), mtx_timedlock(),*
4501 *mtx_trylock(), mtx_unlock(), thrd_create(), thrd_current(), thrd_detach(),*
4502 *thrd_equal(), thrd_exit(), thrd_join(), thrd_sleep(), thrd_yield(), tss_create(),*
4503 *tss_delete(), tss_get(), tss_set()*

4504 POSIX_C_LANG_UCHAR: ISO C Unicode Utilities

4505 *c16rtomb(), c32rtomb(), mbrtoc16(), mbrtoc32()*