

1 **TODO**

2 Check for overlaps with Mantis bugs that get tagged tc3 or issue8 after 2021-08-12.

3 **Introduction**

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

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

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

11 **Global Change**

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

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

16 **Changes to XBD**

17 Ref G.1 para 1

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

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

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

22 Ref (none)

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

24 the ISO/IEC 9899: 1999 standard

25 to:

26 the ISO C standard

27 Ref 6.2.8

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

29 See also the ISO C standard, Section B3.

30 to:

31 See also the ISO C standard, Section 6.2.8.

32 Ref 5.1.2.4

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

34 **3.31 Atomic Operation**

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

39 Ref 7.26.3

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

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

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

48 Ref 5.1.2.4

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

50 **3.125 Data Race**

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

54 Ref 5.1.2.4

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

56 **3.215 Lock-Free Operation**

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

59 Ref 7.26.5.1

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

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

63 to:

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

66 Ref 7.26.4

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

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

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

74 Ref 7.26.5.5

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

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

78 to:

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

81 Ref 7.26.5.1

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

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

85 to:

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

88 Ref 5.1.2.4

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

90 **3.382 Synchronization Operation**

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

92 Ref 7.26.5.1

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

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

96 to:

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

- 99 • The initial thread.
- 100 • A thread created using *pthread_create()*.

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

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

109 Ref 5.1.2.4
110 On page 99 line 2752 section 3.407 Thread-Safe, change:

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

113 to:

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

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

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

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

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

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

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

130 to:

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

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

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

141 Ref 5.1.2.4, 7.17.3

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

143 **4.12 Memory Synchronization**

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

150 to:

151 **4.12 Memory Ordering and Synchronization**

152 **4.12.1 Memory Ordering**

153 *4.12.1.1 Data Races*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

218 **Note:** The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced
219 before”, “synchronizes with”, and “dependency-ordered before” relationships, with two
220 exceptions. The first exception is that a concatenation is not permitted to end with

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

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

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

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

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

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

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

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

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

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

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

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

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

263 If a side effect *X* on an atomic object *M* happens before a value computation *B* of *M*, then the

264 evaluation *B* shall take its value from *X* or from a side effect *Y* that follows *X* in the
265 modification order of *M*. (This is known as “write-read coherence”.)

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

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

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

277 4.12.1.2 Memory Order and Consistency

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

282 For `memory_order_relaxed`, no operation orders memory.

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

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

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

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

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

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

303 **Note:** Atomic operations specifying `memory_order_relaxed` are relaxed only with respect to

304 memory ordering. Implementations must still guarantee that any given atomic access to a
305 particular atomic object be indivisible with respect to all other atomic accesses to that object.

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

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

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

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

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

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

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

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

335 4.12.2 Memory Synchronization

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

342 Ref 7.26.3, 7.26.4

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

345 *cnd_broadcast()* *thrd_create()*
346 *cnd_signal()* *thrd_join()*

347 Ref 7.26.2.1, 7.26.4

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

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

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

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

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

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

370 to:

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

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

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

385 The *pthread_cond_clockwait()*, *pthread_cond_wait()*, and *pthread_cond_timedwait()*
386 functions shall synchronize memory on all calls that release and re-acquire the specified

387 mutex, including calls that return [EOWNERDEAD], both when the mutex is released and
388 when it is re-acquired.

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

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

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

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

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

403 Ref 7.26.4

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

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

408 Ref 7.12.1 para 7

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

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

412 to:

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

418 Ref 7.12.1 para 3

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

420 A ``pole error'' occurs if the mathematical result of the function is an exact infinity (for
421 example, *log(0.0)*).

422 to:

423 A ``pole error'' shall occur if the mathematical result of the function has an exact infinite
424 result as the finite input argument(s) are approached in the limit (for example, *log(0.0)*). The

425 description of each function lists any required pole errors; an implementation may define
426 additional pole errors, provided that such errors are consistent with the mathematical
427 definition of the function.

428 Ref 7.12.1 para 4

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

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

432 add:

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

436 Ref 7.29.1 para 5

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

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

445 Ref 7.3.1 para 2

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

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

452 Ref G.6 para 1

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

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

456 add:

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

460 Ref 7.3.9.3

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

462 The following shall be defined as macros.

463 double complex CMPLX(double x, double y);
464 float complex CMPLXF(float x, float y);
465 long double complex CMPLXL(long double x, long double y);

466 Ref 7.3.1 para 2

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

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

473 Ref 7.3.9.3

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

475 Ref 7.5 para 2

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

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

480 to:

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

483 Ref (none)

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

485 the ISO/IEC 9899: 1999 standard

486 to:

487 the ISO C standard

488 Ref 5.2.4.2.2 para 11

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

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

-1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

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

497 Ref 5.2.4.2.2 para 12

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

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

502 [math stuff]

503 FLT_DECIMAL_DIG 6

504 DBL_DECIMAL_DIG 10

505 LDBL_DECIMAL_DIG 10

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

508 Ref 5.2.4.2.2 para 14

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

510 Minimum positive floating-point number.

511 FLT_TRUE_MIN 1E-37

512 DBL_TRUE_MIN 1E-37

513 LDBL_TRUE_MIN 1E-37

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

517 Ref (none)

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

519 the ISO/IEC 9899: 1999 standard

520 to:

521 the ISO C standard

522 Ref 7.22.4.3

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

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

525 to:

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

528 Ref 5.2.4.2.1 para 2

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

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

535 to:

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

540 Ref (none)

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

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

543 to:

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

545 Ref 7.26.5.5

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

547 void pthread_exit(void *);

548 to:

549 _Noreturn void pthread_exit(void *);

550 Ref 7.13.2.1 para 1

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

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

554 to:

555 _Noreturn void longjmp(jmp_buf, int);
556 [CX]_Noreturn void siglongjmp(sigjmp_buf, int);[/CX]

557 Ref 7.15

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

559 **NAME**

560 stdalign.h — alignment macros

561 **SYNOPSIS**

562 `#include <stdalign.h>`

563 **DESCRIPTION**

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

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

568 `alignas` Expands to `__Alignas`

569 `alignof` Expands to `__Alignof`

570 `__alignas_is_defined`
571 Expands to the integer constant 1

572 `__alignof_is_defined`
573 Expands to the integer constant 1

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

576 **APPLICATION USAGE**

577 None.

578 **RATIONALE**

579 None.

580 **FUTURE DIRECTIONS**

581 None.

582 **SEE ALSO**

583 None.

584 **CHANGE HISTORY**

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

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

588 **NAME**

589 `stdatomic.h` — atomics

590 **SYNOPSIS**

591 `#include <stdatomic.h>`

592 **DESCRIPTION**

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

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

597 header nor support any of its facilities.

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

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

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

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

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

608 memory_order_relaxed
609 memory_order_consume
610 memory_order_acquire
611 memory_order_release
612 memory_order_acq_rel
613 memory_order_seq_cst

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

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

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

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

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

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

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

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

```

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

```

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

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

```

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

```

686 APPLICATION USAGE

687 None.

688 RATIONALE

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

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

696 FUTURE DIRECTIONS

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

699 **SEE ALSO**

700 Section 4.12.1

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

705 **CHANGE HISTORY**

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

707 Ref 7.31.9

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

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

710 Ref 7.19 para 2

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

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

713 Ref (none)

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

715 the ISO/IEC 9899: 1999 standard

716 to:

717 the ISO C standard

718 Ref 7.20.1.1 para 1

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

720 denotes a signed integer type

721 to:

722 denotes such a signed integer type

723 Ref 7.20.1.1 para 2

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

725 ... designates an unsigned integer type with width *N*. Thus, **uint24_t** denotes an unsigned
726 integer type ...

727 to:

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

730 Ref 7.21.1 para 2

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

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

734 to:

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

737 Ref 7.21.1 para 3

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

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

743 to:

744 None.

745 Ref 7.22.4.5 para 1

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

747 void _Exit(int);

748 to:

749 _Noreturn void _Exit(int);

750 Ref 7.22.4.1 para 1

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

752 void abort(void);

753 to:

754 _Noreturn void abort(void);

755 Ref 7.22.3.1, 7.22.4.3

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

757 void *aligned_alloc(size_t, size_t);
758 int at_quick_exit(void (*)(void));

759 Ref 7.22.4.4 para 1

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

761 void exit(int);

762 to:

763 `_Noreturn void exit(int);`

764 Ref 7.22.4.7

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

766 `_Noreturn void quick_exit(int);`

767 Ref 7.23

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

769 **NAME**

770 `stdnoreturn.h` — noreturn macro

771 **SYNOPSIS**

772 `#include <stdnoreturn.h>`

773 **DESCRIPTION**

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

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

779 **APPLICATION USAGE**

780 None.

781 **RATIONALE**

782 None.

783 **FUTURE DIRECTIONS**

784 None.

785 **SEE ALSO**

786 None.

787 **CHANGE HISTORY**

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

789 Ref G.7

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

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

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

800 $\cos(iy)$ = $\cosh(y)$
801 $\sin(iy)$ = $i \sinh(y)$
802 $\tan(iy)$ = $i \tanh(y)$
803 $\cosh(iy)$ = $\cos(y)$
804 $\sinh(iy)$ = $i \sin(y)$
805 $\tanh(iy)$ = $i \tan(y)$
806 $\operatorname{asin}(iy)$ = $i \operatorname{asinh}(y)$
807 $\operatorname{atan}(iy)$ = $i \operatorname{atanh}(y)$
808 $\operatorname{asinh}(iy)$ = $i \operatorname{asin}(y)$
809 $\operatorname{atanh}(iy)$ = $i \operatorname{atan}(y)$
810 [/MXC]

811 Ref (none)

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

813 the ISO/IEC 9899: 1999 standard

814 to:

815 the ISO C standard

816 Ref 7.26

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

818 **NAME**

819 threads.h — ISO C threads

820 **SYNOPSIS**

821 #include <threads.h>

822 **DESCRIPTION**

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

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

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

831 `thread_local` Expands to **`_Thread_local`**.

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

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

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

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

849 The <threads.h> header shall define the enumeration constants **mtx_plain**,
850 **mtx_recursive**, **mtx_timed**, **thrd_busy**, **thrd_error**, **thrd_nomem**, **thrd_success**
851 and **thrd_timedout**.

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

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

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

884 APPLICATION USAGE

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

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

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

896 RATIONALE

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

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

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

919 FUTURE DIRECTIONS

920 None.

921 SEE ALSO

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

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

927 CHANGE HISTORY

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

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

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

933 `time_t` `tv_sec` Seconds.
934 `long` `tv_nsec` Nanoseconds.

935 and change the members to:

936 `time_t` `tv_sec` Whole seconds.
937 `long` `tv_nsec` Nanoseconds [0, 999 999 999].

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

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

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

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

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

948 **NAME**
949 `uchar.h` — Unicode character handling

950 **SYNOPSIS**
951 `#include <uchar.h>`

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

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

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

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

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

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

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

962 prototypes shall be provided.

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

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

973 APPLICATION USAGE

974 None.

975 RATIONALE

976 None.

977 FUTURE DIRECTIONS

978 None.

979 SEE ALSO

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

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

982 CHANGE HISTORY

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

984 Ref 7.22.4.5 para 1

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

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

987 to:

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

989 Ref 7.29.1 para 2

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

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

992 to:

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

994 Changes to XSH

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

998 6. Functions shall prevent data races as follows: A function shall not directly or indirectly
 999 access objects accessible by threads other than the current thread unless the objects are
 1000 accessed directly or indirectly via the function's arguments. A function shall not directly or
 1001 indirectly modify objects accessible by threads other than the current thread unless the
 1002 objects are accessed directly or indirectly via the function's non-const arguments.
 1003 Implementations may share their own internal objects between threads if the objects are not
 1004 visible to applications and are protected against data races.

1005 7. Functions shall perform all operations solely within the current thread if those operations
 1006 have effects that are visible to applications.

1007 Ref K.3.1.1
 1008 On page 473 line 16283 section 2.2.1, add a new subsection:

1009 2.2.1.3 *The `__STDC_WANT_LIB_EXT1__` Feature Test Macro*

1010 A POSIX-conforming [XSI] or XSI-conforming[/XSI] application can define the feature test
 1011 macro `__STDC_WANT_LIB_EXT1__` before inclusion of any header.

1012 When an application includes a header described by POSIX.1-20xx, and when this feature
 1013 test macro is defined to have the value 1, the header may make visible those symbols
 1014 specified for the header in Annex K of the ISO C standard that are not already explicitly
 1015 permitted by POSIX.1-20xx to be made visible in the header. These symbols are listed in
 1016 [xref to 2.2.2].

1017 When an application includes a header described by POSIX.1-20xx, and when this feature
 1018 test macro is either undefined or defined to have the value 0, the header shall not make any
 1019 additional symbols visible that are not already made visible by the feature test macro
 1020 `_POSIX_C_SOURCE` [XSI] or `_XOPEN_SOURCE`[/XSI] as described above, except when
 1021 enabled by another feature test macro.

1022 Ref 7.31.8 para 1
 1023 On page 475 line 16347 section 2.2.2, insert a row in the table:

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

1024 Ref 7.31.15 para 1
 1025 On page 476 line 16373 section 2.2.2, insert a row in the table:

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

1026 Ref 7.31.8 para 1
 1027 On page 477 line 16410 section 2.2.2, insert a row in the table:

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

1028 Ref 7.31.14 para 1

1029 On page 477 line 16417 section 2.2.2, insert a row in the table:

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

1030 Ref K.3.4 - K.3.9

1031 On page 477 line 16436 section 2.2.2 The Name Space, add:

1032 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
1033 (see [xref to 2.2.1]), implementations may add symbols to the headers shown in the
1034 following table provided the identifiers for those symbols have one of the corresponding
1035 complete names in the table.

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

1036 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1
1037 (see [xref to 2.2.1]), if any header in the following table is included, macros with the
1038 complete names shown may be defined.

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

1039 **Note:** The above two tables only include those symbols from Annex K of the ISO C standard that
1040 are not already allowed to be visible by entries in earlier tables in this section.

1041 Ref 7.1.3 para 1

1042 On page 478 line 16438 section 2.2.2, change:

1043 With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that
1044 begin with an <underscore> and either an uppercase letter or another <underscore> are
1045 always reserved for any use by the implementation.

1046 to:

1047 With the exception of identifiers beginning with the prefix `_POSIX_` and those identifiers
1048 which are lexically identical to keywords defined by the ISO C standard (for example
1049 `_Bool`), all identifiers that begin with an <underscore> and either an uppercase letter or
1050 another <underscore> are always reserved for any use by the implementation.

1051 Ref 7.1.3 para 1

1052 On page 478 line 16448 section 2.2.2, change:

1053 that have external linkage are always reserved

1054 to:

1055 that have external linkage and *errno* are always reserved

1056 Ref 7.1.3 para 1

1057 On page 479 line 16453 section 2.2.2, add the following in the appropriate place in the list:

1058	aligned_alloc	c32rtomb
1059	at_quick_exit	call_once
1060	atomic_compare_exchange_strong	cnd_broadcast
1061	atomic_compare_exchange_strong_explicit	cnd_destroy
1062	atomic_compare_exchange_weak	cnd_init
1063	atomic_compare_exchange_weak_explicit	cnd_signal
1064	atomic_exchange	cnd_timedwait
1065	atomic_exchange_explicit	cnd_wait
1066	atomic_fetch_add	kill_dependency
1067	atomic_fetch_add_explicit	mbrtoc16
1068	atomic_fetch_and	mbrtoc32
1069	atomic_fetch_and_explicit	mtx_destroy
1070	atomic_fetch_or	mtx_init
1071	atomic_fetch_or_explicit	mtx_lock
1072	atomic_fetch_sub	mtx_timedlock
1073	atomic_fetch_sub_explicit	mtx_trylock
1074	atomic_fetch_xor	mtx_unlock
1075	atomic_fetch_xor_explicit	quick_exit
1076	atomic_flag_clear	thrd_create
1077	atomic_flag_clear_explicit	thrd_current
1078	atomic_flag_test_and_set	thrd_detach
1079	atomic_flag_test_and_set_explicit	thrd_equal
1080	atomic_init	thrd_exit
1081	atomic_is_lock_free	thrd_join
1082	atomic_load	thrd_sleep
1083	atomic_load_explicit	thrd_yield
1084	atomic_signal_fence	timespec_get
1085	atomic_store	tss_create
1086	atomic_store_explicit	tss_delete
1087	atomic_thread_fence	tss_get
1088	c16rtomb	tss_set

1089 Ref 7.1.2 para 4

1090 On page 480 line 16551 section 2.2.2, change:

1091 Prior to the inclusion of a header, the application shall not define any macros with names
1092 lexically identical to symbols defined by that header.

1093 to:

1094 Prior to the inclusion of a header, or when any macro defined in the header is expanded, the
1095 application shall not define any macros with names lexically identical to symbols defined by
1096 that header.

1097 Ref 7.26.5.1

1098 On page 490 line 16980 section 2.4.2 Realtime Signal Generation and Delivery, change:

1099 The function shall be executed in an environment as if it were the *start_routine* for a newly
1100 created thread with thread attributes specified by *sigev_notify_attributes*.

1101 to:

1102 The function shall be executed in a newly created thread as if it were the *start_routine* for a
1103 call to *pthread_create()* with the thread attributes specified by *sigev_notify_attributes*.

1104 Ref 7.14.1.1 para 5

1105 On page 493 line 17088 section 2.4.3 Signal Actions, change:

1106 with static storage duration

1107 to:

1108 with static or thread storage duration that is not a lock-free atomic object

1109 Ref 7.14.1.1 para 5

1110 On page 493 line 17090 section 2.4.3 Signal Actions, after applying bug 711 change:

1111 other than one of the functions and macros listed in the following table

1112 to:

1113 other than one of the functions and macros specified below as being async-signal-safe

1114 Ref 7.14.1.1 para 5

1115 On page 494 line 17133 section 2.4.3 Signal Actions, add *quick_exit()* to the table of async-signal-
1116 safe functions.

1117 Ref 7.14.1.1 para 5

1118 On page 494 line 17147 section 2.4.3 Signal Actions, change:

1119 Any function or function-like macro not in the above table may be unsafe with respect to
1120 signals.

1121 to:

1122 In addition, the functions in **<stdatomic.h>** other than *atomic_init()* shall be async-signal-
1123 safe when the atomic arguments are lock-free, and the *atomic_is_lock_free()* function shall
1124 be async-signal-safe when called with an atomic argument.

1125 All other functions (including generic functions) and function-like macros may be unsafe
1126 with respect to signals.

1127 Ref 7.21.2 para 7,8

1128 On page 496 line 17228 section 2.5 Standard I/O Streams, add a new paragraph:

1129 Each stream shall have an associated lock that is used to prevent data races when multiple
1130 threads of execution access a stream, and to restrict the interleaving of stream operations
1131 performed by multiple threads. Only one thread can hold this lock at a time. The lock shall
1132 be reentrant: a single thread can hold the lock multiple times at a given time. All functions
1133 that read, write, position, or query the position of a stream, [CX]except those with names
1134 ending *_unlocked*[/CX], shall lock the stream [CX] as if by a call to *flockfile()*[/CX] before
1135 accessing it and release the lock [CX] as if by a call to *funlockfile()*[/CX] when the access is
1136 complete.

1137 Ref (none)

1138 On page 498 line 17312 section 2.5.2 Stream Orientation and Encoding Rules, change:

1139 For conformance to the ISO/IEC 9899: 1999 standard, the definition of a stream includes an
1140 “orientation”.

1141 to:

1142 The definition of a stream includes an “orientation”.

1143 Ref 7.26.5.8

1144 On page 508 line 17720 section 2.8.4 Process Scheduling, change:

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

1146 to:

1147 When a running thread issues the *sched_yield()* or *thrd_yield()* function

1148 Ref 7.17.2.2 para 3, 7.22.2.2 para 3

1149 On page 513 line 17907,17916 section 2.9.1 Thread-Safety, add *atomic_init()* and *srand()* to the list
1150 of functions that need not be thread-safe.

1151 Ref 7.12.8.3, 7.22.4.8

1152 On page 513 line 17907-17927 section 2.9.1 Thread-Safety, delete the following from the list of
1153 functions that need not be thread-safe:

1154 *lgamma()*, *lgammaf()*, *lgammal()*, *system()*

1155 [Note to reviewers: deletion of *mblen\(\)*, *mbtowc\(\)*, and *wctomb\(\)* from this list is the subject of](#)
1156 [Mantis bug 708.](#)

1157 Ref 7.28.1 para 1

1158 On page 513 line 17928 section 2.9.1 Thread-Safety, change:

1159 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument.
1160 The *mbrlen()*, *mbrtowc()*, *mbsnrrowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrrowcs()*, and
1161 *wcsrtombs()* functions need not be thread-safe if passed a NULL *ps* argument.

1162 to:

1163 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a null pointer
1164 argument. The *c16rtomb()*, *c32rtomb()*, *mbrlen()*, *mbrtoc16()*, *mbrtoc32()*, *mbrtowc()*,
1165 *mbsnrtowcs()*, *mbsrtowcs()*, *wcrtomb()*, *wcsnrtombs()*, and *wcsrtombs()* functions need not
1166 be thread-safe if passed a null *ps* argument. The *lgamma()*, *lgammaf()*, and *lgammal()*
1167 functions shall be thread-safe [XSI]except that they need not avoid data races when storing a
1168 value in the *siggam* variable[XSI].

1169 Ref 7.1.4 para 5

1170 On page 513 line 17934 section 2.9.1 Thread-Safety, change:

1171 Implementations shall provide internal synchronization as necessary in order to satisfy this
1172 requirement.

1173 to:

1174 Some functions that are not required to be thread-safe are nevertheless required to avoid data
1175 races with either all or some other functions, as specified on their individual reference pages.

1176 Implementations shall provide internal synchronization as necessary in order to satisfy
1177 thread-safety requirements.

1178 Ref 7.26.5

1179 On page 513 line 17944 section 2.9.2 Thread IDs, change:

1180 The lifetime of a thread ID ends after the thread terminates if it was created with the
1181 *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or if *pthread_detach()* or
1182 *pthread_join()* has been called for that thread.

1183 to:

1184 The lifetime of a thread ID ends after the thread terminates if it was created using
1185 *pthread_create()* with the *detachstate* attribute set to *PTHREAD_CREATE_DETACHED* or
1186 if *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()* has been called for that
1187 thread.

1188 Ref 7.26.5

1189 On page 514 line 17950 section 2.9.2 Thread IDs, change:

1190 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1191 *pthread_detach()* or *pthread_join()*.

1192 to:

1193 If a thread is detached, its thread ID is invalid for use as an argument in a call to
1194 *pthread_detach()*, *pthread_join()*, *thrd_detach()* or *thrd_join()*.

1195 Ref 7.26.4

1196 On page 514 line 17956 section 2.9.3 Thread Mutexes, change:

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

1198 to:

1199 A thread shall become the owner of a mutex, *m*, of type **pthread_mutex_t** when one of the
1200 following occurs:

1201 Ref 7.26.3, 7.26.4

1202 On page 514 line 17972 section 2.9.3 Thread Mutexes, add two new paragraphs and lists:

1203 A thread shall become the owner of a mutex, *m*, of type **mtx_t** when one of the following
1204 occurs:

- 1205 • It calls *mtx_lock()* with *m* as the *mtx* argument and the call returns `thrd_success`.
- 1206 • It calls *mtx_trylock()* with *m* as the *mtx* argument and the call returns
1207 `thrd_success`.
- 1208 • It calls *mtx_timedlock()* with *m* as the *mtx* argument and the call returns
1209 `thrd_success`.
- 1210 • It calls *cond_wait()* with *m* as the *mtx* argument and the call returns `thrd_success`.
- 1211 • It calls *cond_timedwait()* with *m* as the *mtx* argument and the call returns
1212 `thrd_success` or `thrd_timedout`.

1213 The thread shall remain the owner of *m* until one of the following occurs:

- 1214 • It executes *mtx_unlock()* with *m* as the *mtx* argument.
- 1215 • It blocks in a call to *cond_wait()* with *m* as the *mtx* argument.
- 1216 • It blocks in a call to *cond_timedwait()* with *m* as the *mtx* argument.

1217 Ref 7.26.4

1218 On page 514 line 17980 section 2.9.3 Thread Mutexes, change:

1219 Robust mutexes provide a means to enable the implementation to notify other threads in the
1220 event of a process terminating while one of its threads holds a mutex lock.

1221 to:

1222 Robust mutexes provide a means to enable the implementation to notify other threads in the
1223 event of a process terminating while one of its threads holds a lock on a mutex of type
1224 **pthread_mutex_t**.

1225 Ref 7.26.5

1226 On page 517 line 18085 section 2.9.5 Thread Cancellation, change:

1227 The thread cancellation mechanism allows a thread to terminate the execution of any other
1228 thread in the process in a controlled manner.

1229 to:

1230 The thread cancellation mechanism allows a thread to terminate the execution of any thread
1231 in the process, except for threads created using *thrd_create()*, in a controlled manner.

1232 Ref 7.26.3, 7.26.5.6

1233 On page 518 line 18119-18137 section 2.9.5.2 Cancellation Points, add the following to the list of

1234 functions that are required to be cancellation points:

1235 *cnd_timedwait()*, *cnd_wait()*, *thrd_join()*, *thrd_sleep()*

1236 Ref 7.26.5

1237 On page 520 line 18225 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1238 Each thread maintains a list of cancellation cleanup handlers.

1239 to:

1240 Each thread that was not created using *thrd_create()* maintains a list of cancellation cleanup
1241 handlers.

1242 Ref 7.26.6.1

1243 On page 521 line 18240 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1244 as described for *pthread_key_create()*

1245 to:

1246 as described for *pthread_key_create()* and *tss_create()*

1247 Ref 7.26

1248 On page 523 line 18337 section 2.9.9 Synchronization Object Copies and Alternative Mappings,
1249 add a new sentence:

1250 For ISO C functions declared in **<threads.h>**, the above requirements shall apply as if
1251 condition variables of type **cnd_t** and mutexes of type **mtx_t** have a process-shared attribute
1252 that is set to PTHREAD_PROCESS_PRIVATE.

1253 Ref 7.26.3

1254 On page 547 line 19279 section 2.12.1 Defined Types, change:

1255 **pthread_cond_t**

1256 to

1257 **pthread_cond_t, cnd_t**

1258 Ref 7.26.6, 7.26.4

1259 On page 547 line 19281 section 2.12.1 Defined Types, change:

1260 **pthread_key_t**

1261 **pthread_mutex_t**

1262 to

1263 **pthread_key_t, tss_t**

1264 **pthread_mutex_t, mtx_t**

1265 Ref 7.26.2.1

1266 On page 547 line 19284 section 2.12.1 Defined Types, change:

1267 **pthread_once_t**

1268 to

1269 **pthread_once_t, once_flag**

1270 Ref 7.26.5

1271 On page 547 line 19287 section 2.12.1 Defined Types, change:

1272 **pthread_t**

1273 to

1274 **pthread_t, thrd_t**

1275 Ref 7.3.9.3

1276 On page 552 line 19370 insert a new CMPLX() section:

1277 **NAME**

1278 CMPLX — make a complex value

1279 **SYNOPSIS**

1280 `#include <complex.h>`

1281 `double complex CMPLX(double x, double y);`

1282 `float complex CMPLXF(float x, float y);`

1283 `long double complex CMPLXL(long double x, long double y);`

1284 **DESCRIPTION**

1285 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1286 Any conflict between the requirements described here and the ISO C standard is
1287 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1288 The CMPLX macros shall expand to an expression of the specified complex type, with the
1289 real part having the (converted) value of x and the imaginary part having the (converted)
1290 value of y . The resulting expression shall be suitable for use as an initializer for an object
1291 with static or thread storage duration, provided both arguments are likewise suitable.

1292 **RETURN VALUE**

1293 The CMPLX macros return the complex value $x + iy$ (where i is the imaginary unit).

1294 These macros shall behave as if the implementation supported imaginary types and the
1295 definitions were:

1296 `#define CMPLX(x, y) ((double complex)((double)(x) + \`
1297 `_Imaginary_I * (double)(y)))`

1298 `#define CMPLXF(x, y) ((float complex)((float)(x) + \`
1299 `_Imaginary_I * (float)(y)))`

1300 `#define CMPLXL(x, y) ((long double complex)((long double)(x) + \`
1301 `_Imaginary_I * (long double)(y)))`

1302 **ERRORS**

1303 No errors are defined.

1304 **EXAMPLES**

1305 None.

1306 **APPLICATION USAGE**

1307 None.

1308 **RATIONALE**

1309 None.

1310 **FUTURE DIRECTIONS**

1311 None.

1312 **SEE ALSO**

1313 XBD <**complex.h**>

1314 **CHANGE HISTORY**

1315 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1316 Ref 7.22.4.5 para 1

1317 On page 553 line 19384 section `_Exit()`, change:

1318 `void _Exit(int status);`

1319 `#include <unistd.h>`

1320 `void _exit(int status);`

1321 to:

1322 `_Noreturn void _Exit(int status);`

1323 `#include <unistd.h>`

1324 `_Noreturn void _exit(int status);`

1325 Ref 7.22.4.5 para 2

1326 On page 553 line 19396 section `_Exit()`, change:

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

1328 to:

1329 shall not call functions registered with `atexit()` nor `at_quick_exit()`, nor any registered signal
1330 handlers

1331 Ref (none)

1332 On page 557 line 19562 section `_Exit()`, change:

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

1334 to:

1335 The ISO/IEC 9899: 1999 standard added the `_Exit()` function

1336 Ref 7.22.4.3, 7.22.4.7

1337 On page 557 line 19568 section `_Exit()`, add `at_quick_exit` and `quick_exit` to the SEE ALSO section.

1338 Ref 7.22.4.1 para 1

1339 On page 565 line 19761 section `abort()`, change:

1340 `void abort(void);`

1341 to:

1342 `_Noreturn void abort(void);`

1343 Ref (none)

1344 On page 565 line 19785 section `abort()`, change:

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

1346 to:

1347 The ISO/IEC 9899: 1999 standard required (and the current standard still requires) the
1348 `abort()` function to be async-signal-safe.

1349 Ref 7.22.3.1

1350 On page 597 line 20771 insert the following new `aligned_alloc()` section:

1351 **NAME**

1352 `aligned_alloc` — allocate memory with a specified alignment

1353 **SYNOPSIS**

1354 `#include <stdlib.h>`

1355 `void *aligned_alloc(size_t alignment, size_t size);`

1356 **DESCRIPTION**

1357 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1358 Any conflict between the requirements described here and the ISO C standard is
1359 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1360 The `aligned_alloc()` function shall allocate unused space for an object whose alignment is
1361 specified by `alignment`, whose size in bytes is specified by `size` and whose value is
1362 indeterminate.

1363 The order and contiguity of storage allocated by successive calls to `aligned_alloc()` is
1364 unspecified. Each such allocation shall yield a pointer to an object disjoint from any other
1365 object. The pointer returned shall point to the start (lowest byte address) of the allocated
1366 space. If the value of `alignment` is not a valid alignment supported by the implementation, a
1367 null pointer shall be returned. If the space cannot be allocated, a null pointer shall be
1368 returned. If the size of the space requested is 0, the behavior is implementation-defined:
1369 either a null pointer shall be returned to indicate an error, or the behavior shall be as if the

1370 size were some non-zero value, except that the behavior is undefined if the returned pointer
1371 is used to access an object.

1372 For purposes of determining the existence of a data race, *aligned_alloc()* shall behave as
1373 though it accessed only memory locations accessible through its arguments and not other
1374 static duration storage. The function may, however, visibly modify the storage that it
1375 allocates. Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*,
1376 [ADV]*posix_memalign()*,[/ADV] [CX]*reallocarray()*,[/CX] and *realloc()* that allocate or
1377 deallocate a particular region of memory shall occur in a single total order (see [xref to XBD
1378 4.12.1]), and each such deallocation call shall synchronize with the next allocation (if any)
1379 in this order.

1380 RETURN VALUE

1381 Upon successful completion, *aligned_alloc()* shall return a pointer to the allocated space; if
1382 *size* is 0, the application shall ensure that the pointer is not used to access an object.

1383 Otherwise, it shall return a null pointer [CX]and set *errno* to indicate the error[/CX].

1384 ERRORS

1385 The *aligned_alloc()* function shall fail if:

1386 [CX][EINVAL] The value of *alignment* is not a valid alignment supported by the
1387 implementation.

1388 [ENOMEM] Insufficient storage space is available.[/CX]

1389 The *aligned_alloc()* function may fail if:

1390 [CX][EINVAL] *size* is 0 and the implementation does not support 0 sized allocations.[/
1391 CX]

1392 EXAMPLES

1393 None.

1394 APPLICATION USAGE

1395 None.

1396 RATIONALE

1397 See the RATIONALE for [xref to *malloc()*].

1398 FUTURE DIRECTIONS

1399 None.

1400 SEE ALSO

1401 *calloc*, *free*, *getrlimit*, *malloc*, *posix_memalign*, *realloc*

1402 XBD <stdlib.h>

1403 CHANGE HISTORY

1404 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1405 Ref 7.27.3, 7.1.4 para 5

1406 On page 600 line 20911 section `asctime()`, change:

1407 [CX]The `asctime()` function need not be thread-safe.[/CX]

1408 to:

1409 The `asctime()` function need not be thread-safe; however, `asctime()` shall avoid data races
1410 with all functions other than itself, `ctime()`, `gmtime()` and `localtime()`.

1411 Ref 7.22.4.3

1412 On page 618 line 21380 insert the following new `at_quick_exit()` section:

1413 **NAME**

1414 `at_quick_exit` — register a function to be called from `quick_exit()`

1415 **SYNOPSIS**

1416 `#include <stdlib.h>`

1417 `int at_quick_exit(void (*func)(void));`

1418 **DESCRIPTION**

1419 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1420 Any conflict between the requirements described here and the ISO C standard is
1421 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1422 The `at_quick_exit()` function shall register the function pointed to by `func`, to be called
1423 without arguments should `quick_exit()` be called. It is unspecified whether a call to the
1424 `at_quick_exit()` function that does not happen before the `quick_exit()` function is called will
1425 succeed.

1426 At least 32 functions can be registered with `at_quick_exit()`.

1427 [CX]After a successful call to any of the `exec` functions, any functions previously registered
1428 by `at_quick_exit()` shall no longer be registered.[/CX]

1429 **RETURN VALUE**

1430 Upon successful completion, `at_quick_exit()` shall return 0; otherwise, it shall return a non-
1431 zero value.

1432 **ERRORS**

1433 No errors are defined.

1434 **EXAMPLES**

1435 None.

1436 **APPLICATION USAGE**

1437 The `at_quick_exit()` function registrations are distinct from the `atexit()` registrations, so
1438 applications might need to call both registration functions with the same argument.

1439 The functions registered by a call to `at_quick_exit()` must return to ensure that all registered
1440 functions are called.

1441 The application should call `sysconf()` to obtain the value of `{ATEXIT_MAX}`, the number of
1442 functions that can be registered. There is no way for an application to tell how many
1443 functions have already been registered with `at_quick_exit()`.

1444 Since the behavior is undefined if the `quick_exit()` function is called more than once,
1445 portable applications calling `at_quick_exit()` must ensure that the `quick_exit()` function is not
1446 called when the functions registered by the `at_quick_exit()` function are called.

1447 If a function registered by the `at_quick_exit()` function is called and a portable application
1448 needs to stop further `quick_exit()` processing, it must call the `_exit()` function or the `_Exit()`
1449 function or one of the functions which cause abnormal process termination.

1450 **RATIONALE**

1451 None.

1452 **FUTURE DIRECTIONS**

1453 None.

1454 **SEE ALSO**

1455 `atexit`, `exec`, `exit`, `quick_exit`, `sysconf`

1456 XBD `<stdlib.h>`

1457 **CHANGE HISTORY**

1458 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1459 Ref 7.22.4.3

1460 On page 618 line 21381 section `atexit()`, change:

1461 `atexit` — register a function to run at process termination

1462 to:

1463 `atexit` — register a function to be called from `exit()` or after return from `main()`

1464 Ref 7.22.4.2 para 2, 7.22.4.3

1465 On page 618 line 21389 section `atexit()`, change:

1466 The `atexit()` function shall register the function pointed to by `func`, to be called without
1467 arguments at normal program termination. At normal program termination, all functions
1468 registered by the `atexit()` function shall be called, in the reverse order of their registration,
1469 except that a function is called after any previously registered functions that had already
1470 been called at the time it was registered. Normal termination occurs either by a call to `exit()`
1471 or a return from `main()`.

1472 to:

1473 The `atexit()` function shall register the function pointed to by `func`, to be called without
1474 arguments from `exit()`, or after return from the initial call to `main()`, or on the last thread
1475 termination. If the `exit()` function is called, it is unspecified whether a call to the `atexit()`
1476 function that does not happen before `exit()` is called will succeed.

1477 [Note to reviewers: the part about all registered functions being called in reverse order is duplicated](#)
1478 [on the exit\(\) page and is not needed here.](#)

1479 Ref 7.22.4.2 para 2

1480 On page 618 line 21405 section `atexit()`, insert a new first APPLICATION USAGE paragraph:

1481 The `atexit()` function registrations are distinct from the `at_quick_exit()` registrations, so
1482 applications might need to call both registration functions with the same argument.

1483 Ref 7.22.4.3

1484 On page 618 line 21410 section `atexit()`, change:

1485 Since the behavior is undefined if the `exit()` function is called more than once, portable
1486 applications calling `atexit()` must ensure that the `exit()` function is not called at normal
1487 process termination when all functions registered by the `atexit()` function are called.

1488 All functions registered by the `atexit()` function are called at normal process termination,
1489 which occurs by a call to the `exit()` function or a return from `main()` or on the last thread
1490 termination, when the behavior is as if the implementation called `exit()` with a zero argument
1491 at thread termination time.

1492 If, at normal process termination, a function registered by the `atexit()` function is called and a
1493 portable application needs to stop further `exit()` processing, it must call the `_exit()` function
1494 or the `_Exit()` function or one of the functions which cause abnormal process termination.

1495 to:

1496 Since the behavior is undefined if the `exit()` function is called more than once, portable
1497 applications calling `atexit()` must ensure that the `exit()` function is not called when the
1498 functions registered by the `atexit()` function are called.

1499 If a function registered by the `atexit()` function is called and a portable application needs to
1500 stop further `exit()` processing, it must call the `_exit()` function or the `_Exit()` function or one
1501 of the functions which cause abnormal process termination.

1502 Ref 7.22.4.3

1503 On page 619 line 21425 section `atexit()`, add `at_quick_exit` to the SEE ALSO section.

1504 Ref 7.16

1505 On page 624 line 21548 insert the following new `atomic_*` sections:

1506 **NAME**

1507 `atomic_compare_exchange_strong`, `atomic_compare_exchange_strong_explicit`,
1508 `atomic_compare_exchange_weak`, `atomic_compare_exchange_weak_explicit` — atomically
1509 compare and exchange the values of two objects

1510 **SYNOPSIS**

```
1511 #include <stdatomic.h>
1512 _Bool atomic_compare_exchange_strong(volatile A *object,
1513     C *expected, C desired);
1514 _Bool atomic_compare_exchange_strong_explicit(volatile A *object,
1515     C *expected, C desired, memory_order success,
```

```
1516     memory_order failure);
1517 _Bool atomic_compare_exchange_weak(volatile A *object,
1518     C *expected, C desired);
1519 _Bool atomic_compare_exchange_weak_explicit(volatile A *object,
1520     C *expected, C desired, memory_order success,
1521     memory_order failure);
```

1522 DESCRIPTION

1523 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1524 Any conflict between the requirements described here and the ISO C standard is
1525 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1526 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1527 `<stdatomic.h>` header nor support these generic functions.

1528 The `atomic_compare_exchange_strong_explicit()` generic function shall atomically compare
1529 the contents of the memory pointed to by *object* for equality with that pointed to by
1530 *expected*, and if true, shall replace the contents of the memory pointed to by *object*
1531 with *desired*, and if false, shall update the contents of the memory pointed to by *expected*
1532 with that pointed to by *object*. This operation shall be an atomic read-modify-write operation
1533 (see [xref to XBD 4.12.1]). If the comparison is true, memory shall be affected according to
1534 the value of *success*, and if the comparison is false, memory shall be affected according to
1535 the value of *failure*. The application shall ensure that *failure* is not
1536 `memory_order_release` nor `memory_order_acq_rel`, and shall ensure that *failure* is
1537 no stronger than *success*.

1538 The `atomic_compare_exchange_strong()` generic function shall be equivalent to
1539 `atomic_compare_exchange_strong_explicit()` called with *success* and *failure* both set to
1540 `memory_order_seq_cst`.

1541 The `atomic_compare_exchange_weak_explicit()` generic function shall be equivalent to
1542 `atomic_compare_exchange_strong_explicit()`, except that the compare-and-exchange
1543 operation may fail spuriously. That is, even when the contents of memory referred to by
1544 *expected* and *object* are equal, it may return zero and store back to *expected* the same
1545 memory contents that were originally there.

1546 The `atomic_compare_exchange_weak()` generic function shall be equivalent to
1547 `atomic_compare_exchange_weak_explicit()` called with *success* and *failure* both set to
1548 `memory_order_seq_cst`.

1549 RETURN VALUE

1550 These generic functions shall return the result of the comparison.

1551 ERRORS

1552 No errors are defined.

1553 EXAMPLES

1554 None.

1555 APPLICATION USAGE

1556 A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will
1557 be in a loop. For example:

```
1558     exp = atomic_load(&cur);
1559     do {
1560         des = function(exp);
1561     } while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

1562 When a compare-and-exchange is in a loop, the weak version will yield better performance
1563 on some platforms. When a weak compare-and-exchange would require a loop and a strong
1564 one would not, the strong one is preferable.

1565 **RATIONALE**

1566 None.

1567 **FUTURE DIRECTIONS**

1568 None.

1569 **SEE ALSO**

1570 XBD Section 4.12.1, <**stdatomic.h**>

1571 **CHANGE HISTORY**

1572 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1573 **NAME**

1574 `atomic_exchange`, `atomic_exchange_explicit` — atomically exchange the value of an object

1575 **SYNOPSIS**

```
1576     #include <stdatomic.h>
1577     C atomic_exchange(volatile A *object, C desired);
1578     C atomic_exchange_explicit(volatile A *object,
1579                               C desired, memory_order order);
```

1580 **DESCRIPTION**

1581 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1582 Any conflict between the requirements described here and the ISO C standard is
1583 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1584 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1585 <**stdatomic.h**> header nor support these generic functions.

1586 The `atomic_exchange_explicit()` generic function shall atomically replace the value pointed
1587 to by *object* with *desired*. This operation shall be an atomic read-modify-write operation (see
1588 [xref to XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1589 The `atomic_exchange()` generic function shall be equivalent to `atomic_exchange_explicit()`
1590 called with *order* set to `memory_order_seq_cst`.

1591 **RETURN VALUE**

1592 These generic functions shall return the value pointed to by *object* immediately before the
1593 effects.

1594 **ERRORS**

1595 No errors are defined.

1596 **EXAMPLES**

1597 None.

1598 **APPLICATION USAGE**

1599 None.

1600 **RATIONALE**

1601 None.

1602 **FUTURE DIRECTIONS**

1603 None.

1604 **SEE ALSO**

1605 XBD Section 4.12.1, <**stdatomic.h**>

1606 **CHANGE HISTORY**

1607 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1608 **NAME**

1609 `atomic_fetch_add`, `atomic_fetch_add_explicit`, `atomic_fetch_and`,
1610 `atomic_fetch_and_explicit`, `atomic_fetch_or`, `atomic_fetch_or_explicit`, `atomic_fetch_sub`,
1611 `atomic_fetch_sub_explicit`, `atomic_fetch_xor`, `atomic_fetch_xor_explicit` — atomically
1612 replace the value of an object with the result of a computation

1613 **SYNOPSIS**

```
1614 #include <stdatomic.h>
1615 C atomic_fetch_add(volatile A *object, M operand);
1616 C atomic_fetch_add_explicit(volatile A *object, M operand,
1617 memory_order order);
1618 C atomic_fetch_and(volatile A *object, M operand);
1619 C atomic_fetch_and_explicit(volatile A *object, M operand,
1620 memory_order order);
1621 C atomic_fetch_or(volatile A *object, M operand);
1622 C atomic_fetch_or_explicit(volatile A *object, M operand,
1623 memory_order order);
1624 C atomic_fetch_sub(volatile A *object, M operand);
1625 C atomic_fetch_sub_explicit(volatile A *object, M operand,
1626 memory_order order);
1627 C atomic_fetch_xor(volatile A *object, M operand);
1628 C atomic_fetch_xor_explicit(volatile A *object, M operand,
1629 memory_order order);
```

1630 **DESCRIPTION**

1631 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1632 Any conflict between the requirements described here and the ISO C standard is
1633 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1634 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1635 `<stdatomic.h>` header nor support these generic functions.

1636 The `atomic_fetch_add_explicit()` generic function shall atomically replace the value pointed
1637 to by `object` with the result of adding `operand` to this value. This operation shall be an
1638 atomic read-modify-write operation (see [xref to XBD 4.12.1]). Memory shall be affected

1639 according to the value of *order*.

1640 The *atomic_fetch_add()* generic function shall be equivalent to *atomic_fetch_add_explicit()*
1641 called with *order* set to *memory_order_seq_cst*.

1642 The other *atomic_fetch_**() generic functions shall be equivalent to
1643 *atomic_fetch_add_explicit()* if their name ends with *explicit*, or to *atomic_fetch_add()* if it
1644 does not, respectively, except that they perform the computation indicated in their name,
1645 instead of addition:

1646 *sub* subtraction
1647 *or* bitwise inclusive OR
1648 *xor* bitwise exclusive OR
1649 *and* bitwise AND

1650 For addition and subtraction, the application shall ensure that **A** is an atomic integer type or
1651 an atomic pointer type and is not **atomic_bool**. For the other operations, the application
1652 shall ensure that **A** is an atomic integer type and is not **atomic_bool**.

1653 For signed integer types, the computation shall silently wrap around on overflow; there are
1654 no undefined results. For pointer types, the result can be an undefined address, but the
1655 computations otherwise have no undefined behavior.

1656 **RETURN VALUE**

1657 These generic functions shall return the value pointed to by *object* immediately before the
1658 effects.

1659 **ERRORS**

1660 No errors are defined.

1661 **EXAMPLES**

1662 None.

1663 **APPLICATION USAGE**

1664 The operation of these generic functions is nearly equivalent to the operation of the
1665 corresponding compound assignment operators +=, -=, etc. The only differences are that the
1666 compound assignment operators are not guaranteed to operate atomically, and the value
1667 yielded by a compound assignment operator is the updated value of the object, whereas the
1668 value returned by these generic functions is the previous value of the atomic object.

1669 **RATIONALE**

1670 None.

1671 **FUTURE DIRECTIONS**

1672 None.

1673 **SEE ALSO**

1674 XBD Section 4.12.1, <**stdatomic.h**>

1675 **CHANGE HISTORY**

1676 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1677 **NAME**

1678 atomic_flag_clear, atomic_flag_clear_explicit — clear an atomic flag

1679 **SYNOPSIS**

```
1680 #include <stdatomic.h>
1681 void atomic_flag_clear(volatile atomic_flag *object);
1682 void atomic_flag_clear_explicit(
1683     volatile atomic_flag *object, memory_order order);
```

1684 **DESCRIPTION**

1685 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1686 Any conflict between the requirements described here and the ISO C standard is
1687 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1688 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1689 `<stdatomic.h>` header nor support these functions.

1690 The `atomic_flag_clear_explicit()` function shall atomically place the atomic flag pointed to
1691 by `object` into the clear state. Memory shall be affected according to the value of `order`,
1692 which the application shall ensure is not `memory_order_acquire` nor
1693 `memory_order_acq_rel`.

1694 The `atomic_flag_clear()` function shall be equivalent to `atomic_flag_clear_explicit()` called
1695 with `order` set to `memory_order_seq_cst`.

1696 **RETURN VALUE**

1697 These functions shall not return a value.

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_flag_test_and_set, atomic_flag_test_and_set_explicit — test and set an atomic flag

1714 **SYNOPSIS**

```
1715     #include <stdatomic.h>
1716     _Bool atomic_flag_test_and_set(volatile atomic_flag *object);
1717     _Bool atomic_flag_test_and_set_explicit(
1718         volatile atomic_flag *object, memory_order order);
```

1719 **DESCRIPTION**

1720 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1721 Any conflict between the requirements described here and the ISO C standard is
1722 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1723 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1724 `<stdatomic.h>` header nor support these functions.

1725 The `atomic_flag_test_and_set_explicit()` function shall atomically place the atomic flag
1726 pointed to by `object` into the set state and return the value corresponding to the immediately
1727 preceding state. This operation shall be an atomic read-modify-write operation (see [xref to
1728 XBD 4.12.1]). Memory shall be affected according to the value of `order`.

1729 The `atomic_flag_test_and_set()` function shall be equivalent to
1730 `atomic_flag_test_and_set_explicit()` called with `order` set to `memory_order_seq_cst`.

1731 **RETURN VALUE**

1732 These functions shall return the value that corresponds to the state of the atomic flag
1733 immediately before the effects. The return value true shall correspond to the set state and the
1734 return value false shall correspond to the clear state.

1735 **ERRORS**

1736 No errors are defined.

1737 **EXAMPLES**

1738 None.

1739 **APPLICATION USAGE**

1740 None.

1741 **RATIONALE**

1742 None.

1743 **FUTURE DIRECTIONS**

1744 None.

1745 **SEE ALSO**

1746 XBD Section 4.12.1, `<stdatomic.h>`

1747 **CHANGE HISTORY**

1748 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1749 **NAME**

1750 `atomic_init` — initialize an atomic object

1751 **SYNOPSIS**

```
1752     #include <stdatomic.h>
1753     void atomic_init(volatile A *obj, C value);
```

1754 **DESCRIPTION**

1755 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1756 Any conflict between the requirements described here and the ISO C standard is
1757 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1758 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1759 `<stdatomic.h>` header nor support this generic function.

1760 The `atomic_init()` generic function shall initialize the atomic object pointed to by `obj` to the
1761 value `value`, while also initializing any additional state that the implementation might need
1762 to carry for the atomic object.

1763 Although this function initializes an atomic object, it does not avoid data races; concurrent
1764 access to the variable being initialized, even via an atomic operation, constitutes a data race.

1765 **RETURN VALUE**

1766 The `atomic_init()` generic function shall not return a value.

1767 **ERRORS**

1768 No errors are defined.

1769 **EXAMPLES**

```
1770     atomic_int guide;
1771     atomic_init(&guide, 42);
```

1772 **APPLICATION USAGE**

1773 None.

1774 **RATIONALE**

1775 None.

1776 **FUTURE DIRECTIONS**

1777 None.

1778 **SEE ALSO**

1779 XBD `<stdatomic.h>`

1780 **CHANGE HISTORY**

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

1782 **NAME**

1783 `atomic_is_lock_free` — indicate whether or not atomic operations are lock-free

1784 **SYNOPSIS**

```
1785     #include <stdatomic.h>
1786     _Bool atomic_is_lock_free(const volatile A *obj);
```

1787 **DESCRIPTION**

1788 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1789 Any conflict between the requirements described here and the ISO C standard is
1790 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1791 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1792 `<stdatomic.h>` header nor support this generic function.

1793 The `atomic_is_lock_free()` generic function shall indicate whether or not atomic operations
1794 on objects of the type pointed to by `obj` are lock-free; `obj` can be a null pointer.

1795 **RETURN VALUE**

1796 The `atomic_is_lock_free()` generic function shall return a non-zero value if and only if
1797 atomic operations on objects of the type pointed to by `obj` are lock-free. During the lifetime
1798 of the calling process, the result of the lock-free query shall be consistent for all pointers of
1799 the same type.

1800 **ERRORS**

1801 No errors are defined.

1802 **EXAMPLES**

1803 None.

1804 **APPLICATION USAGE**

1805 None.

1806 **RATIONALE**

1807 Operations that are lock-free should also be address-free. That is, atomic operations on the
1808 same memory location via two different addresses will communicate atomically. The
1809 implementation should not depend on any per-process state. This restriction enables
1810 communication via memory mapped into a process more than once and memory shared
1811 between two processes.

1812 **FUTURE DIRECTIONS**

1813 None.

1814 **SEE ALSO**

1815 XBD `<stdatomic.h>`

1816 **CHANGE HISTORY**

1817 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1818 **NAME**

1819 `atomic_load`, `atomic_load_explicit` — atomically obtain the value of an object

1820 **SYNOPSIS**

```
1821 #include <stdatomic.h>  
1822 C atomic_load(const volatile A *object);  
1823 C atomic_load_explicit(const volatile A *object,  
1824     memory_order order);
```

1825 **DESCRIPTION**

1826 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1827 Any conflict between the requirements described here and the ISO C standard is
1828 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1829 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1830 `<stdatomic.h>` header nor support these generic functions.

1831 The `atomic_load_explicit()` generic function shall atomically obtain the value pointed to by
1832 *object*. Memory shall be affected according to the value of *order*, which the application shall
1833 ensure is not `memory_order_release` nor `memory_order_acq_rel`.

1834 The `atomic_load()` generic function shall be equivalent to `atomic_load_explicit()` called with
1835 *order* set to `memory_order_seq_cst`.

1836 RETURN VALUE

1837 These generic functions shall return the value pointed to by *object*.

1838 ERRORS

1839 No errors are defined.

1840 EXAMPLES

1841 None.

1842 APPLICATION USAGE

1843 None.

1844 RATIONALE

1845 None.

1846 FUTURE DIRECTIONS

1847 None.

1848 SEE ALSO

1849 XBD Section 4.12.1, `<stdatomic.h>`

1850 CHANGE HISTORY

1851 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1852 NAME

1853 `atomic_signal_fence`, `atomic_thread_fence` — fence operations

1854 SYNOPSIS

```
1855 #include <stdatomic.h>  
1856 void atomic_signal_fence(memory_order order);  
1857 void atomic_thread_fence(memory_order order);
```

1858 DESCRIPTION

1859 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1860 Any conflict between the requirements described here and the ISO C standard is
1861 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1862 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1863 `<stdatomic.h>` header nor support these functions.

1864 The *atomic_signal_fence()* and *atomic_thread_fence()* functions provide synchronization
1865 primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A
1866 fence with acquire semantics is called an *acquire fence*; a fence with release semantics is
1867 called a *release fence*.

1868 A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X*
1869 and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X*
1870 modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* or a value written
1871 by any side effect in the hypothetical release sequence *X* would head if it were a release
1872 operation.

1873 A release fence *A* synchronizes with an atomic operation *B* that performs an acquire
1874 operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is
1875 sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by
1876 any side effect in the hypothetical release sequence *X* would head if it were a release
1877 operation.

1878 An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with
1879 an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced
1880 before *B* and reads the value written by *A* or a value written by any side effect in the release
1881 sequence headed by *A*.

1882 Depending on the value of *order*, the operation performed by *atomic_thread_fence()* shall:

- 1883 • have no effects, if *order* is equal to `memory_order_relaxed`;
- 1884 • be an acquire fence, if *order* is equal to `memory_order_acquire` or
1885 `memory_order_consume`;
- 1886 • be a release fence, if *order* is equal to `memory_order_release`;
- 1887 • be both an acquire fence and a release fence, if *order* is equal to
1888 `memory_order_acq_rel`;
- 1889 • be a sequentially consistent acquire and release fence, if *order* is equal to
1890 `memory_order_seq_cst`.

1891 The *atomic_signal_fence()* function shall be equivalent to *atomic_thread_fence()*, except
1892 that the resulting ordering constraints shall be established only between a thread and a signal
1893 handler executed in the same thread.

1894 **RETURN VALUE**

1895 These functions shall not return a value.

1896 **ERRORS**

1897 No errors are defined.

1898 **EXAMPLES**

1899 None.

1900 **APPLICATION USAGE**

1901 The *atomic_signal_fence()* function can be used to specify the order in which actions
1902 performed by the thread become visible to the signal handler. Implementation reorderings of
1903 loads and stores are inhibited in the same way as with *atomic_thread_fence()*, but the
1904 hardware fence instructions that *atomic_thread_fence()* would have inserted are not
1905 emitted.

1906 **RATIONALE**

1907 None.

1908 **FUTURE DIRECTIONS**

1909 None.

1910 **SEE ALSO**

1911 XBD Section 4.12.1, <**stdatomic.h**>

1912 **CHANGE HISTORY**

1913 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1914 **NAME**

1915 *atomic_store*, *atomic_store_explicit* — atomically store a value in an object

1916 **SYNOPSIS**

```
1917 #include <stdatomic.h>
1918 void atomic_store(volatile A *object, C desired);
1919 void atomic_store_explicit(volatile A *object, C desired,
1920 memory_order order);
```

1921 **DESCRIPTION**

1922 [CX] The functionality described on this reference page is aligned with the ISO C standard.
1923 Any conflict between the requirements described here and the ISO C standard is
1924 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1925 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
1926 `<stdatomic.h>` header nor support these generic functions.

1927 The *atomic_store_explicit()* generic function shall atomically replace the value pointed to by
1928 *object* with the value of *desired*. Memory shall be affected according to the value of *order*,
1929 which the application shall ensure is not `memory_order_acquire`,
1930 `memory_order_consume`, nor `memory_order_acq_rel`.

1931 The *atomic_store()* generic function shall be equivalent to *atomic_store_explicit()* called
1932 with *order* set to `memory_order_seq_cst`.

1933 **RETURN VALUE**

1934 These generic functions shall not return a value.

1935 **ERRORS**

1936 No errors are defined.

1937 **EXAMPLES**

1938 None.

1939 **APPLICATION USAGE**

1940 None.

1941 **RATIONALE**

1942 None.

1943 **FUTURE DIRECTIONS**

1944 None.

1945 **SEE ALSO**

1946 XBD Section 4.12.1, <**stdatomic.h**>

1947 **CHANGE HISTORY**

1948 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1949 Ref 7.28.1, 7.1.4 para 5

1950 On page 633 line 21891 insert a new `c16rtomb()` section:

1951 **NAME**

1952 `c16rtomb`, `c32rtomb` — convert a Unicode character code to a character (restartable)

1953 **SYNOPSIS**

1954 `#include <uchar.h>`

1955 `size_t c16rtomb(char *restrict s, char16_t c16,`

1956 `mbstate_t *restrict ps);`

1957 `size_t c32rtomb(char *restrict s, char32_t c32,`

1958 `mbstate_t *restrict ps);`

1959 **DESCRIPTION**

1960 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1961 Any conflict between the requirements described here and the ISO C standard is

1962 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1963 If *s* is a null pointer, the `c16rtomb()` function shall be equivalent to the call:

1964 `c16rtomb(buf, L'\0', ps)`

1965 where *buf* is an internal buffer.

1966 If *s* is not a null pointer, the `c16rtomb()` function shall determine the number of bytes needed

1967 to represent the character that corresponds to the wide character given by *c16* (including any

1968 shift sequences), and store the resulting bytes in the array whose first element is pointed to

1969 by *s*. At most {MB_CUR_MAX} bytes shall be stored. If *c16* is a null wide character, a null

1970 byte shall be stored, preceded by any shift sequence needed to restore the initial shift state;

1971 the resulting state described shall be the initial conversion state.

1972 If *ps* is a null pointer, the `c16rtomb()` function shall use its own internal **mbstate_t** object,

1973 which shall be initialized at program start-up to the initial conversion state. Otherwise, the

1974 **mbstate_t** object pointed to by *ps* shall be used to completely describe the current

1975 conversion state of the associated character sequence.

1976 The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

- 1977 The *mbrtoc16()* function shall not change the setting of *errno* if successful.
- 1978 The *c32rtomb()* function shall behave the same way as *c16rtomb()* except that the second
1979 parameter shall be an object of type **char32_t** instead of **char16_t**. References to *c16* in the
1980 above description shall apply as if they were *c32* when they are being read as describing
1981 *c32rtomb()*.
- 1982 If called with a null *ps* argument, the *c16rtomb()* function need not be thread-safe; however,
1983 such calls shall avoid data races with calls to *c16rtomb()* with a non-null argument and with
1984 calls to all other functions.
- 1985 If called with a null *ps* argument, the *c32rtomb()* function need not be thread-safe; however,
1986 such calls shall avoid data races with calls to *c32rtomb()* with a non-null argument and with
1987 calls to all other functions.
- 1988 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
1989 calls *c16rtomb()* or *c32rtomb()* with a null pointer for *ps*.
- 1990 **RETURN VALUE**
- 1991 These functions shall return the number of bytes stored in the array object (including any
1992 shift sequences). When *c16* or *c32* is not a valid wide character, an encoding error shall
1993 occur. In this case, the function shall store the value of the macro [EILSEQ] in *errno* and
1994 shall return (**size_t**)-1; the conversion state is unspecified.
- 1995 **ERRORS**
- 1996 These function shall fail if:
- 1997 [EILSEQ] An invalid wide-character code is detected.
- 1998 These functions may fail if:
- 1999 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]
- 2000 **EXAMPLES**
- 2001 None.
- 2002 **APPLICATION USAGE**
- 2003 None.
- 2004 **RATIONALE**
- 2005 None.
- 2006 **FUTURE DIRECTIONS**
- 2007 None.
- 2008 **SEE ALSO**
- 2009 *mbrtoc16*
- 2010 XBD <**uchar.h**>
- 2011 **CHANGE HISTORY**
- 2012 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

- 2013 Ref G.6 para 6, F.10.4.3, F.10.4.2, F.10 para 11
2014 On page 633 line 21905 section `cabs()`, add:
- 2015 [MXC]`cabs(x + iy)`, `cabs(y + ix)`, and `cabs(x - iy)` shall return exactly the same value.
- 2016 If z is $\pm 0 \pm i0$, $+0$ shall be returned.
- 2017 If the real or imaginary part of z is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned, even if the other part is NaN .
- 2018 If the real or imaginary part of z is NaN and the other part is not $\pm\text{Inf}$, NaN shall be returned.
2019 [/MXC]
- 2020 Ref G.6.1.1
2021 On page 634 line 21935 section `cacos()`, add:
- 2022 [MXC]`cacos(conj(z))`, `cacosf(conjf(z))` and `cacosl(conjl(z))` shall return exactly the same
2023 value as `conj(cacos(z))`, `conjf(cacosf(z))` and `conjl(cacosl(z))`, respectively, including for the
2024 special values of z below.
- 2025 If z is $\pm 0 + i0$, $\pi/2 - i0$ shall be returned.
- 2026 If z is $\pm 0 + i\text{NaN}$, $\pi/2 + i\text{NaN}$ shall be returned.
- 2027 If z is $x + i\text{Inf}$ where x is finite, $\pi/2 - i\text{Inf}$ shall be returned.
- 2028 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
2029 floating-point exception may be raised.
- 2030 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $\pi - i\text{Inf}$ shall be returned.
- 2031 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 - i\text{Inf}$ shall be returned.
- 2032 If z is $-\text{Inf} + i\text{Inf}$, $3\pi/4 - i\text{Inf}$ shall be returned.
- 2033 If z is $+\text{Inf} + i\text{Inf}$, $\pi/4 - i\text{Inf}$ shall be returned.
- 2034 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
2035 is unspecified.
- 2036 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2037 point exception may be raised.
- 2038 If z is $\text{NaN} + i\text{Inf}$, $\text{NaN} - i\text{Inf}$ shall be returned.
- 2039 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} - i\text{NaN}$ shall be returned. [/MXC]
- 2040 Ref G.6.2.1
2041 On page 635 line 21966 section `cacosh()`, add:
- 2042 [MXC]`cacosh(conj(z))`, `cacoshf(conjf(z))` and `cacoshl(conjl(z))` shall return exactly the same
2043 value as `conj(cacosh(z))`, `conjf(cacoshf(z))` and `conjl(cacoshl(z))`, respectively, including for
2044 the special values of z below.

2045 If z is $\pm 0 + i0$, $+0 + i\pi/2$ shall be returned.

2046 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2047 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
2048 unspecified.

2049 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
2050 floating-point exception may be raised.

2051 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2052 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2053 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

2054 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2055 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2056 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2057 point exception may be raised.

2058 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2059 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2060 Ref 7.26.2.1

2061 On page 637 line 21989 insert the following new `call_once()` section:

2062 **NAME**

2063 `call_once` — dynamic package initialization

2064 **SYNOPSIS**

2065 `#include <threads.h>`

2066 `void call_once(once_flag *flag, void (*init_routine)(void));`

2067 `once_flag flag = ONCE_FLAG_INIT;`

2068 **DESCRIPTION**

2069 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2070 Any conflict between the requirements described here and the ISO C standard is
2071 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2072 The `call_once()` function shall use the **once_flag** pointed to by `flag` to ensure that
2073 `init_routine` is called exactly once, the first time the `call_once()` function is called with that
2074 value of `flag`. Completion of an effective call to the `call_once()` function shall synchronize
2075 with all subsequent calls to the `call_once()` function with the same value of `flag`.

2076 [CX]The `call_once()` function is not a cancellation point. However, if `init_routine` is a
2077 cancellation point and is canceled, the effect on `flag` shall be as if `call_once()` was never
2078 called.

2079 If the call to *init_routine* is terminated by a call to *longjmp()* or *siglongjmp()*, the behavior is
2080 undefined.

2081 The behavior of *call_once()* is undefined if *flag* has automatic storage duration or is not
2082 initialized by `ONCE_FLAG_INIT`.

2083 The *call_once()* function shall not be affected if the calling thread executes a signal handler
2084 during the call.[/CX]

2085 RETURN VALUE

2086 The *call_once()* function shall not return a value.

2087 ERRORS

2088 No errors are defined.

2089 EXAMPLES

2090 None.

2091 APPLICATION USAGE

2092 If *init_routine* recursively calls *call_once()* with the same *flag*, the recursive call will not call
2093 the specified *init_routine*, and thus the specified *init_routine* will not complete, and thus the
2094 recursive call to *call_once()* will not return. Use of *longjmp()* or *siglongjmp()* within an
2095 *init_routine* to jump to a point outside of *init_routine* prevents *init_routine* from returning.

2096 RATIONALE

2097 For dynamic library initialization in a multi-threaded process, if an initialization flag is used
2098 the flag needs to be protected against modification by multiple threads simultaneously
2099 calling into the library. This can be done by using a statically-initialized mutex. However,
2100 the better solution is to use *call_once()* or *pthread_once()* which are designed for exactly
2101 this purpose, for example:

```
2102 #include <threads.h>  
2103 static once_flag random_is_initialized = ONCE_FLAG_INIT;  
2104 extern void initialize_random(void);
```

```
2105 int random_function()  
2106 {  
2107     call_once(&random_is_initialized, initialize_random);  
2108     ...  
2109     /* Operations performed after initialization. */  
2110 }
```

2111 The *call_once()* function is not affected by signal handlers for the reasons stated in [xref to
2112 XRAT B.2.3].

2113 FUTURE DIRECTIONS

2114 None.

2115 SEE ALSO

2116 *pthread_once*

- 2117 XBD Section 4.12.2, <**threads.h**>
- 2118 **CHANGE HISTORY**
- 2119 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.
- 2120 Ref 7.22.3 para 1
- 2121 On page 637 line 22002 section `calloc()`, change:
- 2122 a pointer to any type of object
- 2123 to:
- 2124 a pointer to any type of object with a fundamental alignment requirement
- 2125 Ref 7.22.3 para 2
- 2126 On page 637 line 22008 section `calloc()`, add a new paragraph:
- 2127 For purposes of determining the existence of a data race, `calloc()` shall behave as though it
- 2128 accessed only memory locations accessible through its arguments and not other static
- 2129 duration storage. The function may, however, visibly modify the storage that it allocates.
- 2130 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV]
- 2131 [CX]`reallocarray()`,[/CX] and `realloc()` that allocate or deallocate a particular region of
- 2132 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
- 2133 deallocation call shall synchronize with the next allocation (if any) in this order.
- 2134 Ref 7.22.3.1
- 2135 On page 637 line 22029 section `calloc()`, add `aligned_alloc` to the SEE ALSO section.
- 2136 Ref G.6 para 6, F.10.1.4, F.10 para 11
- 2137 On page 639 line 22055 section `carg()`, add:
- 2138 [MXC]If z is $-0 \pm i0$, $\pm\pi$ shall be returned.
- 2139 If z is $+0 \pm i0$, ± 0 shall be returned.
- 2140 If z is $x \pm i0$ where x is negative, $\pm\pi$ shall be returned.
- 2141 If z is $x \pm i0$ where x is positive, ± 0 shall be returned.
- 2142 If z is $\pm 0 + iy$ where y is negative, $-\pi/2$ shall be returned.
- 2143 If z is $\pm 0 + iy$ where y is positive, $\pi/2$ shall be returned.
- 2144 If z is $-\text{Inf} \pm iy$ where y is positive and finite, $\pm\pi$ shall be returned.
- 2145 If z is $+\text{Inf} \pm iy$ where y is positive and finite, ± 0 shall be returned.
- 2146 If z is $x \pm i\text{Inf}$ where x is finite, $\pm\pi/2$ shall be returned.
- 2147 If z is $-\text{Inf} \pm i\text{Inf}$, $\pm 3\pi/4$ shall be returned.

- 2148 If z is $+\text{Inf} \pm i\text{Inf}$, $\pm\pi/4$ shall be returned.
- 2149 If the real or imaginary part of z is NaN, NaN shall be returned.[/MXC]
- 2150 Ref G.6 para 7, G.6.2.2
- 2151 On page 640 line 22086 section `casin()`, add:
- 2152 [MXC]`casin(conj(iz))`, `casinf(conjf(iz))` and `casinl(conjl(iz))` shall return exactly the same
2153 value as `conj(casin(iz))`, `conjf(casinf(iz))` and `conjl(casinl(iz))`, respectively, and `casin(-iz)`,
2154 `casinf(-iz)` and `casinl(-iz)` shall return exactly the same value as `-casin(iz)`, `-casinf(iz)` and
2155 `-casinl(iz)`, respectively, including for the special values of iz below.
- 2156 If iz is $+0 + i0$, $-i(0 + i0)$ shall be returned.
- 2157 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i(+\text{Inf} + i\pi/2)$ shall be returned.
- 2158 If iz is $x + i\text{NaN}$ where x is finite, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the invalid
2159 floating-point exception may be raised.
- 2160 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i(+\text{Inf} + i0)$ shall be returned.
- 2161 If iz is $+\text{Inf} + i\text{Inf}$, $-i(+\text{Inf} + i\pi/4)$ shall be returned.
- 2162 If iz is $+\text{Inf} + i\text{NaN}$, $-i(+\text{Inf} + i\text{NaN})$ shall be returned.
- 2163 If iz is $\text{NaN} + i0$, $-i(\text{NaN} + i0)$ shall be returned.
- 2164 If iz is $\text{NaN} + iy$ where y is non-zero and finite, $-i(\text{NaN} + i\text{NaN})$ shall be returned and the
2165 invalid floating-point exception may be raised.
- 2166 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
2167 result is unspecified.
- 2168 If iz is $\text{NaN} + i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]
- 2169 Ref G.6 para 7
- 2170 On page 640 line 22094 section `casin()`, change RATIONALE from:
- 2171 None.
- 2172 to:
- 2173 The MXC special cases for `casin()` are derived from those for `casinh()` by applying the
2174 formula $\text{casin}(z) = -i \text{casinh}(iz)$.
- 2175 Ref G.6.2.2
- 2176 On page 641 line 22118 section `casinh()`, add:
- 2177 [MXC]`casinh(conj(z))`, `casinhf(conjf(z))` and `casinhl(conjl(z))` shall return exactly the same
2178 value as `conj(casinh(z))`, `conjf(casinhf(z))` and `conjl(casinhl(z))`, respectively, and `casinh(-z)`,
2179 `casinhf(-z)` and `casinhl(-z)` shall return exactly the same value as `-casinh(z)`, `-casinhf(z)`
2180 and `-casinhl(z)`, respectively, including for the special values of z below.

- 2181 If z is $+0 + i0$, $0 + i0$ shall be returned.
- 2182 If z is $x + i\text{Inf}$ where x is positive-signed and finite, $+\text{Inf} + i\pi/2$ shall be returned.
- 2183 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2184 point exception may be raised.
- 2185 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.
- 2186 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.
- 2187 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2188 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.
- 2189 If z is $\text{NaN} + iy$ where y is non-zero and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2190 floating-point exception may be raised.
- 2191 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
2192 unspecified.
- 2193 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2194 Ref G.6 para 7, G.6.2.3
2195 On page 643 line 22157 section *catan*, add:
- 2196 [MXC]*catan(conj(iz))*, *catanf(conjf(iz))* and *catanl(conjl(iz))* shall return exactly the same
2197 value as *conj(catan(iz))*, *conjf(catanf(iz))* and *conjl(catanl(iz))*, respectively, and *catan(-iz)*,
2198 *catanf(-iz)* and *catanl(-iz)* shall return exactly the same value as $-\text{catan}(iz)$, $-\text{catanf}(iz)$ and
2199 $-\text{catanl}(iz)$, respectively, including for the special values of iz below.
- 2200 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.
- 2201 If iz is $+0 + i\text{NaN}$, $-i (+0 + i\text{NaN})$ shall be returned.
- 2202 If iz is $+1 + i0$, $-i (+\text{Inf} + i0)$ shall be returned and the divide-by-zero floating-point
2203 exception shall be raised.
- 2204 If iz is $x + i\text{Inf}$ where x is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2205 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
2206 invalid floating-point exception may be raised.
- 2207 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i (+0 + i\pi/2)$ shall be returned.
- 2208 If iz is $+\text{Inf} + i\text{Inf}$, $-i (+0 + i\pi/2)$ shall be returned.
- 2209 If iz is $+\text{Inf} + i\text{NaN}$, $-i (+0 + i\text{NaN})$ shall be returned.
- 2210 If iz is $\text{NaN} + iy$ where y is finite, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the invalid
2211 floating-point exception may be raised.

2212 If iz is NaN + $i\text{Inf}$, $-i(\pm 0 + i\pi/2)$ shall be returned; the sign of the imaginary part of the
2213 result is unspecified.

2214 If iz is NaN + $i\text{NaN}$, $-i(\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2215 Ref G.6 para 7
2216 On page 643 line 22165 section `catan()`, change RATIONALE from:

2217 None.

2218 to:

2219 The MXC special cases for `catan()` are derived from those for `catanh()` by applying the
2220 formula $\text{catan}(z) = -i \text{catanh}(iz)$.

2221 Ref G.6.2.3
2222 On page 644 line 22189 section `catanh`, add:

2223 [MXC]`catanh(conj(z))`, `catanhf(conjf(z))` and `catanhl(conj(z))` shall return exactly the same
2224 value as `conj(catanh(z))`, `conjf(catanhf(z))` and `conjl(catanhl(z))`, respectively, and
2225 `catanh(-z)`, `catanhf(-z)` and `catanhl(-z)` shall return exactly the same value as $-\text{catanh}(z)$,
2226 $-\text{catanhf}(z)$ and $-\text{catanhl}(z)$, respectively, including for the special values of z below.

2227 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2228 If z is $+0 + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.

2229 If z is $+1 + i0$, $+\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2230 shall be raised.

2231 If z is $x + i\text{Inf}$ where x is positive-signed and finite, $+0 + i\pi/2$ shall be returned.

2232 If z is $x + i\text{NaN}$ where x is non-zero and finite, NaN + $i\text{NaN}$ shall be returned and the invalid
2233 floating-point exception may be raised.

2234 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\pi/2$ shall be returned.

2235 If z is $+\text{Inf} + i\text{Inf}$, $+0 + i\pi/2$ shall be returned.

2236 If z is $+\text{Inf} + i\text{NaN}$, $+0 + i\text{NaN}$ shall be returned.

2237 If z is NaN + iy where y is finite, NaN + $i\text{NaN}$ shall be returned and the invalid floating-
2238 point exception may be raised.

2239 If z is NaN + $i\text{Inf}$, $\pm 0 + i\pi/2$ shall be returned; the sign of the real part of the result is
2240 unspecified.

2241 If z is NaN + $i\text{NaN}$, NaN + $i\text{NaN}$ shall be returned.[/MXC]

2242 Ref G.6 para 7, G.6.2.4
2243 On page 652 line 22426 section `ccos()`, add:

2244 [MXC] $\text{ccos}(\text{conj}(iz))$, $\text{ccosf}(\text{conjf}(iz))$ and $\text{ccosl}(\text{conjl}(iz))$ shall return exactly the same value
2245 as $\text{conj}(\text{ccos}(iz))$, $\text{conjf}(\text{ccosf}(iz))$ and $\text{conjl}(\text{ccosl}(iz))$, respectively, and $\text{ccos}(-iz)$, $\text{ccosf}(-iz)$
2246 and $\text{ccosl}(-iz)$ shall return exactly the same value as $\text{ccos}(iz)$, $\text{ccosf}(iz)$ and $\text{ccosl}(iz)$,
2247 respectively, including for the special values of iz below.

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

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

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

2253 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
2254 floating-point exception shall be raised.

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

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

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

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

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

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

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

2266 If iz is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2267 Ref G.6 para 7
2268 On page 652 line 22434 section $\text{ccos}()$, change RATIONALE from:

2269 None.

2270 to:

2271 The MXC special cases for $\text{ccos}()$ are derived from those for $\text{ccosh}()$ by applying the
2272 formula $\text{ccos}(z) = \text{ccosh}(iz)$.

2273 Ref G.6.2.4
2274 On page 653 line 22455 section $\text{ccosh}()$, add:

2275 [MXC] $\text{ccosh}(\text{conj}(z))$, $\text{ccoshf}(\text{conjf}(z))$ and $\text{ccoshl}(\text{conjl}(z))$ shall return exactly the same
2276 value as $\text{conj}(\text{ccosh}(z))$, $\text{conjf}(\text{ccoshf}(z))$ and $\text{conjl}(\text{ccoshl}(z))$, respectively, and $\text{ccosh}(-z)$,

- 2277 $ccoshf(-z)$ and $ccoshl(-z)$ shall return exactly the same value as $ccosh(z)$, $ccoshf(z)$ and
2278 $ccoshl(z)$, respectively, including for the special values of z below.
- 2279 If z is $+0 + i0$, $1 + i0$ shall be returned.
- 2280 If z is $+0 + i\text{Inf}$, $\text{NaN} \pm i0$ shall be returned and the invalid floating-point exception shall be
2281 raised; the sign of the imaginary part of the result is unspecified.
- 2282 If z is $+0 + i\text{NaN}$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2283 unspecified.
- 2284 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
2285 floating-point exception shall be raised.
- 2286 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
2287 floating-point exception may be raised.
- 2288 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.
- 2289 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.
- 2290 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2291 shall be raised; the sign of the real part of the result is unspecified.
- 2292 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.
- 2293 If z is $\text{NaN} + i0$, $\text{NaN} \pm i0$ shall be returned; the sign of the imaginary part of the result is
2294 unspecified.
- 2295 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2296 invalid floating-point exception may be raised.
- 2297 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]
- 2298 Ref F.10.6.1 para 4
2299 On page 655 line 22489 section `ceil()`, add a new paragraph:
- 2300 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2301 arguments.[/MX]
- 2302 Ref F.10.6.1 para 2
2303 On page 655 line 22491 section `ceil()`, change:
- 2304 [MX]The result shall have the same sign as x .[/MX]
- 2305 to:
- 2306 [MX]The returned value shall be independent of the current rounding direction mode and
2307 shall have the same sign as x .[/MX]
- 2308 Ref F.10.6.1 para 4
2309 On page 655 line 22504 section `ceil()`, delete from APPLICATION USAGE:

2310 These functions may raise the inexact floating-point exception if the result differs in value
2311 from the argument.

2312 Ref G.6.3.1
2313 On page 657 line 22539 section `cexp()`, add:

2314 [MXC]`cexp(conj(z))`, `cexpf(conjf(z))` and `cexpl(conjl(z))` shall return exactly the same value
2315 as `conj(cexp(z))`, `conjf(cexpf(z))` and `conjl(cexpl(z))`, respectively, including for the special
2316 values of z below.

2317 If z is $\pm 0 + i0$, $1 + i0$ shall be returned.

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

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 $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2323 If z is $-\text{Inf} + iy$ where y is finite, $+0 (\cos(y) + i\sin(y))$ shall be returned.

2324 If z is $+\text{Inf} + iy$ where y is non-zero and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2325 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
2326 result are unspecified.

2327 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2328 shall be raised; the sign of the real part of the result is unspecified.

2329 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
2330 result are unspecified.

2331 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
2332 unspecified.

2333 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2334 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2335 invalid floating-point exception may be raised.

2336 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2337 Ref 7.26.5.7
2338 On page 679 line 23268 section `clock_getres()`, change:

2339 including the `nanosleep()` function

2340 to:

2341 including the `nanosleep()` and `thrd_sleep()` functions

2342 Ref G.6.3.2
2343 On page 687 line 23495 section `clog()`, add:

2344 `[MXC]``clog(conj(z))`, `clogf(conjf(z))` and `clogl(conjl(z))` shall return exactly the same value as
2345 `conj(clog(z))`, `conjf(clogf(z))` and `conjl(clogl(z))`, respectively, including for the special
2346 values of z below.

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

2349 If z is $+0 + i0$, $-\text{Inf} + i0$ shall be returned and the divide-by-zero floating-point exception
2350 shall be raised.

2351 If z is $x + i\text{Inf}$ where x is finite, $+\text{Inf} + i\pi/2$ shall be returned.

2352 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2353 point exception may be raised.

2354 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i\pi$ shall be returned.

2355 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2356 If z is $-\text{Inf} + i\text{Inf}$, $+\text{Inf} + i3\pi/4$ shall be returned.

2357 If z is $+\text{Inf} + i\text{Inf}$, $+\text{Inf} + i\pi/4$ shall be returned.

2358 If z is $\pm\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2359 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2360 point exception may be raised.

2361 If z is $\text{NaN} + i\text{Inf}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2362 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.`[/MXC]`

2363 Ref 7.26.3
2364 On page 698 line 23854 insert the following new `cnd_*`() sections:

2365 [Note to reviewers: changes to `cnd_broadcast` and `cnd_signal` may be needed depending on the](#)
2366 [outcome of Mantis bug 609.](#)

2367 **NAME**
2368 `cnd_broadcast`, `cnd_signal` — broadcast or signal a condition

2369 **SYNOPSIS**
2370 `#include <threads.h>`

2371 `int cnd_broadcast(cnd_t *cond);`
2372 `int cnd_signal(cnd_t *cond);`

2373 **DESCRIPTION**
2374 `[CX]` The functionality described on this reference page is aligned with the ISO C standard.

2375 Any conflict between the requirements described here and the ISO C standard is
2376 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2377 The *cond_broadcast()* function shall unblock all of the threads that are blocked on the
2378 condition variable pointed to by *cond* at the time of the call.

2379 The *cond_signal()* function shall unblock one of the threads that are blocked on the condition
2380 variable pointed to by *cond* at the time of the call (if any threads are blocked on *cond*).

2381 If no threads are blocked on the condition variable pointed to by *cond* at the time of the call,
2382 these functions shall have no effect and shall return *thrd_success*.

2383 [CX]If more than one thread is blocked on a condition variable, the scheduling policy shall
2384 determine the order in which threads are unblocked. When each thread unblocked as a result
2385 of a *cond_broadcast()* or *cond_signal()* returns from its call to *cond_wait()* or *cond_timedwait()*,
2386 the thread shall own the mutex with which it called *cond_wait()* or *cond_timedwait()*. The
2387 thread(s) that are unblocked shall contend for the mutex according to the scheduling policy
2388 (if applicable), and as if each had called *mtx_lock()*.

2389 The *cond_broadcast()* and *cond_signal()* functions can be called by a thread whether or not it
2390 currently owns the mutex that threads calling *cond_wait()* or *cond_timedwait()* have associated
2391 with the condition variable during their waits; however, if predictable scheduling behavior is
2392 required, then that mutex shall be locked by the thread calling *cond_broadcast()* or
2393 *cond_signal()*.

2394 These functions shall not be affected if the calling thread executes a signal handler during
2395 the call.[/CX]

2396 The behavior is undefined if the value specified by the *cond* argument to *cond_broadcast()* or
2397 *cond_signal()* does not refer to an initialized condition variable.

2398 **RETURN VALUE**

2399 These functions shall return *thrd_success* on success, or *thrd_error* if the request
2400 could not be honored.

2401 **ERRORS**

2402 No errors are defined.

2403 **EXAMPLES**

2404 None.

2405 **APPLICATION USAGE**

2406 See the APPLICATION USAGE section for *pthread_cond_broadcast()*, substituting
2407 *cond_broadcast()* for *pthread_cond_broadcast()* and *cond_signal()* for *pthread_cond_signal()*.

2408 **RATIONALE**

2409 As for *pthread_cond_broadcast()* and *pthread_cond_signal()*, spurious wakeups may occur
2410 with *cond_broadcast()* and *cond_signal()*, necessitating that applications code a predicate-
2411 testing-loop around the condition wait. (See the RATIONALE section for
2412 *pthread_cond_broadcast()*.)

2413 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT

2414 B.2.3].

2415 **FUTURE DIRECTIONS**

2416 None.

2417 **SEE ALSO**

2418 *cnd_destroy*, *cnd_timedwait*, *pthread_cond_broadcast*

2419 XBD Section 4.12.2, <**threads.h**>

2420 **CHANGE HISTORY**

2421 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2422 **NAME**

2423 *cnd_destroy*, *cnd_init* — destroy and initialize condition variables

2424 **SYNOPSIS**

2425 `#include <threads.h>`

2426 `void cnd_destroy(cnd_t *cond);`

2427 `int cnd_init(cnd_t *cond);`

2428 **DESCRIPTION**

2429 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2430 Any conflict between the requirements described here and the ISO C standard is
2431 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2432 The *cnd_destroy()* function shall release all resources used by the condition variable pointed
2433 to by *cond*. It shall be safe to destroy an initialized condition variable upon which no threads
2434 are currently blocked. Attempting to destroy a condition variable upon which other threads
2435 are currently blocked results in undefined behavior. A destroyed condition variable object
2436 can be reinitialized using *cnd_init()*; the results of otherwise referencing the object after it
2437 has been destroyed are undefined. The behavior is undefined if the value specified by the
2438 *cond* argument to *cnd_destroy()* does not refer to an initialized condition variable.

2439 The *cnd_init()* function shall initialize a condition variable. If it succeeds it shall set the
2440 variable pointed to by *cond* to a value that uniquely identifies the newly initialized condition
2441 variable. Attempting to initialize an already initialized condition variable results in
2442 undefined behavior. A thread that calls *cnd_wait()* on a newly initialized condition variable
2443 shall block.

2444 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
2445 further requirements.

2446 These functions shall not be affected if the calling thread executes a signal handler during
2447 the call.[/CX]

2448 **RETURN VALUE**

2449 The *cnd_destroy()* function shall not return a value.

2450 The *cnd_init()* function shall return *thrd_success* on success, or *thrd_nomem* if no
2451 memory could be allocated for the newly created condition, or *thrd_error* if the request

2452 could not be honored.

2453 **ERRORS**

2454 See RETURN VALUE.

2455 **EXAMPLES**

2456 None.

2457 **APPLICATION USAGE**

2458 None.

2459 **RATIONALE**

2460 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
2461 B.2.3].

2462 **FUTURE DIRECTIONS**

2463 None.

2464 **SEE ALSO**

2465 *cond_broadcast, cond_timedwait*

2466 XBD <**threads.h**>

2467 **CHANGE HISTORY**

2468 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2469 **NAME**

2470 *cond_timedwait, cond_wait* — wait on a condition

2471 **SYNOPSIS**

```
2472 #include <threads.h>  
2473 int cond_timedwait(cond_t * restrict cond, mtx_t * restrict mtx,  
2474                   const struct timespec * restrict ts);  
2475 int cond_wait(cond_t *cond, mtx_t *mtx);
```

2476 **DESCRIPTION**

2477 [CX] The functionality described on this reference page is aligned with the ISO C standard.
2478 Any conflict between the requirements described here and the ISO C standard is
2479 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2480 The *cond_timedwait()* function shall atomically unlock the mutex pointed to by *mtx* and block
2481 until the condition variable pointed to by *cond* is signaled by a call to *cond_signal()* or to
2482 *cond_broadcast()*, or until after the TIME_UTC-based calendar time pointed to by *ts*, or until
2483 it is unblocked due to an unspecified reason.

2484 The *cond_wait()* function shall atomically unlock the mutex pointed to by *mtx* and block until
2485 the condition variable pointed to by *cond* is signaled by a call to *cond_signal()* or to
2486 *cond_broadcast()*, or until it is unblocked due to an unspecified reason.

2487 [CX]Atomically here means "atomically with respect to access by another thread to the
2488 mutex and then the condition variable". That is, if another thread is able to acquire the mutex
2489 after the about-to-block thread has released it, then a subsequent call to *cond_broadcast()* or

2490 *cond_signal()* in that thread shall behave as if it were issued after the about-to-block thread
2491 has blocked.[/CX]

2492 When the calling thread becomes unblocked, these functions shall lock the mutex pointed to
2493 by *mtx* before they return. The application shall ensure that the mutex pointed to by *mtx* is
2494 locked by the calling thread before it calls these functions.

2495 When using condition variables there is always a Boolean predicate involving shared
2496 variables associated with each condition wait that is true if the thread should proceed.
2497 Spurious wakeups from the *cond_timedwait()* and *cond_wait()* functions may occur. Since the
2498 return from *cond_timedwait()* or *cond_wait()* does not imply anything about the value of this
2499 predicate, the predicate should be re-evaluated upon such return.

2500 When a thread waits on a condition variable, having specified a particular mutex to either
2501 the *cond_timedwait()* or the *cond_wait()* operation, a dynamic binding is formed between that
2502 mutex and condition variable that remains in effect as long as at least one thread is blocked
2503 on the condition variable. During this time, the effect of an attempt by any thread to wait on
2504 that condition variable using a different mutex is undefined. Once all waiting threads have
2505 been unblocked (as by the *cond_broadcast()* operation), the next wait operation on
2506 that condition variable shall form a new dynamic binding with the mutex specified by that
2507 wait operation. Even though the dynamic binding between condition variable and mutex
2508 might be removed or replaced between the time a thread is unblocked from a wait on the
2509 condition variable and the time that it returns to the caller or begins cancellation cleanup, the
2510 unblocked thread shall always re-acquire the mutex specified in the condition wait operation
2511 call from which it is returning.

2512 [CX]A condition wait (whether timed or not) is a cancellation point. When the cancelability
2513 type of a thread is set to `PTHREAD_CANCEL_DEFERRED`, a side-effect of acting upon a
2514 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired
2515 before calling the first cancellation cleanup handler. The effect is as if the thread were
2516 unblocked, allowed to execute up to the point of returning from the call to *cond_timedwait()*
2517 or *cond_wait()*, but at that point notices the cancellation request and instead of returning to
2518 the caller of *cond_timedwait()* or *cond_wait()*, starts the thread cancellation activities, which
2519 includes calling cancellation cleanup handlers.

2520 A thread that has been unblocked because it has been canceled while blocked in a call to
2521 *cond_timedwait()* or *cond_wait()* shall not consume any condition signal that may be directed
2522 concurrently at the condition variable if there are other threads blocked on the condition
2523 variable.[/CX]

2524 When *cond_timedwait()* times out, it shall nonetheless release and re-acquire the mutex
2525 referenced by *mutex*, and may consume a condition signal directed concurrently at the
2526 condition variable.

2527 [CX]These functions shall not be affected if the calling thread executes a signal handler
2528 during the call, except that if a signal is delivered to a thread waiting for a condition
2529 variable, upon return from the signal handler either the thread shall resume waiting for the
2530 condition variable as if it was not interrupted, or it shall return `thrd_success` due to
2531 spurious wakeup.[/CX]

2532 The behavior is undefined if the value specified by the *cond* or *mtx* argument to these
2533 functions does not refer to an initialized condition variable or an initialized mutex object,

2534 respectively.

2535 **RETURN VALUE**

2536 The *cond_timedwait()* function shall return *thrd_success* upon success, or

2537 *thrd_timedout* if the time specified in the call was reached without acquiring the

2538 requested resource, or *thrd_error* if the request could not be honored.

2539 The *cond_wait()* function shall return *thrd_success* upon success or *thrd_error* if the

2540 request could not be honored.

2541 **ERRORS**

2542 See RETURN VALUE.

2543 **EXAMPLES**

2544 None.

2545 **APPLICATION USAGE**

2546 None.

2547 **RATIONALE**

2548 These functions are not affected by signal handlers (except as stated in the DESCRIPTION)

2549 for the reasons stated in [xref to XRAT B.2.3].

2550 **FUTURE DIRECTIONS**

2551 None.

2552 **SEE ALSO**

2553 *cond_broadcast*, *cond_destroy*, *timespec_get*

2554 XBD Section 4.12.2, <**threads.h**>

2555 **CHANGE HISTORY**

2556 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2557 Ref F.10.8.1 para 2

2558 On page 705 line 24155 section *copysign()*, add a new paragraph:

2559 [MX]The returned value shall be exact and shall be independent of the current rounding

2560 direction mode.[/MX]

2561 Ref G.6.4.1 para 1

2562 On page 711 line 24308 section *cpow()*, add a new paragraph:

2563 [MXC]These functions shall raise floating-point exceptions if appropriate for the calculation

2564 of the parts of the result, and may also raise spurious floating-point exceptions.[/MXC]

2565 Ref G.6.4.1 footnote 386

2566 On page 711 line 24318 section *cpow()*, change RATIONALE from:

2567 None.

2568 to:

2569 Permitting spurious floating-point exceptions allows $cpow(z, c)$ to be implemented as $cexp(c$
2570 $clog(z))$ without precluding implementations that treat special cases more carefully.

2571 Ref G.6 para 7, G.6.2.5

2572 On page 718 line 24545 section $csin()$, add:

2573 [MXC] $csin(conj(iz))$, $csinf(conjf(iz))$ and $csinl(conjl(iz))$ shall return exactly the same value
2574 as $conj(csin(iz))$, $conjf(csinf(iz))$ and $conjl(csinl(iz))$, respectively, and $csin(-iz)$, $csinf(-iz)$
2575 and $csinl(-iz)$ shall return exactly the same value as $-csin(iz)$, $-csinf(iz)$ and $-csinl(iz)$,
2576 respectively, including for the special values of iz below.

2577 If iz is $+0 + i0$, $-i(+0 + i0)$ shall be returned.

2578 If iz is $+0 + iInf$, $-i(\pm 0 + iNaN)$ shall be returned and the invalid floating-point exception
2579 shall be raised; the sign of the imaginary part of the result is unspecified.

2580 If iz is $+0 + iNaN$, $-i(\pm 0 + iNaN)$ shall be returned; the sign of the imaginary part of the
2581 result is unspecified.

2582 If iz is $x + iInf$ where x is positive and finite, $-i(NaN + iNaN)$ shall be returned and the
2583 invalid floating-point exception shall be raised.

2584 If iz is $x + iNaN$ where x is non-zero and finite, $-i(NaN + iNaN)$ shall be returned and the
2585 invalid floating-point exception may be raised.

2586 If iz is $+Inf + i0$, $-i(+Inf + i0)$ shall be returned.

2587 If iz is $+Inf + iy$ where y is positive and finite, $-iInf(\cos(y) + isin(y))$ shall be returned.

2588 If iz is $+Inf + iInf$, $-i(\pm Inf + iNaN)$ shall be returned and the invalid floating-point exception
2589 shall be raised; the sign of the imaginary part of the result is unspecified.

2590 If iz is $+Inf + iNaN$, $-i(\pm Inf + iNaN)$ shall be returned; the sign of the imaginary part of the
2591 result is unspecified.

2592 If iz is $NaN + i0$, $-i(NaN + i0)$ shall be returned.

2593 If iz is $NaN + iy$ where y is any non-zero number, $-i(NaN + iNaN)$ shall be returned and the
2594 invalid floating-point exception may be raised.

2595 If iz is $NaN + iNaN$, $-i(NaN + iNaN)$ shall be returned.[/MXC]

2596 Ref G.6 para 7

2597 On page 718 line 24553 section $csin()$, change RATIONALE from:

2598 None.

2599 to:

2600 The MXC special cases for $csin()$ are derived from those for $csinh()$ by applying the formula

2601 $\text{csin}(z) = -i \text{csinh}(iz)$.

2602 Ref G.6.2.5

2603 On page 719 line 24574 section `csinh()`, add:

2604 [MXC]`csinh(conj(z))`, `csinhf(conjf(z))` and `csinhl(conjl(z))` shall return exactly the same
2605 value as `conj(csinh(z))`, `conjf(csinhf(z))` and `conjl(csinhl(z))`, respectively, and `csinh(-z)`,
2606 `csinhf(-z)` and `csinhl(-z)` shall return exactly the same value as `-csinh(z)`, `-csinhf(z)` and
2607 `-csinhl(z)`, respectively, including for the special values of z below.

2608 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2609 If z is $+0 + i\text{Inf}$, $\pm 0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2610 raised; the sign of the real part of the result is unspecified.

2611 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
2612 unspecified.

2613 If z is $x + i\text{Inf}$ where x is positive and finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid
2614 floating-point exception shall be raised.

2615 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
2616 floating-point exception may be raised.

2617 If z is $+\text{Inf} + i0$, $+\text{Inf} + i0$ shall be returned.

2618 If z is $+\text{Inf} + iy$ where y is positive and finite, $+\text{Inf} (\cos(y) + i\sin(y))$ shall be returned.

2619 If z is $+\text{Inf} + i\text{Inf}$, $\pm\text{Inf} + i\text{NaN}$ shall be returned and the invalid floating-point exception
2620 shall be raised; the sign of the real part of the result is unspecified.

2621 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
2622 unspecified.

2623 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2624 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2625 invalid floating-point exception may be raised.

2626 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2627 Ref G.6.4.2

2628 On page 721 line 24612 section `csqrt()`, add:

2629 [MXC]`csqrt(conj(z))`, `csqrtf(conjf(z))` and `csqrtl(conjl(z))` shall return exactly the same value
2630 as `conj(csqrt(z))`, `conjf(csqrtf(z))` and `conjl(csqrtl(z))`, respectively, including for the special
2631 values of z below.

2632 If z is $\pm 0 + i0$, $+0 + i0$ shall be returned.

2633 If the imaginary part of z is Inf , $+\text{Inf} + i\text{Inf}$, shall be returned.

2634 If z is $x + i\text{NaN}$ where x is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2635 point exception may be raised.

2636 If z is $-\text{Inf} + iy$ where y is positive-signed and finite, $+0 + i\text{Inf}$ shall be returned.

2637 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $+\text{Inf} + i0$ shall be returned.

2638 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
2639 is unspecified.

2640 If z is $+\text{Inf} + i\text{NaN}$, $+\text{Inf} + i\text{NaN}$ shall be returned.

2641 If z is $\text{NaN} + iy$ where y is finite, $\text{NaN} + i\text{NaN}$ shall be returned and the invalid floating-
2642 point exception may be raised.

2643 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2644 Ref G.6 para 7, G.6.2.6
2645 On page 722 line 24641 section `ctan()`, add:

2646 [MXC]`ctan(conj(iz))`, `ctanf(conjf(iz))` and `ctanl(conjl(iz))` shall return exactly the same value
2647 as `conj(ctan(iz))`, `conjf(ctanf(iz))` and `conjl(ctanl(iz))`, respectively, and `ctan(-iz)`, `ctanf(-iz)`
2648 and `ctanl(-iz)` shall return exactly the same value as `-ctan(iz)`, `-ctanf(iz)` and `-ctanl(iz)`,
2649 respectively, including for the special values of iz below.

2650 If iz is $+0 + i0$, $-i (+0 + i0)$ shall be returned.

2651 If iz is $0 + i\text{Inf}$, $-i (0 + i\text{NaN})$ shall be returned and the invalid floating-point exception shall
2652 be raised.

2653 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
2654 invalid floating-point exception shall be raised.

2655 If iz is $0 + i\text{NaN}$, $-i (0 + i\text{NaN})$ shall be returned.

2656 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
2657 invalid floating-point exception may be raised.

2658 If iz is $+\text{Inf} + iy$ where y is positive-signed and finite, $-i (1 + i0 \sin(2y))$ shall be returned.

2659 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
2660 unspecified.

2661 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
2662 unspecified.

2663 If iz is $\text{NaN} + i0$, $-i (\text{NaN} + i0)$ shall be returned.

2664 If iz is $\text{NaN} + iy$ where y is any non-zero number, $-i (\text{NaN} + i\text{NaN})$ shall be returned and the
2665 invalid floating-point exception may be raised.

2666 If iz is $\text{NaN} + i\text{NaN}$, $-i (\text{NaN} + i\text{NaN})$ shall be returned.[/MXC]

2667 Ref G.6 para 7
2668 On page 722 line 24649 section `ctan()`, change RATIONALE from:

2669 None.

2670 to:

2671 The MXC special cases for `ctan()` are derived from those for `ctanh()` by applying the
2672 formula $ctan(z) = -i ctanh(iz)$.

2673 Ref G.6.2.6
2674 On page 723 line 24670 section `ctanh()`, add:

2675 [MXC]`ctanh(conj(z))`, `ctanhf(conjf(z))` and `ctanhl(conjl(z))` shall return exactly the same
2676 value as `conj(ctanh(z))`, `conjf(ctanhf(z))` and `conjl(ctanhl(z))`, respectively, and `ctanh(-z)`,
2677 `ctanhf(-z)` and `ctanhl(-z)` shall return exactly the same value as $-ctanh(z)$, $-ctanhf(z)$ and
2678 $-ctanhl(z)$, respectively, including for the special values of z below.

2679 If z is $+0 + i0$, $+0 + i0$ shall be returned.

2680 If z is $0 + i\text{Inf}$, $0 + i\text{NaN}$ shall be returned and the invalid floating-point exception shall be
2681 raised.

2682 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
2683 floating-point exception shall be raised.

2684 If z is $0 + i\text{NaN}$, $0 + i\text{NaN}$ shall be returned.

2685 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
2686 floating-point exception may be raised.

2687 If z is $+\text{Inf} + iy$ where y is positive-signed and finite, $1 + i0 \sin(2y)$ shall be returned.

2688 If z is $+\text{Inf} + i\text{Inf}$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2689 unspecified.

2690 If z is $+\text{Inf} + i\text{NaN}$, $1 \pm i0$ shall be returned; the sign of the imaginary part of the result is
2691 unspecified.

2692 If z is $\text{NaN} + i0$, $\text{NaN} + i0$ shall be returned.

2693 If z is $\text{NaN} + iy$ where y is any non-zero number, $\text{NaN} + i\text{NaN}$ shall be returned and the
2694 invalid floating-point exception may be raised.

2695 If z is $\text{NaN} + i\text{NaN}$, $\text{NaN} + i\text{NaN}$ shall be returned.[/MXC]

2696 Ref 7.27.3, 7.1.4 para 5
2697 On page 727 line 24774 section `ctime()`, change:

2698 [CX]The `ctime()` function need not be thread-safe.[/CX]

2699 to:
2700 The *ctime()* function need not be thread-safe; however, *ctime()* shall avoid data races with all
2701 functions other than itself, *asctime()*, *gmtime()* and *localtime()*.

2702 Ref 7.5 para 2
2703 On page 781 line 26447 section *errno*, change:

2704 The lvalue *errno* is used by many functions to return error values.

2705 to:

2706 The lvalue to which the macro *errno* expands is used by many functions to return error
2707 values.

2708 Ref 7.5 para 3
2709 On page 781 line 26449 section *errno*, change:

2710 The value of *errno* shall be defined only after a call to a function for which it is explicitly
2711 stated to be set and until it is changed by the next function call or if the application assigns it
2712 a value.

2713 to:

2714 The value of *errno* in the initial thread shall be zero at program startup (the initial value of
2715 *errno* in other threads is an indeterminate value) and shall otherwise be defined only after a
2716 call to a function for which it is explicitly stated to be set and until it is changed by the next
2717 function call or if the application assigns it a value.

2718 Ref 7.5 para 2
2719 On page 781 line 26456 section *errno*, delete:

2720 It is unspecified whether *errno* is a macro or an identifier declared with external linkage.

2721 Ref 7.22.4.4 para 2
2722 On page 796 line 27057 section *exit()*, add a new (unshaded) paragraph:

2723 The *exit()* function shall cause normal process termination to occur. No functions registered
2724 by the *at_quick_exit()* function shall be called. If a process calls the *exit()* function more
2725 than once, or calls the *quick_exit()* function in addition to the *exit()* function, the behavior is
2726 undefined.

2727 Ref 7.22.4.4 para 2
2728 On page 796 line 27068 section *exit()*, delete:

2729 If *exit()* is called more than once, the behavior is undefined.

2730 Ref 7.22.4.3, 7.22.4.7
2731 On page 796 line 27086 section *exit()*, add *at_quick_exit* and *quick_exit* to the SEE ALSO section.

2732 Ref F.10.4.2 para 2
2733 On page 804 line 27323 section *fabs()*, add a new paragraph:

2734 [MX]The returned value shall be exact and shall be independent of the current rounding
2735 direction mode.[/MX]

2736 Ref 7.21.2 para 7,8
2737 On page 874 line 29483 section flockfile(), change:

2738 These functions shall provide for explicit application-level locking of stdio (**FILE ***)
2739 objects.

2740 to:

2741 These functions shall provide for explicit application-level locking of the locks associated
2742 with standard I/O streams (see [xref to 2.5]).

2743 Ref 7.21.2 para 7,8
2744 On page 874 line 29499 section flockfile(), delete:

2745 All functions that reference (**FILE ***) objects, except those with names ending in *_unlocked*,
2746 shall behave as if they use *flockfile()* and *funlockfile()* internally to obtain ownership of these
2747 (**FILE ***) objects.

2748 Ref F.10.6.2 para 3
2749 On page 876 line 29560 section floor(), add a new paragraph:

2750 [MX]These functions may raise the inexact floating-point exception for finite non-integer
2751 arguments.[/MX]

2752 Ref F.10.6.2 para 2
2753 On page 876 line 29562 section floor(), change:

2754 [MX]The result shall have the same sign as *x*.[/MX]

2755 to:

2756 [MX]The returned value shall be independent of the current rounding direction mode and
2757 shall have the same sign as *x*.[/MX]

2758 Ref F.10.6.2 para 3
2759 On page 876 line 29576 section floor(), delete from APPLICATION USAGE:

2760 These functions may raise the inexact floating-point exception if the result differs in value
2761 from the argument.

2762 Ref F.10.9.2 para 2
2763 On page 880 line 29695 section fmax(), add a new paragraph:

2764 [MX]The returned value shall be exact and shall be independent of the current rounding
2765 direction mode.[/MX]

2766 Ref F.10.9.3 para 2
2767 On page 884 line 29844 section fmin(), add a new paragraph:

2768 [MX]The returned value shall be exact and shall be independent of the current rounding
2769 direction mode.[/MX]

2770 Ref F.10.7.1 para 2

2771 On page 885 line 29892 section `fmod()`, change:

2772 [MXX]If the correct value would cause underflow, and is representable, a range error may
2773 occur and the correct value shall be returned.[/MXX]

2774 to:

2775 [MX]When subnormal results are supported, the returned value shall be exact and shall be
2776 independent of the current rounding direction mode.[/MX]

2777 Ref 7.21.5.3 para 5

2778 On page 892 line 30117 section `fopen()`, change:

2779 [CX]The functionality described on this reference page is aligned with the ISO C standard.
2780 Any conflict between the requirements described here and the ISO C standard is
2781 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2782 to:

2783 [CX]Except for the “exclusive access” requirement (see below), the functionality described
2784 on this reference page is aligned with the ISO C standard. Any other conflict between the
2785 requirements described here and the ISO C standard is unintentional. This volume of
2786 POSIX.1-202x defers to the ISO C standard for all *fopen()* functionality except in relation to
2787 “exclusive access”.[/CX]

2788 Ref 7.21.5.3 para 5

2789 On page 892 line 30132 section `fopen()`, after applying bug 411, change:

2790 'x' If specified with a prefix beginning with 'w' [CX]or 'a'[/CX], then the function shall
2791 fail if the file already exists, [CX]as if by the `O_EXCL` flag to *open()*. If specified
2792 with a prefix beginning with 'r', this modifier shall have no effect.[/CX]

2793 to:

2794 'x' If specified with a prefix beginning with 'w' [CX]or 'a'[/CX], then the function shall
2795 fail if the file already exists or cannot be created; if the file does not exist and can be
2796 created, it shall be created with [CX]an implementation-defined form of[/CX]
2797 exclusive (also known as non-shared) access, [CX]if supported by the underlying file
2798 system, provided the resulting file permissions are the same as they would be without
2799 the 'x' modifier. If specified with a prefix beginning with 'r', this modifier shall have
2800 no effect.[/CX]

2801 **Note:** The ISO C standard requires exclusive access “to the extent that the underlying file
2802 system supports exclusive access”, but does not define what it means by this. Taken
2803 at face value—that systems must do whatever they are capable of, at the file system
2804 level, in order to exclude access by others—this would require POSIX.1 systems to
2805 set the file permissions in a way that prevents access by other users and groups.
2806 Consequently, this volume of POSIX.1-202x does not defer to the ISO C standard as

2807 regards the “exclusive access” requirement.

2808 [Note to reviewers: This “exclusive access” requirement may be clarified in C2x, in which case the](#)
2809 [above text may be changed to match the proposed C2x text.](#)

2810 Ref 7.21.5.3 para 3
2811 On page 892 line 30144 section `fopen()`, change:

2812 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and ...

2813 to:

2814 If the first character in *mode* is *w* or *a*, and ...

2815 Ref 7.21.5.3 para 3,5
2816 On page 892 line 30148 section `fopen()`, change:

2817 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously
2818 exist, the `fopen()` function shall create a file as if it called the `creat()` function with a value
2819 appropriate for the *path* argument interpreted from *pathname* and a value of `S_IRUSR` |
2820 `S_IWUSR` | `S_IRGRP` | `S_IWGRP` | `S_IROTH` | `S_IWOTH` for the *mode* argument.

2821 to:

2822 If the first character in *mode* is *w* or *a*, and the file did not previously exist, the `fopen()`
2823 function shall create a file as if it called the `open()` function with a value appropriate for the
2824 *path* argument interpreted from *pathname*, a value for the *oflag* argument as specified below,
2825 and a value of `S_IRUSR` | `S_IWUSR` | `S_IRGRP` | `S_IWGRP` | `S_IROTH` | `S_IWOTH` for
2826 the third argument.

2827 Ref 7.21.5.3 para 5
2828 On page 893 line 30158 section `fopen()`, change:

2829 The file descriptor ...

2830 to:

2831 If the first character in *mode* is *r*, or the suffix of *mode* does not include *x*, the file descriptor
2832 ...

2833 Ref (none; see bug 411)
2834 On page 893 line 30160 section `fopen()`, change the first column heading from:

2835 ***fopen()* Mode**

2836 to:

2837 ***fopen()* Mode Without Suffix**

2838 and add the following text after the table:

2839 with the addition of the `O_CLOEXEC` flag if the suffix of *mode* includes *e*.

2840 Ref 7.21.5.3 para 5
2841 On page 893 line 30166 section `fopen()`, add the following new paragraphs:

2842 [CX]If the first character in *mode* is *w* or *a*, the suffix of *mode* includes *x*, and the underlying
2843 file system does not support exclusive access, then the file descriptor associated with the
2844 opened stream shall be allocated and opened as if by a call to *open()* with the following
2845 flags:

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

2846 with the addition of the O_CLOEXEC flag if the suffix of *mode* includes *e*.

2847 If the first character in *mode* is *w* or *a*, the suffix of *mode* includes *x*, and the underlying file
2848 system supports exclusive access, then the file descriptor associated with the opened stream
2849 shall be allocated and opened as if by a call to *open()* with the above flags or with the above
2850 flags ORed with an implementation-defined file creation flag if necessary to enable
2851 exclusive access (see above).[/CX]

2852 [Note to reviewers: The above change may need to be updated depending on whether WG14 clarify](#)
2853 [the “exclusive access” requirement.](#)

2854 Ref 7.21.5.3 para 5
2855 On page 895 line 30236 section `fopen()`, change APPLICATION USAGE from:

2856 None.

2857 to:

2858 If an application needs to create a file in a way that fails if the file already exists, and either
2859 requires that it does not have exclusive access to the file or does not need exclusive access, it
2860 should use *open()* with the O_CREAT and O_EXCL flags instead of using *fopen()* with an *x*
2861 in the *mode*. A stream can then be created, if needed, by calling *fdopen()* on the file
2862 descriptor returned by *open()*.

2863 [Note to reviewers: The above change may need to be updated depending on whether WG14 clarify](#)
2864 [the “exclusive access” requirement.](#)

2865 Ref 7.21.5.3 para 5
2866 On page 895 line 30238 section `fopen()`, after applying bug 411, change:

2867 The *x* mode suffix character was added by C1x only for files opened with a mode string
2868 beginning with *w*.

2869 to:

2870 The *x* mode suffix character is specified by the ISO C standard only for files opened with a
2871 mode string beginning with *w*.

2872 and then add two new paragraphs after the one that starts with the above text:

2873 When the last character in *mode* is *x*, the ISO C standard requires that the file is created with
2874 exclusive access to the extent that the underlying system supports exclusive access.
2875 Although POSIX.1 does not specify any method of enabling exclusive access, it allows for
2876 the existence of an implementation-defined file creation flag that enables it. Note that it must
2877 be a file creation flag, not a file access mode flag (that is, one that is included in
2878 O_ACCMODE) or a file status flag, so that it does not affect the value returned by *fcntl()*
2879 with F_GETFL. On implementations that have such a flag, if support for it is file system
2880 dependent and exclusive access is requested when using *fopen()* to create a file on a file
2881 system that does not support it, the flag must not be used if it would cause *fopen()* to fail.

2882 Some implementations support mandatory file locking as a means of enabling exclusive
2883 access to a file. Locks are set in the normal way, but instead of only preventing others from
2884 setting conflicting locks they prevent others from accessing the contents of the locked part
2885 of the file in a way that conflicts with the lock. However, unless the implementation has a
2886 way of setting a whole-file write lock on file creation, this does not satisfy the requirement
2887 in the ISO C standard that the file is “created with exclusive access to the extent that the
2888 underlying system supports exclusive access”. (Having *fopen()* create the file and set a lock
2889 on the file as two separate operations is not the same, and it would introduce a race
2890 condition whereby another process could open the file and write to it (or set a lock) in
2891 between the two operations.) However, on all implementations that support mandatory file
2892 locking, its use is discouraged; therefore, it is recommended that implementations which
2893 support mandatory file locking do **not** add a means of creating a file with a whole-file
2894 exclusive lock set, so that *fopen()* is not required to enable mandatory file locking in order to
2895 conform to the ISO C standard. Note also that, since mandatory file locking is enabled via a
2896 file permissions change, the requirement that the ‘*x*’ modifier does not alter the permissions
2897 means that this standard does not allow mandatory file locking to be enabled. An
2898 implementation that has a means of creating a file with a whole-file exclusive lock set would
2899 need to provide a way to change the behavior of *fopen()* depending on whether the calling
2900 process is executing in a POSIX.1 conforming environment or an ISO C conforming
2901 environment.

2902 [Note to reviewers: The above change may need to be updated depending on whether WG14 clarify](#)
2903 [the “exclusive access” requirement.](#)

2904 Ref 7.22.3.3 para 2

2905 On page 933 line 31673 section *free()*, after applying bug 1218 change:

2906 Otherwise, if the argument does not match a pointer earlier returned by a function in
2907 POSIX.1-2017 that allocates memory as if by *malloc()*, or if the space has been deallocated
2908 by a call to *free()*, *realloc()*, [CX]or *reallocarray()*,[/CX] the behavior is undefined.

2909 to:

2910 Otherwise, if the argument does not match a pointer earlier returned by *aligned_alloc()*,
2911 *calloc()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] *realloc()*, [CX]*reallocarray()*, or a
2912 function in POSIX.1-20xx that allocates memory as if by *malloc()*,[/CX] or if the space has
2913 been deallocated by a call to *free()*, [CX]*reallocarray()*,[/CX] or *realloc()*, the behavior is

2914 undefined.

2915 Ref 7.22.3 para 2

2916 On page 933 line 31677 section `free()`, add a new paragraph:

2917 For purposes of determining the existence of a data race, `free()` shall behave as though it
2918 accessed only memory locations accessible through its argument and not other static
2919 duration storage. The function may, however, visibly modify the storage that it deallocates.
2920 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV]
2921 [CX]`reallocarray()`,[/CX] and `realloc()` that allocate or deallocate a particular region of
2922 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
2923 deallocation call shall synchronize with the next allocation (if any) in this order.

2924 Ref 7.22.3.1

2925 On page 933 line 31691 section `free()`, add `aligned_alloc` to the SEE ALSO section.

2926 Ref 7.21.5.3 para 5

2927 On page 942 line 31988 section `freopen()`, change:

2928 [CX]The functionality described on this reference page is aligned with the ISO C standard.
2929 Any conflict between the requirements described here and the ISO C standard is
2930 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2931 to:

2932 [CX]Except for the “exclusive access” requirement (see [xref to `fopen()`]), the functionality
2933 described on this reference page is aligned with the ISO C standard. Any other conflict
2934 between the requirements described here and the ISO C standard is unintentional. This
2935 volume of POSIX.1-202x defers to the ISO C standard for all `freopen()` functionality except
2936 in relation to “exclusive access”.[/CX]

2937 Ref 7.21.5.3 para 3,5; 7.21.5.4 para 2

2938 On page 942 line 32010 section `freopen()`, replace the following text:

2939 shall be allocated and opened as if by a call to `open()` with the following flags:

2940 and the table that follows it, and the paragraph added by bug 411 after the table, with:

2941 shall be allocated and opened as if by a call to `open()` with the flags specified for `fopen()`
2942 with the same *mode* argument.

2943 Ref (none)

2944 On page 944 line 32094 section `freopen()`, change:

2945 It is possible that these side-effects are an unintended consequence of the way the feature is
2946 specified in the ISO/IEC 9899: 1999 standard, but unless or until the ISO C standard is
2947 changed, ...

2948 to:

2949 It is possible that these side-effects are an unintended consequence of the way the feature
2950 was specified in the ISO/IEC 9899: 1999 standard (and still is in the current standard), but

2951 unless or until the ISO C standard is changed, ...

2952 [Note to reviewers: if the APPLICATION USAGE and RATIONALE additions for fopen\(\) are](#)
2953 [retained, changes should be added here to make the equivalent sections for freopen\(\) refer to those](#)
2954 [for fopen\(\)](#).

2955 Ref (none)
2956 On page 944 line 32102 section freopen(), after applying bug 411 change:

2957 The *x* mode suffix character was added by C1x only for files opened with a *mode* string
2958 beginning with *w*.

2959 to:

2960 The *x* mode suffix character is specified by the ISO C standard only for files opened with a
2961 mode string beginning with *w*.

2962 Ref 7.12.6.4 para 3
2963 On page 947 line 32161 section frexp(), change:

2964 The integer exponent shall be stored in the **int** object pointed to by *exp*.

2965 to:

2966 The integer exponent shall be stored in the **int** object pointed to by *exp*; if the integer
2967 exponent is outside the range of **int**, the results are unspecified.

2968 Ref F.10.3.4 para 3
2969 On page 947 line 32164 section frexp(), add a new paragraph:

2970 [MX]When the radix of the argument is a power of 2, the returned value shall be exact and
2971 shall be independent of the current rounding direction mode.[/MX]

2972 Ref 7.21.6.2 para 4
2973 On page 950 line 32239 section fscanf(), change:

2974 If a directive fails, as detailed below, the function shall return.

2975 to:

2976 When all directives have been executed, or if a directive fails (as detailed below), the
2977 function shall return.

2978 Ref 7.21.6.2 para 5
2979 On page 950 line 32242 section fscanf(), after applying bug 1163 change:

2980 A directive composed of one or more white-space bytes shall be executed by reading input
2981 until no more valid input can be read, or up to the first non-white-space byte , which remains
2982 unread.

2983 to:

2984 A directive composed of one or more white-space bytes shall be executed by reading input
2985 up to the first non-white-space byte, which shall remain unread, or until no more bytes can
2986 be read. The directive shall never fail.

2987 Ref (none)

2988 On page 955 line 32471 section `fscanf()`, change:

2989 This function is aligned with the ISO/IEC 9899: 1999 standard, and in doing so a few
2990 “obvious” things were not included. Specifically, the set of characters allowed in a scanset is
2991 limited to single-byte characters. In other similar places, multi-byte characters have been
2992 permitted, but for alignment with the ISO/IEC 9899: 1999 standard, it has not been done
2993 here.

2994 to:

2995 The set of characters allowed in a scanset is limited to single-byte characters. In other
2996 similar places, multi-byte characters have been permitted, but for alignment with the ISO C
2997 standard, it has not been done here.

2998 Ref 7.29.2.2 para 4

2999 On page 1004 line 34144 section `fwscanf()`, change:

3000 If a directive fails, as detailed below, the function shall return.

3001 to:

3002 When all directives have been executed, or if a directive fails (as detailed below), the
3003 function shall return.

3004 Ref 7.29.2.2 para 5

3005 On page 1004 line 34147 section `fwscanf()`, change:

3006 A directive composed of one or more white-space wide characters is executed by reading
3007 input until no more valid input can be read, or up to the first wide character which is not a
3008 white-space wide character, which remains unread.

3009 to:

3010 A directive composed of one or more white-space wide characters shall be executed by
3011 reading input up to the first wide character that is not a white-space wide character, which
3012 shall remain unread, or until no more wide characters can be read. The directive shall never
3013 fail.

3014 Ref 7.27.3, 7.1.4 para 5

3015 On page 1113 line 37680 section `gmtime()`, change:

3016 [CX]The `gmtime()` function need not be thread-safe.[/CX]

3017 to:

3018 The `gmtime()` function need not be thread-safe; however, `gmtime()` shall avoid data races
3019 with all functions other than itself, `asctime()`, `ctime()` and `localtime()`.

3020 Ref F.10.3.5 para 1
3021 On page 1133 line 38281 section `ilogb()`, add a new paragraph:

3022 [MX]When the correct result is representable in the range of the return type, the returned
3023 value shall be exact and shall be independent of the current rounding direction mode.[/MX]

3024 Ref F.10.3.5 para 3
3025 On page 1133 line 38282,38285,38288 section `ilogb()`, change:

3026 [XSI]On XSI-conformant systems, a domain error shall occur[/XSI]

3027 to:

3028 [XSI|MX]On XSI-conformant systems and on systems that support the IEC 60559 Floating-
3029 Point option, a domain error shall occur[/XSI|MX]

3030 Ref 7.12.6.5 para 2
3031 On page 1133 line 38291 section `ilogb()`, change:

3032 If the correct value is greater than `{INT_MAX}`, [MX]a domain error shall occur and[/MX]
3033 an unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error
3034 shall occur and `{INT_MAX}` shall be returned.[/XSI]

3035 If the correct value is less than `{INT_MIN}`, [MX]a domain error shall occur and[/MX] an
3036 unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error shall
3037 occur and `{INT_MIN}` shall be returned.[/XSI]

3038 to:

3039 If the correct value is greater than `{INT_MAX}` or less than `{INT_MIN}`, an unspecified
3040 value shall be returned. [XSI]On XSI-conformant systems, a domain error shall occur and
3041 `{INT_MAX}` or `{INT_MIN}`, respectively, shall be returned;[/XSI] [MX]if the IEC 60559
3042 Floating-Point option is supported, a domain error shall occur;[/MX] otherwise, a domain
3043 error or range error may occur.

3044 Ref F.10.3.5 para 3
3045 On page 1133 line 38300 section `ilogb()`, change:

3046 [XSI]The `x` argument is zero, NaN, or $\pm\text{Inf}$.[/XSI]

3047 to:

3048 [XSI|MX]The `x` argument is zero, NaN, or $\pm\text{Inf}$.[/XSI|MX]

3049 Ref F.10.11 para 1
3050 On page 1174 line 39604 section `isgreater()`,
3051 and page 1175 line 39642 section `isgreaterequal()`,
3052 and page 1177 line 39708 section `isless()`,
3053 and page 1178 line 39746 section `islessequal()`,
3054 and page 1179 line 39784 section `islessgreater()`, add a new paragraph:

3055 [MX]Relational operators and their corresponding comparison macros shall produce

3056 equivalent result values, even if argument values are represented in wider formats. Thus,
3057 comparison macro arguments represented in formats wider than their semantic types shall
3058 not be converted to the semantic types, unless the wide evaluation method converts operands
3059 of relational operators to their semantic types. The standard wide evaluation methods
3060 characterized by FLT_EVAL_METHOD equal to 1 or 2 (see [xref to <float.h>]) do not
3061 convert operands of relational operators to their semantic types.[/MX]

3062 (The editors may wish to merge the pages for the above interfaces to reduce duplication – they have
3063 duplicate APPLICATION USAGE as well.)

3064 Ref 7.30.2.2.1 para 4

3065 On page 1202 line 40411 section iswctype(), remove the CX shading from:

3066 If *charclass* is (**wctype_t**)0, these functions shall return 0.

3067 Ref 7.17.3.1

3068 On page 1229 line 41126 insert a new *kill_dependency()* section:

3069 **NAME**

3070 *kill_dependency* — terminate a dependency chain

3071 **SYNOPSIS**

3072 #include <stdatomic.h>
3073 *type* *kill_dependency*(*type* *y*);

3074 **DESCRIPTION**

3075 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3076 Any conflict between the requirements described here and the ISO C standard is
3077 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3078 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the
3079 `<stdatomic.h>` header nor support this macro.

3080 The *kill_dependency()* macro shall terminate a dependency chain (see [xref to XBD 4.12.1
3081 Memory Ordering]). The argument shall not carry a dependency to the return value.

3082 **RETURN VALUE**

3083 The *kill_dependency()* macro shall return the value of *y*.

3084 **ERRORS**

3085 No errors are defined.

3086 **EXAMPLES**

3087 None.

3088 **APPLICATION USAGE**

3089 None.

3090 **RATIONALE**

3091 None.

3092 **FUTURE DIRECTIONS**

- 3093 None.
- 3094 **SEE ALSO**
3095 XBD Section 4.12.1, <stdatomic.h>
- 3096 **CHANGE HISTORY**
3097 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.
- 3098 Ref 7.12.8.3, 7.1.4 para 5
3099 On page 1241 line 41433 section lgamma(), change:
- 3100 [CX]These functions need not be thread-safe.[/CX]
- 3101 to:
- 3102 [XSI]If concurrent calls are made to these functions, the value of *signgam* is indeterminate.[/
3103 XSI]
- 3104 Ref 7.12.8.3, 7.1.4 para 5
3105 On page 1242 line 41464 section lgamma(), add a new paragraph to APPLICATION USAGE:
- 3106 If the value of *signgam* will be obtained after a call to *lgamma()*, *lgammaf()*, or *lgammal()*,
3107 in order to ensure that the value will not be altered by another call in a different thread,
3108 applications should either restrict calls to these functions to be from a single thread or use a
3109 lock such as a mutex or spin lock to protect a critical section starting before the function call
3110 and ending after the value of *signgam* has been obtained.
- 3111 Ref 7.12.8.3, 7.1.4 para 5
3112 On page 1242 line 41466 section lgamma(), change RATIONALE from:
- 3113 None.
- 3114 to:
- 3115 Earlier versions of this standard did not require *lgamma()*, *lgammaf()*, and *lgammal()* to be
3116 thread-safe because *signgam* was a global variable. They are now required to be thread-safe
3117 to align with the ISO C standard (which, since the introduction of threads in 2011, requires
3118 that they avoid data races), with the exception that they need not avoid data races when
3119 storing a value in the *signgam* variable. Since *signgam* is not specified by the ISO C
3120 standard, this exception is not a conflict with that standard.
- 3121 Ref 7.11.2.1, 7.1.4 para 5
3122 On page 1262 line 42124 section localeconv(), change:
- 3123 [CX]The *localeconv()* function need not be thread-safe.[/CX]
- 3124 to:
- 3125 The *localeconv()* function need not be thread-safe; however, *localeconv()* shall avoid data
3126 races with all other functions.

3127 Ref 7.27.3, 7.1.4 para 5
3128 On page 1265 line 42217 section `localtime()`, change:

3129 [CX]The *localtime()* function need not be thread-safe.[/CX]

3130 to:
3131 The *localtime()* function need not be thread-safe; however, *localtime()* shall avoid data races
3132 with all functions other than itself, *asctime()*, *ctime()* and *gmtime()*.

3133 Ref F.10.3.11 para 2
3134 On page 1280 line 42723 section `logb()`, add a new paragraph:

3135 [MX]The returned value shall be exact and shall be independent of the current rounding
3136 direction mode.[/MX]

3137 Ref 7.13.2.1 para 1
3138 On page 1283 line 42780 section `longjmp()`, change:

3139 void `longjmp(jmp_buf env, int val)`;

3140 to:
3141 _Noreturn void `longjmp(jmp_buf env, int val)`;

3142 Ref 7.13.2.1 para 2
3143 On page 1283 line 42804 section `longjmp()`, remove the CX shading from:

3144 The effect of a call to *longjmp()* where initialization of the **jmp_buf** structure was not
3145 performed in the calling thread is undefined.

3146 Ref 7.13.2.1 para 4
3147 On page 1283 line 42807 section `longjmp()`, change:

3148 After *longjmp()* is completed, program execution continues ...

3149 to:
3150 After *longjmp()* is completed, thread execution shall continue ...

3151 Ref 7.22.3 para 1
3152 On page 1295 line 43144 section `malloc()`, change:

3153 a pointer to any type of object

3154 to:
3155 a pointer to any type of object with a fundamental alignment requirement

3156 Ref 7.22.3 para 2
3157 On page 1295 line 43150 section `malloc()`, add a new paragraph:

3158 For purposes of determining the existence of a data race, *malloc()* shall behave as though it

3159 accessed only memory locations accessible through its argument and not other static
3160 duration storage. The function may, however, visibly modify the storage that it allocates.
3161 Calls to *aligned_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV]
3162 [CX]*reallocarray()*,[/CX] and *realloc()* that allocate or deallocate a particular region of
3163 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such
3164 deallocation call shall synchronize with the next allocation (if any) in this order.

3165 Ref 7.22.3.1

3166 On page 1295 line 43171 section *malloc()*, add *aligned_alloc* to the SEE ALSO section.

3167 Ref 7.22.7.1 para 2

3168 On page 1297 line 43194 section *mblen()*, change:

3169 `mbtowl((wchar_t *)0, s, n);`

3170 to:

3171 `mbtowl((wchar_t *)0, (const char *)0, 0);`

3172 `mbtowl((wchar_t *)0, s, n);`

3173 Ref 7.22.7 para 1

3174 On page 1297 line 43198 section *mblen()*, change:

3175 this function shall be placed into its initial state by a call for which

3176 to:

3177 this function shall be placed into its initial state at program startup and can be returned to
3178 that state by a call for which

3179 Ref 7.22.7 para 1, 7.1.4 para 5

3180 On page 1297 line 43206 section *mblen()*, change:

3181 [CX]The *mblen()* function need not be thread-safe.[/CX]

3182 to:

3183 The *mblen()* function need not be thread-safe; however, it shall avoid data races with all
3184 other functions.

3185 Ref 7.29.6.3 para 1, 7.1.4 para 5

3186 On page 1299 line 43254 section *mbrlen()*, change:

3187 [CX]The *mbrlen()* function need not be thread-safe if called with a NULL *ps*
3188 argument.[/CX]

3189 to:

3190 If called with a null *ps* argument, the *mbrlen()* function need not be thread-safe; however,
3191 such calls shall avoid data races with calls to *mbrlen()* with a non-null argument and with
3192 calls to all other functions.

3193 Ref 7.28.1, 7.1.4 para 5

3194 On page 1301 line 43296 insert a new `mbrtoc16()` section:

3195 **NAME**

3196 `mbrtoc16`, `mbrtoc32` — convert a character to a Unicode character code (restartable)

3197 **SYNOPSIS**

3198 `#include <uchar.h>`

```
3199     size_t mbrtoc16(char16_t *restrict pc16, const char *restrict s,  
3200                   size_t n, mbstate_t *restrict ps);  
3201     size_t mbrtoc32(char32_t *restrict pc32, const char *restrict s,  
3202                   size_t n, mbstate_t *restrict ps);
```

3203 **DESCRIPTION**

3204 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3205 Any conflict between the requirements described here and the ISO C standard is
3206 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3207 If `s` is a null pointer, the `mbrtoc16()` function shall be equivalent to the call:

```
3208 mbrtoc16(NULL, "", 1, ps)
```

3209 In this case, the values of the parameters `pc16` and `n` are ignored.

3210 If `s` is not a null pointer, the `mbrtoc16()` function shall inspect at most `n` bytes beginning with
3211 the byte pointed to by `s` to determine the number of bytes needed to complete the next
3212 character (including any shift sequences). If the function determines that the next character
3213 is complete and valid, it shall determine the values of the corresponding wide characters and
3214 then, if `pc16` is not a null pointer, shall store the value of the first (or only) such character in
3215 the object pointed to by `pc16`. Subsequent calls shall store successive wide characters
3216 without consuming any additional input until all the characters have been stored. If the
3217 corresponding wide character is the null wide character, the resulting state described shall be
3218 the initial conversion state.

3219 If `ps` is a null pointer, the `mbrtoc16()` function shall use its own internal **`mbstate_t`** object,
3220 which shall be initialized at program start-up to the initial conversion state. Otherwise, the
3221 **`mbstate_t`** object pointed to by `ps` shall be used to completely describe the current
3222 conversion state of the associated character sequence.

3223 The behavior of this function is affected by the `LC_CTYPE` category of the current locale.

3224 The `mbrtoc16()` function shall not change the setting of `errno` if successful.

3225 The `mbrtoc32()` function shall behave the same way as `mbrtoc16()` except that the first
3226 parameter shall point to an object of type **`char32_t`** instead of **`char16_t`**. References to `pc16`
3227 in the above description shall apply as if they were `pc32` when they are being read as
3228 describing `mbrtoc32()`.

3229 If called with a null `ps` argument, the `mbrtoc16()` function need not be thread-safe; however,
3230 such calls shall avoid data races with calls to `mbrtoc16()` with a non-null argument and with
3231 calls to all other functions.

3232 If called with a null `ps` argument, the `mbrtoc32()` function need not be thread-safe; however,
3233 such calls shall avoid data races with calls to `mbrtoc32()` with a non-null argument and with
3234 calls to all other functions.

3235 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx
3236 calls *mbrtoc16()* or *mbrtoc32()* with a null pointer for *ps*.

3237 RETURN VALUE

3238 These functions shall return the first of the following that applies:

3239 0 If the next *n* or fewer bytes complete the character that corresponds to the null
3240 wide character (which is the value stored).

3241 between 1 and *n* inclusive

3242 If the next *n* or fewer bytes complete a valid character (which is the value
3243 stored); the value returned shall be the number of bytes that complete the
3244 character.

3245 **(size_t)**-3 If the next character resulting from a previous call has been stored, in which
3246 case no bytes from the input shall be consumed by the call.

3247 **(size_t)**-2 If the next *n* bytes contribute to an incomplete but potentially valid character,
3248 and all *n* bytes have been processed (no value is stored). When *n* has at least
3249 the value of the {MB_CUR_MAX} macro, this case can only occur if *s*
3250 points at a sequence of redundant shift sequences (for implementations with
3251 state-dependent encodings).

3252 **(size_t)**-1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
3253 contribute to a complete and valid character (no value is stored). In this case,
3254 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

3255 ERRORS

3256 These function shall fail if:

3257 [EILSEQ] An invalid character sequence is detected. [CX]In the POSIX locale
3258 an [EILSEQ] error cannot occur since all byte values are valid
3259 characters.[/CX]

3260 These functions may fail if:

3261 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

3262 EXAMPLES

3263 None.

3264 APPLICATION USAGE

3265 None.

3266 RATIONALE

3267 None.

3268 FUTURE DIRECTIONS

3269 None.

3270 SEE ALSO

3271 *c16rtomb*

3272 XBD <uchar.h>

3273 **CHANGE HISTORY**

3274 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3275 Ref 7.29.6.3 para 1, 7.1.4 para 5

3276 On page 1301 line 43322 section `mbrtowc()`, change:

3277 [CX]The `mbrtowc()` function need not be thread-safe if called with a NULL *ps*
3278 argument.[/CX]

3279 to:

3280 If called with a null *ps* argument, the `mbrtowc()` function need not be thread-safe; however,
3281 such calls shall avoid data races with calls to `mbrtowc()` with a non-null argument and with
3282 calls to all other functions.

3283 Ref 7.29.6.4 para 1, 7.1.4 para 5

3284 On page 1304 line 43451 section `mbsrtowcs()`, change:

3285 [CX]The `mbsnrtowcs()` and `mbsrtowcs()` functions need not be thread-safe if called with a
3286 NULL *ps* argument.[/CX]

3287 to:

3288 [CX]If called with a null *ps* argument, the `mbsnrtowcs()` function need not be thread-safe;
3289 however, such calls shall avoid data races with calls to `mbsnrtowcs()` with a non-null
3290 argument and with calls to all other functions.[/CX]

3291 If called with a null *ps* argument, the `mbsrtowcs()` function need not be thread-safe;
3292 however, such calls shall avoid data races with calls to `mbsrtowcs()` with a non-null
3293 argument and with calls to all other functions.

3294 Ref 7.22.7 para 1

3295 On page 1308 line 43557 section `mbtowc()`, change:

3296 this function is placed into its initial state by a call for which

3297 to:

3298 this function shall be placed into its initial state at program startup and can be returned to
3299 that state by a call for which

3300 Ref 7.22.7 para 1, 7.1.4 para 5

3301 On page 1308 line 43567 section `mbtowc()`, change:

3302 [CX]The `mbtowc()` function need not be thread-safe.[/CX]

3303 to:

3304 The *mbtowc()* function need not be thread-safe; however, it shall avoid data races with all
3305 other functions.

3306 Ref 7.24.5.1 para 2
3307 On page 1311 line 43642 section *memchr()*, change:

3308 Implementations shall behave as if they read the memory byte by byte from the beginning of
3309 the bytes pointed to by *s* and stop at the first occurrence of *c* (if it is found in the initial *n*
3310 bytes).

3311 to:

3312 The implementation shall behave as if it reads the bytes sequentially and stops as soon as a
3313 matching byte is found.

3314 Ref F.10.3.12 para 2
3315 On page 1346 line 44854 section *modf()*, add a new paragraph:

3316 [MX]The returned value shall be exact and shall be independent of the current rounding
3317 direction mode.[/MX]

3318 Ref 7.26.4
3319 On page 1384 line 46032 insert the following new *mtx_**() sections:

3320 **NAME**

3321 *mtx_destroy*, *mtx_init* — destroy and initialize a mutex

3322 **SYNOPSIS**

3323 `#include <threads.h>`

3324 `void mtx_destroy(mtx_t *mtx);`
3325 `int mtx_init(mtx_t *mtx, int type);`

3326 **DESCRIPTION**

3327 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3328 Any conflict between the requirements described here and the ISO C standard is
3329 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3330 The *mtx_destroy()* function shall release any resources used by the mutex pointed to by *mtx*.
3331 A destroyed mutex object can be reinitialized using *mtx_init()*; the results of otherwise
3332 referencing the object after it has been destroyed are undefined. It shall be safe to destroy an
3333 initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that
3334 another thread is attempting to lock, or a mutex that is being used in a *cond_timedwait()* or
3335 *cond_wait()* call by another thread, results in undefined behavior. The behavior is undefined if
3336 the value specified by the *mtx* argument to *mtx_destroy()* does not refer to an initialized
3337 mutex.

3338 The *mtx_init()* function shall initialize a mutex object with properties indicated by *type*,
3339 whose valid values include:

3340 *mtx_plain* for a simple non-recursive mutex,

3341 *mtx_timed* for a non-recursive mutex that supports timeout,

3342 `mtx_plain` | `mtx_recursive` for a simple recursive mutex, or

3343 `mtx_timed` | `mtx_recursive` for a recursive mutex that supports timeout.

3344 If the `mtx_init()` function succeeds, it shall set the mutex pointed to by `mtx` to a value that
3345 uniquely identifies the newly initialized mutex. Upon successful initialization, the state of
3346 the mutex becomes initialized and unlocked. Attempting to initialize an already initialized
3347 mutex results in undefined behavior.

3348 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for
3349 further requirements.

3350 These functions shall not be affected if the calling thread executes a signal handler during
3351 the call.[/CX]

3352 **RETURN VALUE**

3353 The `mtx_destroy()` function shall not return a value.

3354 The `mtx_init()` function shall return `thrd_success` on success or `thrd_error` if the
3355 request could not be honored.

3356 **ERRORS**

3357 No errors are defined.

3358 **EXAMPLES**

3359 None.

3360 **APPLICATION USAGE**

3361 A mutex can be destroyed immediately after it is unlocked. However, since attempting to
3362 destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that
3363 is being used in a `cond_timedwait()` or `cond_wait()` call by another thread results in undefined
3364 behavior, care must be taken to ensure that no other thread may be referencing the mutex.

3365 **RATIONALE**

3366 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3367 B.2.3].

3368 **FUTURE DIRECTIONS**

3369 None.

3370 **SEE ALSO**

3371 `mtx_lock`

3372 XBD <**threads.h**>

3373 **CHANGE HISTORY**

3374 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3375 **NAME**

3376 `mtx_lock`, `mtx_timedlock`, `mtx_trylock`, `mtx_unlock` — lock and unlock a mutex

3377 **SYNOPSIS**

```
3378     #include <threads.h>

3379     int mtx_lock(mtx_t *mtx);
3380     int mtx_timedlock(mtx_t * restrict mtx,
3381                      const struct timespec * restrict ts);
3382     int mtx_trylock(mtx_t *mtx);
3383     int mtx_unlock(mtx_t *mtx);
```

3384 **DESCRIPTION**

3385 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3386 Any conflict between the requirements described here and the ISO C standard is
3387 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3388 The *mtx_lock()* function shall block until it locks the mutex pointed to by *mtx*. If the mutex
3389 is non-recursive, the application shall ensure that it is not already locked by the calling
3390 thread.

3391 The *mtx_timedlock()* function shall block until it locks the mutex pointed to by *mtx* or until
3392 after the *TIME_UTC* -based calendar time pointed to by *ts*. The application shall ensure that
3393 the specified mutex supports timeout. [CX]Under no circumstance shall the function fail
3394 with a timeout if the mutex can be locked immediately. The validity of the *ts* parameter need
3395 not be checked if the mutex can be locked immediately.[/CX]

3396 The *mtx_trylock()* function shall endeavor to lock the mutex pointed to by *mtx*. If the mutex
3397 is already locked (by any thread, including the current thread), the function shall return
3398 without blocking. If the mutex is recursive and the mutex is currently owned by the calling
3399 thread, the mutex lock count (see below) shall be incremented by one and the *mtx_trylock()*
3400 function shall immediately return success.

3401 [CX]These functions shall not be affected if the calling thread executes a signal handler
3402 during the call; if a signal is delivered to a thread waiting for a mutex, upon return from the
3403 signal handler the thread shall resume waiting for the mutex as if it was not
3404 interrupted.[/CX]

3405 If a call to *mtx_lock()*, *mtx_timedlock()* or *mtx_trylock()* locks the mutex, prior calls to
3406 *mtx_unlock()* on the same mutex shall synchronize with this lock operation.

3407 The *mtx_unlock()* function shall unlock the mutex pointed to by *mtx* . The application shall
3408 ensure that the mutex pointed to by *mtx* is locked by the calling thread. [CX]If there are
3409 threads blocked on the mutex object referenced by *mtx* when *mtx_unlock()* is called,
3410 resulting in the mutex becoming available, the scheduling policy shall determine which
3411 thread shall acquire the mutex.[/CX]

3412 A recursive mutex shall maintain the concept of a lock count. When a thread successfully
3413 acquires a mutex for the first time, the lock count shall be set to one. Every time a thread
3414 relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks
3415 the mutex, the lock count shall be decremented by one. When the lock count reaches zero,
3416 the mutex shall become available for other threads to acquire.

3417 For purposes of determining the existence of a data race, mutex lock and unlock operations
3418 on mutexes of type **mtx_t** behave as atomic operations. All lock and unlock operations on a

3419 particular mutex occur in some particular total order.

3420 If *mtx* does not refer to an initialized mutex object, the behavior of these functions is
3421 undefined.

3422 RETURN VALUE

3423 The *mtx_lock()* and *mtx_unlock()* functions shall return `thrd_success` on success, or
3424 `thrd_error` if the request could not be honored.

3425 The *mtx_timedlock()* function shall return `thrd_success` on success, or `thrd_timedout`
3426 if the time specified was reached without acquiring the requested resource, or `thrd_error`
3427 if the request could not be honored.

3428 The *mtx_trylock()* function shall return `thrd_success` on success, or `thrd_busy` if the
3429 resource requested is already in use, or `thrd_error` if the request could not be honored.
3430 The *mtx_trylock()* function can spuriously fail to lock an unused resource, in which case it
3431 shall return `thrd_busy`.

3432 ERRORS

3433 See RETURN VALUE.

3434 EXAMPLES

3435 None.

3436 APPLICATION USAGE

3437 None.

3438 RATIONALE

3439 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
3440 B.2.3].

3441 Since `<pthread.h>` has no equivalent of the `mtx_timed` mutex property, if the `<threads.h>`
3442 interfaces are implemented as a thin wrapper around `<pthread.h>` interfaces (meaning
3443 `mtx_t` and `pthread_mutex_t` are the same type), all mutexes support timeout and
3444 *mtx_timedlock()* will not fail for a mutex that was not initialized with `mtx_timed`.
3445 Alternatively, implementations can use a less thin wrapper where `mtx_t` contains additional
3446 properties that are not held in `pthread_mutex_t` in order to be able to return a failure
3447 indication from *mtx_timedlock()* calls where the mutex was not initialized with
3448 `mtx_timed`.

3449 FUTURE DIRECTIONS

3450 None.

3451 SEE ALSO

3452 *mtx_destroy*, *timespec_get*

3453 XBD Section 4.12.2, `<threads.h>`

3454 CHANGE HISTORY

3455 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3456 Ref F.10.8.2 para 2
3457 On page 1388 line 46143 section `nan()`, add a new paragraph:

3458 [MX]The returned value shall be exact and shall be independent of the current rounding
3459 direction mode.[/MX]

3460 Ref F.10.8.3 para 2, F.10.8.4 para 2
3461 On page 1395 line 46388 section `nextafter()`, add a new paragraph:

3462 [MX]Even though underflow or overflow can occur, the returned value shall be independent
3463 of the current rounding direction mode.[/MX]

3464 Ref 7.22.3 para 2
3465 On page 1448 line 48069 section `posix_memalign()`, add a new (unshaded) paragraph:

3466 For purposes of determining the existence of a data race, `posix_memalign()` shall behave as
3467 though it accessed only memory locations accessible through its arguments and not other
3468 static duration storage. The function may, however, visibly modify the storage that it
3469 allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `posix_memalign()`, `realloc()`,
3470 and `reallocarray()` that allocate or deallocate a particular region of memory shall occur in a
3471 single total order (see [xref to XBD 4.12.1]), and each such deallocation call shall
3472 synchronize with the next allocation (if any) in this order.

3473 Ref 7.22.3.1
3474 On page 1449 line 48107 section `posix_memalign()`, add `aligned_alloc` to the SEE ALSO section.

3475 Ref F.10.4.4 para 1
3476 On page 1548 line 50724 section `pow()`, change:

3477 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 , a pole error shall
3478 occur and `pow()`, `powf()`, and `powl()` shall return `±HUGE_VAL`, `±HUGE_VALF`, and
3479 `±HUGE_VALL`, respectively if y is an odd integer, or `HUGE_VAL`, `HUGE_VALF`, and
3480 `HUGE_VALL`, respectively if y is not an odd integer.

3481 to:

3482 On systems that support the IEC 60559 Floating-Point option, if x is ± 0 :

- 3483 • if y is an odd integer, a pole error shall occur and `pow()`, `powf()`, and `powl()` shall
3484 return `±HUGE_VAL`, `±HUGE_VALF`, and `±HUGE_VALL`, respectively;
- 3485 • if y is finite and is not an odd integer, a pole error shall occur and `pow()`, `powf()`, and
3486 `powl()` shall return `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively;
- 3487 • if y is `-Inf`, a pole error may occur and `pow()`, `powf()`, and `powl()` shall return
3488 `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

3489 Ref 7.26
3490 On page 1603 line 52244 section `pthread_cancel()`, add a new paragraph:

3491 If *thread* refers to a thread that was created using `thrd_create()`, the behavior is undefined.

3492 Ref 7.26.5.6

3493 On page 1603 line 52277 section `pthread_cancel()`, add a new RATIONALE paragraph:

3494 Use of `pthread_cancel()` to cancel a thread that was created using `thrd_create()` is undefined
3495 because `thrd_join()` has no way to indicate a thread was cancelled. The standard developers
3496 considered adding a `thrd_cancelled` enumeration constant that `thrd_join()` would return in
3497 this case. However, this return would be unexpected in code that is written to conform to the
3498 ISO C standard, and it would also not solve the problem that threads which use only ISO C
3499 `<threads.h>` interfaces (such as ones created by third party libraries written to conform to
3500 the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not
3501 provide cancellation cleanup handlers.

3502 Ref 7.26.5.5

3503 On page 1639 line 53422 section `pthread_exit()`, change:

3504 `void pthread_exit(void *value_ptr);`

3505 to:

3506 `_Noreturn void pthread_exit(void *value_ptr);`

3507 Ref 7.26.6

3508 On page 1639 line 53427 section `pthread_exit()`, change:

3509 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3510 specific data, appropriate destructor functions shall be called in an unspecified order.

3511 to:

3512 After all cancellation cleanup handlers have been executed, if the thread has any thread-
3513 specific data (whether associated with key type `tss_t` or `pthread_key_t`), appropriate
3514 destructor functions shall be called in an unspecified order.

3515 Ref 7.26.5.5

3516 On page 1639 line 53432 section `pthread_exit()`, change:

3517 An implicit call to `pthread_exit()` is made when a thread other than the thread in which
3518 `main()` was first invoked returns from the start routine that was used to create it.

3519 to:

3520 An implicit call to `pthread_exit()` is made when a thread that was not created using
3521 `thrd_create()`, and is not the thread in which `main()` was first invoked, returns from the start
3522 routine that was used to create it.

3523 Ref 7.26.5.5

3524 On page 1639 line 53451 section `pthread_exit()`, change APPLICATION USAGE from:

3525 None.

3526 to:

3527 Calls to *pthread_exit()* should not be made from threads created using *thrd_create()*, as their
3528 exit status has a different type (**int** instead of **void ***). If *pthread_exit()* is called from the
3529 initial thread and it is not the last thread to terminate, other threads should not try to obtain
3530 its exit status using *thrd_join()*.

3531 Ref 7.26.5.5
3532 On page 1639 line 53453 section *pthread_exit()*, change:

3533 The normal mechanism by which a thread terminates is to return from the routine that was
3534 specified in the *pthread_create()* call that started it.

3535 to:

3536 The normal mechanism by which a thread that was started using *pthread_create()* terminates
3537 is to return from the routine that was specified in the *pthread_create()* call that started it.

3538 Ref 7.26.5.5, 7.26.6
3539 On page 1640 line 53470 section *pthread_exit()*, add *pthread_key_create*, *thrd_create*, *thrd_exit* and
3540 *tss_create* to the SEE ALSO section.

3541 Ref 7.26.5.5
3542 On page 1649 line 53748 section *pthread_join()*, add a new paragraph:

3543 If *thread* refers to a thread that was created using *thrd_create()* and the thread terminates, or
3544 has already terminated, by returning from its start routine, the behavior of *pthread_join()* is
3545 undefined. If *thread* refers to a thread that terminates, or has already terminated, by calling
3546 *thrd_exit()*, the behavior of *pthread_join()* is undefined.

3547 Ref 7.26.5.5
3548 On page 1651 line 53819 section *pthread_join()*, add a new RATIONALE paragraph:

3549 The *pthread_join()* function cannot be used to obtain the exit status of a thread that was
3550 created using *thrd_create()* and which terminates by returning from its start routine, or of a
3551 thread that terminates by calling *thrd_exit()*, because such threads have an **int** exit status,
3552 instead of the **void *** that *pthread_join()* returns via its *value_ptr* argument.

3553 Ref 7.22.4.7
3554 On page 1765 line 57040 insert the following new *quick_exit()* section:

3555 **NAME**
3556 *quick_exit* — terminate a process

3557 **SYNOPSIS**
3558 `#include <stdlib.h>`
3559 `_Noreturn void quick_exit(int status);`

3560 **DESCRIPTION**
3561 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3562 Any conflict between the requirements described here and the ISO C standard is
3563 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3564 The *quick_exit()* function shall cause normal process termination to occur. It shall not call
3565 functions registered with *atexit()* nor any registered signal handlers. If a process calls the
3566 *quick_exit()* function more than once, or calls the *exit()* function in addition to the
3567 *quick_exit()* function, the behavior is undefined. If a signal is raised while the *quick_exit()*
3568 function is executing, the behavior is undefined.

3569 The *quick_exit()* function shall first call all functions registered by *at_quick_exit()*, in the
3570 reverse order of their registration, except that a function is called after any previously
3571 registered functions that had already been called at the time it was registered. If, during the
3572 call to any such function, a call to the *longjmp()* [CX] or *siglongjmp()*[/CX] function is made
3573 that would terminate the call to the registered function, the behavior is undefined.

3574 If a function registered by a call to *at_quick_exit()* fails to return, the remaining registered
3575 functions shall not be called and the rest of the *quick_exit()* processing shall not be
3576 completed.

3577 Finally, the *quick_exit()* function shall terminate the process as if by a call to *_Exit(status)*.

3578 **RETURN VALUE**

3579 The *quick_exit()* function does not return.

3580 **ERRORS**

3581 No errors are defined.

3582 **EXAMPLES**

3583 None.

3584 **APPLICATION USAGE**

3585 None.

3586 **RATIONALE**

3587 None.

3588 **FUTURE DIRECTIONS**

3589 None.

3590 **SEE ALSO**

3591 *_Exit*, *at_quick_exit*, *atexit*, *exit*

3592 XBD <stdlib.h>

3593 **CHANGE HISTORY**

3594 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3595 Ref 7.22.2.1 para 3, 7.1.4 para 5

3596 On page 1767 line 57095 section *rand()*, change:

3597 [CX]The *rand()* function need not be thread-safe.[/CX]

3598 to:

3599 The *rand()* function need not be thread-safe; however, *rand()* shall avoid data races with all
3600 functions other than non-thread-safe pseudo-random sequence generation functions.

3601 Ref 7.22.2.2 para 3, 7.1.4 para 5

3602 On page 1767 line 57105 section *rand()*, add a new paragraph:

3603 The *srand()* function need not be thread-safe; however, *srand()* shall avoid data races with
3604 all functions other than non-thread-safe pseudo-random sequence generation functions.

3605 Ref 7.22.3 para 1,2; 7.22.3.5 para 2,3,4; 7.31.12 para 2

3606 On page 1788 line 57862-57892 section *realloc()*, after applying bugs 374 and 1218 replace the
3607 DESCRIPTION and RETURN VALUE sections with:

3608 **DESCRIPTION**

3609 For *realloc()*: [CX] The functionality described on this reference page is aligned with the
3610 ISO C standard. Any conflict between the requirements described here and the ISO C
3611 standard is unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3612 The *realloc()* function shall deallocate the old object pointed to by *ptr* and return a pointer to
3613 a new object that has the size specified by *size*. The contents of the new object shall be the
3614 same as that of the old object prior to deallocation, up to the lesser of the new and old sizes.
3615 Any bytes in the new object beyond the size of the old object have indeterminate values.

3616 [CX]The *reallocarray()* function shall be equivalent to the call `realloc(ptr, nelem *
3617 elsize)` except that overflow in the multiplication shall be an error.[/CX]

3618 If *ptr* is a null pointer, *realloc()* [CX]or *reallocarray()*[/CX] shall be equivalent to *malloc()*
3619 function for the specified size. Otherwise, if *ptr* does not match a pointer returned earlier by
3620 *aligned_alloc()*, *calloc()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] *realloc()*,
3621 [CX]*reallocarray()*, or a function in POSIX.1-20xx that allocates memory as if by *malloc()*,
3622 [/CX] or if the space has been deallocated by a call to *free()*, [CX]*reallocarray()*,[/CX] or
3623 *realloc()*, the behavior is undefined.

3624 If *size* is non-zero and memory for the new object is not allocated, the old object shall not be
3625 deallocated.

3626 The order and contiguity of storage allocated by successive calls to *realloc()* [CX]or
3627 *reallocarray()*[/CX] is unspecified. The pointer returned if the allocation succeeds shall be
3628 suitably aligned so that it may be assigned to a pointer to any type of object with a
3629 fundamental alignment requirement and then used to access such an object in the space
3630 allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a
3631 pointer to an object disjoint from any other object. The pointer returned shall point to the
3632 start (lowest byte address) of the allocated space. If the space cannot be allocated, a null
3633 pointer shall be returned.

3634 For purposes of determining the existence of a data race, *realloc()* [CX]or
3635 *reallocarray()*[/CX] shall behave as though it accessed only memory locations accessible
3636 through its arguments and not other static duration storage. The function may, however,
3637 visibly modify the storage that it allocates or deallocates. Calls to *aligned_alloc()*, *calloc()*,
3638 *free()*, *malloc()*, [ADV]*posix_memalign()*,[/ADV] [CX]*reallocarray()*,[/CX] and *realloc()*
3639 that allocate or deallocate a particular region of memory shall occur in a single total order

3640 (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the next
3641 allocation (if any) in this order.

3642 **RETURN VALUE**

3643 Upon successful completion, *realloc()* [CX]and *reallocarray()*[CX] shall return a pointer to
3644 the new object (which can have the same value as a pointer to the old object), or a null
3645 pointer if the new object has not been allocated.

3646 [OB]If size is zero,[/OB]

3647 [OB CX]or either *nelem* or *elsize* is 0,[/OB CX]

3648 [OB]either:

- 3649 • A null pointer shall be returned [CX]and, if *ptr* is not a null pointer, *errno* shall be set
3650 to [EINVAL].[/CX]
- 3651 • A pointer to the allocated space shall be returned, and the memory object pointed to
3652 by *ptr* shall be freed. The application shall ensure that the pointer is not used to
3653 access an object.[/OB]

3654 If there is not enough available memory, *realloc()* [CX]and *reallocarray()*[CX] shall return
3655 a null pointer [CX]and set *errno* to [ENOMEM][CX].

3656 Ref 7.22.3.5 para 3,4

3657 On page 1789 line 57899 section *realloc()*, change:

3658 The description of *realloc()* has been modified from previous versions of this standard to
3659 align with the ISO/IEC 9899: 1999 standard. Previous versions explicitly permitted a call to
3660 *realloc(p, 0)* to free the space pointed to by *p* and return a null pointer. While this behavior
3661 could be interpreted as permitted by this version of the standard, the C language committee
3662 have indicated that this interpretation is incorrect. Applications should assume that if
3663 *realloc()* returns a null pointer, the space pointed to by *p* has not been freed. Since this could
3664 lead to double-frees, implementations should also set *errno* if a null pointer actually
3665 indicates a failure, and applications should only free the space if *errno* was changed.

3666 to:

3667 The ISO C standard makes it implementation-defined whether a call to *realloc(p, 0)* frees the
3668 space pointed to by *p* if it returns a null pointer because memory for the new object was not
3669 allocated. POSIX.1 instead requires that implementations set *errno* if a null pointer is
3670 returned and the space has not been freed, and POSIX applications should only free the
3671 space if *errno* was changed.

3672 Ref 7.31.12 para 2

3673 On page 1789 line 57909-57912 section *realloc()*, change FUTURE DIRECTIONS to:

3674 The ISO C standard states that invoking *realloc()* with a *size* argument equal to zero is an
3675 obsolescent feature. This feature may be removed in a future version of this standard.

3676 Ref 7.22.3.1

3677 On page 1789 line 57914 section *realloc()*, add *aligned_alloc* to the SEE ALSO section.

3678 Ref F.10.7.2 para 2

3679 On page 1809 line 58638 section *remainder()*, add a new paragraph:

3680 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3681 Ref F.10.7.3 para 2
3682 On page 1814 line 58758 section `remquo()`, add a new paragraph:

3683 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3684 Ref F.10.6.6 para 3
3685 On page 1828 line 59258 section `round()`, add a new paragraph:

3686 [MX]These functions may raise the inexact floating-point exception for finite non-integer
3687 arguments.[/MX]

3688 Ref F.10.6.6 para 3
3689 On page 1828 line 59272 section `round()`, delete from APPLICATION USAGE:

3690 These functions may raise the inexact floating-point exception if the result differs in value
3691 from the argument.

3692 Ref F.10.3.13 para 2
3693 On page 1829 line 59306 section `scalbln()`, add a new paragraph:

3694 [MX]If the calculation does not overflow or underflow, the returned value shall be exact and
3695 shall be independent of the current rounding direction mode.[/MX]

3696 Ref 7.11.1.1 para 5
3697 On page 1903 line 61520 section `setlocale()`, change:

3698 [CX]The `setlocale()` function need not be thread-safe.[/CX]

3699 to:

3700 The `setlocale()` function need not be thread-safe; however, it shall avoid data races with all
3701 function calls that do not affect and are not affected by the global locale.

3702 Ref 7.13.2.1 para 1
3703 On page 1970 line 63497 section `siglongjmp()`, change:

3704 `void siglongjmp(sigjmp_buf env, int val);`

3705 to:

3706 `_Noreturn void siglongjmp(sigjmp_buf env, int val);`

3707 Ref 7.13.2.1 para 4
3708 On page 1970 line 63504 section `siglongjmp()`, change:

3709 After `siglongjmp()` is completed, program execution shall continue ...

3710 to:

- 3711 After *siglongjmp()* is completed, thread execution shall continue ...
- 3712 Ref 7.14.1.1 para 5
3713 On page 1971 line 63564 section *signal()*, change:
- 3714 with static storage duration
- 3715 to:
- 3716 with static or thread storage duration that is not a lock-free atomic object
- 3717 Ref 7.14.1.1 para 7
3718 On page 1972 line 63573 section *signal()*, add a new paragraph:
- 3719 [*CX*]The *signal()* function is required to be thread-safe. (See [xref to 2.9.1 Thread-Safety].)
3720 [/*CX*]
- 3721 Ref 7.14.1.1 para 7
3722 On page 1972 line 63591 section *signal()*, change RATIONALE from:
- 3723 None.
- 3724 to:
- 3725 The ISO C standard says that the use of *signal()* in a multi-threaded program results in
3726 undefined behavior. However, POSIX.1 has required *signal()* to be thread-safe since before
3727 threads were added to the ISO C standard.
- 3728 Ref F.10.4.5 para 1
3729 On page 2009 line 64624 section *sqrt()*, add:
- 3730 [*MX*]The returned value shall be dependent on the current rounding direction mode.[/*MX*]
- 3731 Ref 7.24.6.2 para 3, 7.1.4 para 5
3732 On page 2035 line 65231 section *strerror()*, change:
- 3733 [*CX*]The *strerror()* function need not be thread-safe.[/*CX*]
- 3734 to:
- 3735 The *strerror()* function need not be thread-safe; however, *strerror()* shall avoid data races
3736 with all other functions.
- 3737 Ref 7.22.1.3 para 10
3738 On page 2073 line 66514 section *strtod()*, change:
- 3739 If the correct value is outside the range of representable values
- 3740 to:
- 3741 If the correct value would cause an overflow and default rounding is in effect
- 3742 Ref 7.24.5.8 para 6, 7.1.4 para 5

3743 On page 2078 line 66674 section `strtok()`, change:

3744 [CX]The `strtok()` function need not be thread-safe.[/CX]

3745 to:

3746 The `strtok()` function need not be thread-safe; however, `strtok()` shall avoid data races with
3747 all other functions.

3748 Ref 7.22.4.8, 7.1.4 para 5

3749 On page 2107 line 67579 section `system()`, change:

3750 The `system()` function need not be thread-safe.

3751 to:

3752 [CX]If concurrent calls to `system()` are made from multiple threads, it is unspecified
3753 whether:

- 3754 • each call saves and restores the dispositions of the SIGINT and SIGQUIT signals
3755 independently, or
- 3756 • in a set of concurrent calls the dispositions in effect after the last call returns are
3757 those that were in effect on entry to the first call.

3758 If a thread is cancelled while it is in a call to `system()`, it is unspecified whether the child
3759 process is terminated and waited for, or is left running.[/CX]

3760 Ref 7.22.4.8, 7.1.4 para 5

3761 On page 2108 line 67627 section `system()`, change:

3762 Using the `system()` function in more than one thread in a process or when the SIGCHLD
3763 signal is being manipulated by more than one thread in a process may produce unexpected
3764 results.

3765 to:

3766 Although `system()` is required to be thread-safe, it is recommended that concurrent calls
3767 from multiple threads are avoided, since `system()` is not required to coordinate the saving
3768 and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of
3769 overlapping calls, and therefore the signals might end up being set to ignored after the last
3770 call returns. Applications should also avoid cancelling a thread while it is in a call to
3771 `system()` as the child process may be left running in that event. In addition, if another thread
3772 alters the disposition of the SIGCHLD signal, a call to `signal()` may produce unexpected
3773 results.

3774 Ref 7.22.4.8, 7.1.4 para 5

3775 On page 2109 line 67675 section `system()`, delete:

3776 `#include <signal.h>`

3777 Ref 7.22.4.8, 7.1.4 para 5

3778 On page 2109 line 67692,67696,67712 section `system()`, change `sigprocmask` to
3779 `pthread_sigmask`.

3780 Ref 7.22.4.8, 7.1.4 para 5
3781 On page 2110 line 67718 section `system()`, change:

3782 Note also that the above example implementation is not thread-safe. Implementations can
3783 provide a thread-safe `system()` function, but doing so involves complications such as how to
3784 restore the signal dispositions for SIGINT and SIGQUIT correctly if there are overlapping
3785 calls, and how to deal with cancellation. The example above would not restore the signal
3786 dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-
3787 safe implementation since canceling a non-thread-safe function results in undefined
3788 behavior (see Section 2.9.5.2, on page 518). To avoid leaking a process ID, a thread-safe
3789 implementation would need to terminate the child process when acting on a cancellation.

3790 to:

3791 Earlier versions of this standard did not require `system()` to be thread-safe because it alters
3792 the process-wide disposition of the SIGINT and SIGQUIT signals. It is now required to be
3793 thread-safe to align with the ISO C standard, which (since the introduction of threads in
3794 2011) requires that it avoids data races. However, the function is not required to coordinate
3795 the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set
3796 of overlapping calls, and the above example does not do so. The example also does not
3797 terminate and wait for the child process if the calling thread is cancelled, and so would leak
3798 a process ID in that event.

3799 Ref 7.26.5
3800 On page 2148 line 68796 insert the following new `thrd_*` sections:

3801 **NAME**

3802 `thrd_create` — thread creation

3803 **SYNOPSIS**

3804 `#include <threads.h>`

3805 `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`

3806 **DESCRIPTION**

3807 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3808 Any conflict between the requirements described here and the ISO C standard is
3809 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3810 The `thrd_create()` function shall create a new thread executing `func(arg)`. If the `thrd_create()`
3811 function succeeds, it shall set the object pointed to by `thr` to the identifier of the newly
3812 created thread. (A thread's identifier might be reused for a different thread once the original
3813 thread has exited and either been detached or joined to another thread.) The completion of
3814 the `thrd_create()` function shall synchronize with the beginning of the execution of the new
3815 thread.

3816 [CX]The signal state of the new thread shall be initialized as follows:

- 3817 • The signal mask shall be inherited from the creating thread.
- 3818 • The set of signals pending for the new thread shall be empty.

3819 The thread-local current locale shall not be inherited from the creating thread.

3820 The floating-point environment shall be inherited from the creating thread.[/CX]

3821 [XSI] The alternate stack shall not be inherited from the creating thread.[/XSI]

3822 Returning from *func* shall have the same behavior as invoking *thrd_exit()* with the value
3823 returned from *func*.

3824 If *thrd_create()* fails, no new thread shall be created and the contents of the location
3825 referenced by *thr* are undefined.

3826 [CX]The *thrd_create()* function shall not be affected if the calling thread executes a signal
3827 handler during the call.[/CX]

3828 **RETURN VALUE**

3829 The *thrd_create()* function shall return *thrd_success* on success; or *thrd_nomem* if no
3830 memory could be allocated for the thread requested; or *thrd_error* if the request could not
3831 be honored, [CX]such as if the system-imposed limit on the total number of threads in a
3832 process {*PTHREAD_THREADS_MAX*} would be exceeded.[/CX]

3833 **ERRORS**

3834 See RETURN VALUE.

3835 **EXAMPLES**

3836 None.

3837 **APPLICATION USAGE**

3838 There is no requirement on the implementation that the ID of the created thread be available
3839 before the newly created thread starts executing. The calling thread can obtain the ID of the
3840 created thread through the *thr* argument of the *thrd_create()* function, and the newly created
3841 thread can obtain its ID by a call to *thrd_current()*.

3842 **RATIONALE**

3843 The *thrd_create()* function is not affected by signal handlers for the reasons stated in [xref to
3844 XRAT B.2.3].

3845 **FUTURE DIRECTIONS**

3846 None.

3847 **SEE ALSO**

3848 *pthread_create*, *thrd_current*, *thrd_detach*, *thrd_exit*, *thrd_join*

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_current* — get the calling thread ID

3854 **SYNOPSIS**

3855 `#include <threads.h>`

3856 `thrd_t thrd_current(void);`

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_current()* function shall identify the thread that called it.

3862 **RETURN VALUE**

3863 The *thrd_current()* function shall return the thread ID of the thread that called it.

3864 The *thrd_current()* function shall always be successful. No return value is reserved to
3865 indicate an error.

3866 **ERRORS**

3867 No errors are defined.

3868 **EXAMPLES**

3869 None.

3870 **APPLICATION USAGE**

3871 None.

3872 **RATIONALE**

3873 None.

3874 **FUTURE DIRECTIONS**

3875 None.

3876 **SEE ALSO**

3877 *pthread_self*, *thrd_create*, *thrd_equal*

3878 XBD Section 4.12.2, **<threads.h>**

3879 **CHANGE HISTORY**

3880 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3881 **NAME**

3882 *thrd_detach* — detach a thread

3883 **SYNOPSIS**

3884 `#include <threads.h>`

3885 `int thrd_detach(thrd_t thr);`

3886 **DESCRIPTION**

3887 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3888 Any conflict between the requirements described here and the ISO C standard is

3889 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]
3890 The *thrd_detach()* function shall change the thread *thr* from joinable to detached, indicating
3891 to the implementation that any resources allocated to the thread can be reclaimed when that
3892 thread terminates. The application shall ensure that the thread identified by *thr* has not been
3893 previously detached or joined with another thread.
3894 [CX]The *thrd_detach()* function shall not be affected if the calling thread executes a signal
3895 handler during the call.[/CX]

3896 **RETURN VALUE**

3897 The *thrd_detach()* function shall return `thrd_success` on success or `thrd_error` if the
3898 request could not be honored.

3899 **ERRORS**

3900 No errors are defined.

3901 **EXAMPLES**

3902 None.

3903 **APPLICATION USAGE**

3904 None.

3905 **RATIONALE**

3906 The *thrd_detach()* function is not affected by signal handlers for the reasons stated in [xref
3907 to XRAT B.2.3].

3908 **FUTURE DIRECTIONS**

3909 None.

3910 **SEE ALSO**

3911 *pthread_detach*, *thrd_create*, *thrd_join*

3912 XBD <**threads.h**>

3913 **CHANGE HISTORY**

3914 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3915 **NAME**

3916 `thrd_equal` — compare thread IDs

3917 **SYNOPSIS**

3918 `#include <threads.h>`

3919 `int thrd_equal(thrd_t thr0, thrd_t thr1);`

3920 **DESCRIPTION**

3921 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3922 Any conflict between the requirements described here and the ISO C standard is
3923 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3924 The *thrd_equal()* function shall determine whether the thread identified by *thr0* refers to the

3925 thread identified by *thr1*.

3926 [CX]The *thrd_equal()* function shall not be affected if the calling thread executes a signal
3927 handler during the call.[/CX]

3928 **RETURN VALUE**

3929 The *thrd_equal()* function shall return a non-zero value if *thr0* and *thr1* are equal; otherwise,
3930 zero shall be returned.

3931 If either *thr0* or *thr1* is not a valid thread ID [CX]and is not equal to `PTHREAD_NULL`
3932 (which is defined in `<pthread.h>`[/CX]), the behavior is undefined.

3933 **ERRORS**

3934 No errors are defined.

3935 **EXAMPLES**

3936 None.

3937 **APPLICATION USAGE**

3938 None.

3939 **RATIONALE**

3940 See the RATIONALE section for *pthread_equal()*.

3941 The *thrd_equal()* function is not affected by signal handlers for the reasons stated in [xref to
3942 XRAT B.2.3].

3943 **FUTURE DIRECTIONS**

3944 None.

3945 **SEE ALSO**

3946 *pthread_equal*, *thrd_current*

3947 XBD `<pthread.h>`, `<threads.h>`

3948 **CHANGE HISTORY**

3949 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3950 **NAME**

3951 *thrd_exit* — thread termination

3952 **SYNOPSIS**

3953 `#include <threads.h>`

3954 `_Noreturn void thrd_exit(int res);`

3955 **DESCRIPTION**

3956 [CX] The functionality described on this reference page is aligned with the ISO C standard.
3957 Any conflict between the requirements described here and the ISO C standard is
3958 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3959 For every thread-specific storage key [CX](regardless of whether it has type `tss_t` or

3960 **pthread_key_t**[/CX] which was created with a non-null destructor and for which the value
3961 is non-null, *thrd_exit()* shall set the value associated with the key to a null pointer value and
3962 then invoke the destructor with its previous value. The order in which destructors are
3963 invoked is unspecified.

3964 If after this process there remain keys with both non-null destructors and values, the
3965 implementation shall repeat this process up to [CX]
3966 {PTHREAD_DESTRUCTOR_ITERATIONS}[/CX] times.

3967 Following this, the *thrd_exit()* function shall terminate execution of the calling thread and
3968 shall set its exit status to *res*. [CX]Thread termination shall not release any application
3969 visible process resources, including, but not limited to, mutexes and file descriptors, nor
3970 shall it perform any process-level cleanup actions, including, but not limited to, calling any
3971 *atexit()* routines that might exist.[/CX]

3972 An implicit call to *thrd_exit()* is made when a thread that was created using *thrd_create()*
3973 returns from the start routine that was used to create it (see [xref to *thrd_create()*]).

3974 [CX]The behavior of *thrd_exit()* is undefined if called from a destructor function that was
3975 invoked as a result of either an implicit or explicit call to *thrd_exit()*.[/CX]

3976 The process shall exit with an exit status of zero after the last thread has been terminated.
3977 The behavior shall be as if the implementation called *exit()* with a zero argument at thread
3978 termination time.

3979 **RETURN VALUE**

3980 This function shall not return a value.

3981 **ERRORS**

3982 No errors are defined.

3983 **EXAMPLES**

3984 None.

3985 **APPLICATION USAGE**

3986 Calls to *thrd_exit()* should not be made from threads created using *pthread_create()* or via a
3987 SIGEV_THREAD notification, as their exit status has a different type (**void *** instead of
3988 **int**). If *thrd_exit()* is called from the initial thread and it is not the last thread to terminate,
3989 other threads should not try to obtain its exit status using *pthread_join()*.

3990 **RATIONALE**

3991 The normal mechanism by which a thread that was started using *thrd_create()* terminates is
3992 to return from the function that was specified in the *thrd_create()* call that started it. The
3993 *thrd_exit()* function provides the capability for such a thread to terminate without requiring a
3994 return from the start routine of that thread, thereby providing a function analogous to *exit()*.

3995 Regardless of the method of thread termination, the destructors for any existing thread-
3996 specific data are executed.

3997 **FUTURE DIRECTIONS**

3998 None.

3999 **SEE ALSO**

4000 *exit, pthread_create, thrd_join*

4001 XBD <threads.h>

4002 CHANGE HISTORY

4003 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4004 NAME

4005 *thrd_join* — wait for thread termination

4006 SYNOPSIS

4007 `#include <threads.h>`

4008 `int thrd_join(thrd_t thr, int *res);`

4009 DESCRIPTION

4010 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4011 Any conflict between the requirements described here and the ISO C standard is
4012 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4013 The *thrd_join()* function shall join the thread identified by *thr* with the current thread by
4014 blocking until the other thread has terminated. If the parameter *res* is not a null pointer,
4015 *thrd_join()* shall store the thread's exit status in the integer pointed to by *res*. The
4016 termination of the other thread shall synchronize with the completion of the *thrd_join()*
4017 function. The application shall ensure that the thread identified by *thr* has not been
4018 previously detached or joined with another thread.

4019 The results of multiple simultaneous calls to *thrd_join()* specifying the same target thread
4020 are undefined.

4021 The behavior is undefined if the value specified by the *thr* argument to *thrd_join()* refers to
4022 the calling thread.

4023 [CX]It is unspecified whether a thread that has exited but remains unjoined counts against
4024 {PTHREAD_THREADS_MAX}.

4025 If *thr* refers to a thread that was created using *pthread_create()* or via a SIGEV_THREAD
4026 notification and the thread terminates, or has already terminated, by returning from its start
4027 routine, the behavior of *thrd_join()* is undefined. If *thr* refers to a thread that terminates, or
4028 has already terminated, by calling *pthread_exit()* or by being cancelled, the behavior of
4029 *thrd_join()* is undefined.

4030 The *thrd_join()* function shall not be affected if the calling thread executes a signal handler
4031 during the call.[/CX]

4032 RETURN VALUE

4033 The *thrd_join()* function shall return *thrd_success* on success or *thrd_error* if the
4034 request could not be honored.

4035 [CX]It is implementation-defined whether *thrd_join()* detects deadlock situations; if it does
4036 detect them, it shall return *thrd_error* when one is detected.[/CX]

4037 **ERRORS**

4038 See RETURN VALUE.

4039 **EXAMPLES**

4040 None.

4041 **APPLICATION USAGE**

4042 None.

4043 **RATIONALE**

4044 The *thrd_join()* function provides a simple mechanism allowing an application to wait for a
4045 thread to terminate. After the thread terminates, the application may then choose to clean up
4046 resources that were used by the thread. For instance, after *thrd_join()* returns, any
4047 application-provided stack storage could be reclaimed.

4048 The *thrd_join()* or *thrd_detach()* function should eventually be called for every thread that is
4049 created using *thrd_create()* so that storage associated with the thread may be reclaimed.

4050 The *thrd_join()* function cannot be used to obtain the exit status of a thread that was created
4051 using *pthread_create()* or via a SIGEV_THREAD notification and which terminates by
4052 returning from its start routine, or of a thread that terminates by calling *pthread_exit()*,
4053 because such threads have a **void *** exit status, instead of the **int** that *thrd_join()* returns via
4054 its *res* argument.

4055 The *thrd_join()* function cannot be used to obtain the exit status of a thread that terminates
4056 by being cancelled because it has no way to indicate that a thread was cancelled. (The
4057 *pthread_join()* function does this by returning a reserved **void *** exit status; it is not possible
4058 to reserve an **int** value for this purpose without introducing a conflict with the ISO C
4059 standard.) The standard developers considered adding a *thrd_cancelled* enumeration
4060 constant that *thrd_join()* would return in this case. However, this return would be
4061 unexpected in code that is written to conform to the ISO C standard, and it would also not
4062 solve the problem that threads which use only ISO C **<threads.h>** interfaces (such as ones
4063 created by third party libraries written to conform to the ISO C standard) have no way to
4064 handle being cancelled, as the ISO C standard does not provide cancellation cleanup
4065 handlers.

4066 The *thrd_join()* function is not affected by signal handlers for the reasons stated in [xref to
4067 XRAT B.2.3].

4068 **FUTURE DIRECTIONS**

4069 None.

4070 **SEE ALSO**

4071 *pthread_create*, *pthread_exit*, *pthread_join*, *thrd_create*, *thrd_exit*

4072 XBD Section 4.12.2, **<threads.h>**

4073 **CHANGE HISTORY**

4074 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4075 **NAME**

4076 *thrd_sleep* — suspend execution for an interval

4077 **SYNOPSIS**

4078 `#include <threads.h>`

4079 `int thrd_sleep(const struct timespec *duration,`
4080 `struct timespec *remaining);`

4081 **DESCRIPTION**

4082 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4083 Any conflict between the requirements described here and the ISO C standard is
4084 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4085 The *thrd_sleep()* function shall suspend execution of the calling thread until either the
4086 interval specified by *duration* has elapsed or a signal is delivered to the calling thread whose
4087 action is to invoke a signal-catching function or to terminate the process. If interrupted by a
4088 signal and the *remaining* argument is not null, the amount of time remaining (the requested
4089 interval minus the time actually slept) shall be stored in the interval it points to. The
4090 *duration* and *remaining* arguments can point to the same object.

4091 The suspension time may be longer than requested because the interval is rounded up to an
4092 integer multiple of the sleep resolution or because of the scheduling of other activity by the
4093 system. But, except for the case of being interrupted by a signal, the suspension time shall
4094 not be less than that specified, as measured by the system clock `TIME_UTC`.

4095 **RETURN VALUE**

4096 The *thrd_sleep()* function shall return zero if the requested time has elapsed, `-1` if it has
4097 been interrupted by a signal, or a negative value (which may also be `-1`) if it fails for any
4098 other reason. [CX]If it returns a negative value, it shall set *errno* to indicate the error.[/CX]

4099 **ERRORS**

4100 [CX]The *thrd_sleep()* function shall fail if:

4101 [EINTR]

4102 The *thrd_sleep()* function was interrupted by a signal.

4103 [EINVAL]

4104 The *duration* argument specified a nanosecond value less than zero or greater than or
4105 equal to 1000 million.[/CX]

4106 **EXAMPLES**

4107 None.

4108 **APPLICATION USAGE**

4109 Since the return value may be `-1` for errors other than [EINTR], applications should examine
4110 *errno* to distinguish [EINTR] from other errors (and thus determine whether the unslept time
4111 is available in the interval pointed to by *remaining*).

4112 **RATIONALE**

4113 The *thrd_sleep()* function is identical to the *nanosleep()* function except that the return value
4114 may be any negative value when it fails with an error other than [EINTR].

4115 **FUTURE DIRECTIONS**

4116 None.

4117 **SEE ALSO**

4118 *nanosleep*

4119 XBD <**threads.h**>, <**time.h**>

4120 **CHANGE HISTORY**

4121 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4122 **NAME**

4123 *thrd_yield* — yield the processor

4124 **SYNOPSIS**

4125 `#include <threads.h>`

4126 `void thrd_yield(void);`

4127 **DESCRIPTION**

4128 [CX] The functionality described on this reference page is aligned with the ISO C standard.

4129 Any conflict between the requirements described here and the ISO C standard is

4130 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4131 [CX]The *thrd_yield()* function shall force the running thread to relinquish the processor until

4132 it again becomes the head of its thread list.[/CX]

4133 **RETURN VALUE**

4134 This function shall not return a value.

4135 **ERRORS**

4136 No errors are defined.

4137 **EXAMPLES**

4138 None.

4139 **APPLICATION USAGE**

4140 See the APPLICATION USAGE section for *sched_yield()*.

4141 **RATIONALE**

4142 The *thrd_yield()* function is identical to the *sched_yield()* function except that it does not

4143 return a value.

4144 **FUTURE DIRECTIONS**

4145 None.

4146 **SEE ALSO**

4147 *sched_yield*

4148 XBD <**threads.h**>

4149 **CHANGE HISTORY**

4150 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4151 Ref 7.27.2.5

4152 On page 2161 line 69278 insert a new `timespec_get()` section:

4153 **NAME**

4154 `timespec_get` — get time

4155 **SYNOPSIS**

4156 `#include <time.h>`

4157 `int timespec_get(struct timespec *ts, int base);`

4158 **DESCRIPTION**

4159 [CX] The functionality described on this reference page is aligned with the ISO C standard.

4160 Any conflict between the requirements described here and the ISO C standard is

4161 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4162 The `timespec_get()` function shall set the interval pointed to by `ts` to hold the current

4163 calendar time based on the specified time base.

4164 [CX]If `base` is `TIME_UTC`, the members of `ts` shall be set to the same values as would be

4165 set by a call to `clock_gettime(CLOCK_REALTIME, ts)`. If the number of seconds will not

4166 fit in an object of type `time_t`, the function shall return zero.[/CX]

4167 **RETURN VALUE**

4168 If the `timespec_get()` function is successful it shall return the non-zero value `base`; otherwise,

4169 it shall return zero.

4170 **ERRORS**

4171 See DESCRIPTION.

4172 **EXAMPLES**

4173 None.

4174 **APPLICATION USAGE**

4175 None.

4176 **RATIONALE**

4177 None.

4178 **FUTURE DIRECTIONS**

4179 None.

4180 **SEE ALSO**

4181 `clock_getres`, `time`

4182 XBD <`time.h`>

4183 **CHANGE HISTORY**

4184 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4185 Ref 7.21.4.4 para 4, 7.1.4 para 5

4186 On page 2164 line 69377 section `tmpnam()`, change:

4187 [CX]The *tmpnam()* function need not be thread-safe if called with a NULL parameter.[/CX]

4188 to:

4189 If called with a null pointer argument, the *tmpnam()* function need not be thread-safe;
4190 however, such calls shall avoid data races with calls to *tmpnam()* with a non-null argument
4191 and with calls to all other functions.

4192 Ref 7.30.3.2.1 para 4

4193 On page 2171 line 69568 section *towctrans()*, change:

4194 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped
4195 value of *wc* using the mapping described by *desc*. Otherwise, they shall return *wc*
4196 unchanged.

4197 to:

4198 If successful, the *towctrans()* [CX]and *towctrans_l()*[/CX] functions shall return the mapped
4199 value of *wc* using the mapping described by *desc*, or the value of *wc* unchanged if *desc* is
4200 zero. [CX]Otherwise, they shall return *wc* unchanged.[/CX]

4201 Ref F.10.6.8 para 2

4202 On page 2177 line 69716 section *trunc()*, add a new paragraph:

4203 [MX]These functions may raise the inexact floating-point exception for finite non-integer
4204 arguments.[/MX]

4205 Ref F.10.6.8 para 1,2

4206 On page 2177 line 69719 section *trunc()*, change:

4207 [MX]The result shall have the same sign as *x*.[/MX]

4208 to:

4209 [MX]The returned value shall be exact, shall be independent of the current rounding
4210 direction mode, and shall have the same sign as *x*.[/MX]

4211 Ref F.10.6.8 para 2

4212 On page 2177 line 69730 section *trunc()*, delete from APPLICATION USAGE:

4213 These functions may raise the inexact floating-point exception if the result differs in value
4214 from the argument.

4215 Ref 7.26.6

4216 On page 2182 line 69835 insert the following new *tss_**() sections:

4217 **NAME**

4218 *tss_create* — thread-specific data key creation

4219 **SYNOPSIS**

4220 `#include <threads.h>`

4221 `int tss_create(tss_t *key, tss_dtor_t dtor);`

4222 **DESCRIPTION**

4223 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4224 Any conflict between the requirements described here and the ISO C standard is
4225 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4226 The *tss_create()* function shall create a thread-specific storage pointer with destructor *dtor*,
4227 which can be null.

4228 A null pointer value shall be associated with the newly created key in all existing threads.
4229 Upon subsequent thread creation, the value associated with all keys shall be initialized to a
4230 null pointer value in the new thread.

4231 Destructors associated with thread-specific storage shall not be invoked at process
4232 termination.

4233 The behavior is undefined if the *tss_create()* function is called from within a destructor.

4234 [CX]The *tss_create()* function shall not be affected if the calling thread executes a signal
4235 handler during the call.[/CX]

4236 **RETURN VALUE**

4237 If the *tss_create()* function is successful, it shall set the thread-specific storage pointed to by
4238 *key* to a value that uniquely identifies the newly created pointer and shall return
4239 *thrd_success*; otherwise, *thrd_error* shall be returned and the thread-specific storage
4240 pointed to by *key* has an indeterminate value.

4241 **ERRORS**

4242 No errors are defined.

4243 **EXAMPLES**

4244 None.

4245 **APPLICATION USAGE**

4246 The *tss_create()* function performs no implicit synchronization. It is the responsibility of the
4247 programmer to ensure that it is called exactly once per key before use of the key.

4248 **RATIONALE**

4249 If the value associated with a key needs to be updated during the lifetime of the thread, it
4250 may be necessary to release the storage associated with the old value before the new value is
4251 bound. Although the *tss_set()* function could do this automatically, this feature is not needed
4252 often enough to justify the added complexity. Instead, the programmer is responsible for
4253 freeing the stale storage:

```
4254 old = tss_get(key);  
4255 new = allocate();  
4256 destructor(old);  
4257 tss_set(key, new);
```

4258 There is no notion of a destructor-safe function. If an application does not call *thrd_exit()* or
4259 *pthread_exit()* from a signal handler, or if it blocks any signal whose handler may call
4260 *thrd_exit()* or *pthread_exit()* while calling async-unsafe functions, all functions can be safely
4261 called from destructors.

4262 The *tss_create()* function is not affected by signal handlers for the reasons stated in [xref to
4263 XRAT B.2.3].

4264 **FUTURE DIRECTIONS**

4265 None.

4266 **SEE ALSO**

4267 *pthread_exit*, *pthread_key_create*, *thrd_exit*, *tss_delete*, *tss_get*

4268 XBD <**threads.h**>

4269 **CHANGE HISTORY**

4270 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4271 **NAME**

4272 *tss_delete* — thread-specific data key deletion

4273 **SYNOPSIS**

4274 `#include <threads.h>`

4275 `void tss_delete(tss_t key);`

4276 **DESCRIPTION**

4277 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4278 Any conflict between the requirements described here and the ISO C standard is
4279 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4280 The *tss_delete()* function shall release any resources used by the thread-specific storage
4281 identified by *key*. The thread-specific data values associated with *key* need not be null at the
4282 time *tss_delete()* is called. It is the responsibility of the application to free any application
4283 storage or perform any cleanup actions for data structures related to the deleted key or
4284 associated thread-specific data in any threads; this cleanup can be done either before or after
4285 *tss_delete()* is called.

4286 The application shall ensure that the *tss_delete()* function is only called with a value for *key*
4287 that was returned by a call to *tss_create()* before the thread commenced executing
4288 destructors.

4289 If *tss_delete()* is called while another thread is executing destructors, whether this will affect
4290 the number of invocations of the destructor associated with *key* on that thread is unspecified.

4291 The *tss_delete()* function shall be callable from within destructor functions. Calling
4292 *tss_delete()* shall not result in the invocation of any destructors. Any destructor function that
4293 was associated with *key* shall no longer be called upon thread exit.

4294 Any attempt to use *key* following the call to *tss_delete()* results in undefined behavior.

4295 [CX]The *tss_delete()* function shall not be affected if the calling thread executes a signal
4296 handler during the call.[/CX]

4297 **RETURN VALUE**

4298 This function shall not return a value.

4299 **ERRORS**

4300 No errors are defined.

4301 **EXAMPLES**

4302 None.

4303 **APPLICATION USAGE**

4304 None.

4305 **RATIONALE**

4306 A thread-specific data key deletion function has been included in order to allow the
4307 resources associated with an unused thread-specific data key to be freed. Unused thread-
4308 specific data keys can arise, among other scenarios, when a dynamically loaded module that
4309 allocated a key is unloaded.

4310 Conforming applications are responsible for performing any cleanup actions needed for data
4311 structures associated with the key to be deleted, including data referenced by thread-specific
4312 data values. No such cleanup is done by *tss_delete()*. In particular, destructor functions
4313 are not called. See the RATIONALE for *pthread_key_delete()* for the reasons for this
4314 division of responsibility.

4315 The *tss_delete()* function is not affected by signal handlers for the reasons stated in [xref to
4316 XRAT B.2.3].

4317 **FUTURE DIRECTIONS**

4318 None.

4319 **SEE ALSO**

4320 *pthread_key_create*, *tss_create*

4321 XBD <**threads.h**>

4322 **CHANGE HISTORY**

4323 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4324 **NAME**

4325 *tss_get*, *tss_set* — thread-specific data management

4326 **SYNOPSIS**

4327 `#include <threads.h>`

4328 `void *tss_get(tss_t key);`
4329 `int tss_set(tss_t key, void *val);`

4330 **DESCRIPTION**

4331 [CX] The functionality described on this reference page is aligned with the ISO C standard.
4332 Any conflict between the requirements described here and the ISO C standard is
4333 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4334 The *tss_get()* function shall return the value for the current thread held in the thread-specific

4335 storage identified by *key*.

4336 The *tss_set()* function shall set the value for the current thread held in the thread-specific
4337 storage identified by *key* to *val*. This action shall not invoke the destructor associated with
4338 the key on the value being replaced.

4339 The application shall ensure that the *tss_get()* and *tss_set()* functions are only called with a
4340 value for *key* that was returned by a call to *tss_create()* before the thread commenced
4341 executing destructors.

4342 The effect of calling *tss_get()* or *tss_set()* after *key* has been deleted with *tss_delete()* is
4343 undefined.

4344 [CX]Both *tss_get()* and *tss_set()* can be called from a thread-specific data destructor
4345 function. A call to *tss_get()* for the thread-specific data key being destroyed shall return a
4346 null pointer, unless the value is changed (after the destructor starts) by a call to *tss_set()*.
4347 Calling *tss_set()* from a thread-specific data destructor function may result either in lost
4348 storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction)
4349 or in an infinite loop.

4350 These functions shall not be affected if the calling thread executes a signal handler during
4351 the call.[/CX]

4352 **RETURN VALUE**

4353 The *tss_get()* function shall return the value for the current thread. If no thread-specific data
4354 value is associated with *key*, then a null pointer shall be returned.

4355 The *tss_set()* function shall return *thrd_success* on success or *thrd_error* if the request
4356 could not be honored.

4357 **ERRORS**

4358 No errors are defined.

4359 **EXAMPLES**

4360 None.

4361 **APPLICATION USAGE**

4362 None.

4363 **RATIONALE**

4364 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT
4365 B.2.3].

4366 **FUTURE DIRECTIONS**

4367 None.

4368 **SEE ALSO**

4369 *pthread_getspecific*, *tss_create*

4370 XBD <threads.h>

4371 **CHANGE HISTORY**

4372 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4373 Ref 7.31.11 para 2

4374 On page 2193 line 70145 section `ungetc()`, change FUTURE DIRECTIONS from:

4375 None.

4376 to:

4377 The ISO C standard states that the use of `ungetc()` on a binary stream where the file position
4378 indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction
4379 between binary and text streams, so this applies to all streams. This feature may be removed
4380 in a future version of this standard.

4381 Ref 7.29.6.3 para 1, 7.1.4 para 5

4382 On page 2242 line 71441 section `wcrtomb()`, change:

4383 [CX]The `wcrtomb()` function need not be thread-safe if called with a NULL *ps*
4384 argument.[/CX]

4385 to:

4386 If called with a null *ps* argument, the `wcrtomb()` function need not be thread-safe; however,
4387 such calls shall avoid data races with calls to `wcrtomb()` with a non-null argument and with
4388 calls to all other functions.

4389 Ref 7.29.6.4 para 1, 7.1.4 para 5

4390 On page 2266 line 72111 section `wcsrtombs()`, change:

4391 [CX]The `wcsnrtombs()` and `wcsrtombs()` functions need not be thread-safe if called with a
4392 NULL *ps* argument.[/CX]

4393 to:

4394 [CX]If called with a null *ps* argument, the `wcsnrtombs()` function need not be thread-safe;
4395 however, such calls shall avoid data races with calls to `wcsnrtombs()` with a non-null
4396 argument and with calls to all other functions.[/CX]

4397 If called with a null *ps* argument, the `wcsrtombs()` function need not be thread-safe;
4398 however, such calls shall avoid data races with calls to `wcsrtombs()` with a non-null
4399 argument and with calls to all other functions.

4400 Ref 7.22.7 para 1, 7.1.4 para 5

4401 On page 2292 line 72879 section `wctomb()`, change:

4402 [CX]The `wctomb()` function need not be thread-safe.[/CX]

4403 to:

4404 The `wctomb()` function need not be thread-safe; however, it shall avoid data races with all
4405 other functions.

4406 Changes to XCU

4407 Ref 7.22.2

4408 On page 2333 line 74167 section 1.1.2.2 Mathematical Functions, change:

4409 Section 7.20.2, Pseudo-Random Sequence Generation Functions

4410 to:

4411 Section 7.22.2, Pseudo-Random Sequence Generation Functions

4412 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4413 On page 2542 line 82220 section c99, rename the c99 page to c17.

4414 Ref 7.26

4415 On page 2545 line 82375 section c99 (now c17), change:

4416 ... , `<spawn.h>`, `<sys/socket.h>`, ...

4417 to:

4418 ... , `<spawn.h>`, `<sys/socket.h>`, `<threads.h>`, ...

4419 Ref 7.26

4420 On page 2545 line 82382 section c99 (now c17), change:

4421 This option shall make available all interfaces referenced in `<pthread.h>` and `pthread_kill()`
4422 and `pthread_sigmask()` referenced in `<signal.h>`.

4423 to:

4424 This option shall make available all interfaces referenced in `<pthread.h>` and `<threads.h>`,
4425 and also `pthread_kill()` and `pthread_sigmask()` referenced in `<signal.h>`.

4426 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4427 On page 2552-2553 line 82641-82677 section c99 (now c17), change CHANGE HISTORY to:

4428 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4429 Changes to XRAT

4430 Ref G.1 para 1

4431 On page 3483 line 117680 section A.1.7.1 Codes, add a new tagged paragraph:

4432 MXC This margin code is used to denote functionality related to the IEC 60559 Complex
4433 Floating-Point option.

4434 Ref (none)

4435 On page 3489 line 117909 section A.3 Definitions (Byte), change:

4436 alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types are now defined.

4437 to:

4438 alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types were first defined.

4439 Ref 5.1.2.4, 7.17.3

4440 On page 3515 line 118946 section A.4.12 Memory Synchronization, change:

4441 **A.4.12 Memory Synchronization**

4442 to:

4443 **A.4.12 Memory Ordering and Synchronization**

4444 *A.4.12.1 Memory Ordering*

4445 There is no additional rationale provided for this section.

4446 *A.4.12.2 Memory Synchronization*

4447 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4448 On page 3556 line 120684 section A.12.2 Utility Syntax Guidelines, change:

4449 Thus, they had to devise a new name, *c89* (now superseded by *c99*), rather than ...

4450 to:

4451 Thus, they had to devise a new name, *c89* (subsequently superseded by *c99* and now by

4452 *c17*), rather than ...

4453 Ref K.3.1.1

4454 On page 3567 line 121053 section B.2.2.1 POSIX.1 Symbols, add a new unnumbered subsection:

4455 **The `__STDC_WANT_LIB_EXT1__` Feature Test Macro**

4456 The ISO C standard specifies the feature test macro `__STDC_WANT_LIB_EXT1__` as the

4457 announcement mechanism for the application that it requires functionality from Annex K. It

4458 specifies that the symbols specified in Annex K (if supported) are made visible when

4459 `__STDC_WANT_LIB_EXT1__` is 1 and are not made visible when it is 0, but leaves it

4460 unspecified whether they are made visible when `__STDC_WANT_LIB_EXT1__` is

4461 undefined. POSIX.1 requires that they are not made visible when the macro is undefined

4462 (except for those symbols that are already explicitly allowed to be visible through the

4463 definition of `_POSIX_C_SOURCE` or `_XOPEN_SOURCE`, or both).

4464 POSIX.1 does not include the interfaces specified in Annex K of the ISO C standard, but

4465 allows the symbols to be made visible in headers when requested by the application in order

4466 that applications can use symbols from Annex K and symbols from POSIX.1 in the same

4467 translation unit.

4468 Ref 6.10.3.4

4469 On page 3570 line 121176 section B.2.2.2 The Name Space, change:

4470 as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C
4471 standard

4472 to:

4473 as described for macros that expand to their own name as in Section 6.10.3.4 of the ISO C
4474 standard

4475 Ref 7.5 para 2
4476 On page 3571 line 121228-121243 section B.2.3 Error Numbers, change:

4477 The ISO C standard requires that *errno* be an assignable lvalue. Originally, ...
4478 [...]
4479 ... using the return value for a mixed purpose was judged to be of limited use and
4480 error prone.

4481 to:

4482 The original ISO C standard just required that *errno* be an modifiable lvalue. Since the
4483 introduction of threads in 2011, the ISO C standard has instead required that *errno* be a
4484 macro which expands to a modifiable lvalue that has thread local storage duration.

4485 Ref 7.26
4486 On page 3575 line 121390 section B.2.3 Error Numbers, change:

4487 In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a
4488 special case since it will never occur.

4489 to:

4490 In particular, applications calling blocking interfaces need not handle any possible [EINTR]
4491 return as a special case since it will never occur. In the case of threads functions in
4492 <threads.h>, the requirement is stated in terms of the call not being affected if the calling
4493 thread executes a signal handler during the call, since these functions return errors in a
4494 different way and cannot distinguish an [EINTR] condition from other error conditions.

4495 Ref (none)
4496 On page 3733 line 128128 section C.2.6.4 Arithmetic Expansion, change:

4497 Although the ISO/IEC 9899: 1999 standard now requires support for ...

4498 to:

4499 Although the ISO C standard requires support for ...

4500 Ref 7.17
4501 On page 3789 line 129986 section E.1 Subprofiling Option Groups, change:

4502 by collecting sets of related functions

4503 to:

4504 by collecting sets of related functions and generic functions

4505 Ref 7.22.3.1, 7.27.2.5, 7.22.4

4506 On page 3789, 3792 line 130022-130032, 130112-130114 section E.1 Subprofiling Option Groups,

4507 add new functions (in sorted order) to the existing groups as indicated:

4508 POSIX_C_LANG_SUPPORT

4509 *aligned_alloc()*, *timespec_get()*

4510 POSIX_MULTI_PROCESS

4511 *at_quick_exit()*, *quick_exit()*

4512 Ref 7.17

4513 On page 3789 line 129991 section E.1 Subprofiling Option Groups, add:

4514 POSIX_C_LANG_ATOMICS: ISO C Atomic Operations

4515 *atomic_compare_exchange_strong()*, *atomic_compare_exchange_strong_explicit()*,

4516 *atomic_compare_exchange_weak()*, *atomic_compare_exchange_weak_explicit()*,

4517 *atomic_exchange()*, *atomic_exchange_explicit()*, *atomic_fetch_add()*,

4518 *atomic_fetch_add_explicit()*, *atomic_fetch_and()*, *atomic_fetch_and_explicit()*,

4519 *atomic_fetch_or()*, *atomic_fetch_or_explicit()*, *atomic_fetch_sub()*,

4520 *atomic_fetch_sub_explicit()*, *atomic_fetch_xor()*, *atomic_fetch_xor_explicit()*,

4521 *atomic_flag_clear()*, *atomic_flag_clear_explicit()*, *atomic_flag_test_and_set()*,

4522 *atomic_flag_test_and_set_explicit()*, *atomic_init()*, *atomic_is_lock_free()*,

4523 *atomic_load()*, *atomic_load_explicit()*, *atomic_signal_fence()*,

4524 *atomic_thread_fence()*, *atomic_store()*, *atomic_store_explicit()*, *kill_dependency()*

4525 Ref 7.26

4526 On page 3790 line 1300349 section E.1 Subprofiling Option Groups, add:

4527 POSIX_C_LANG_THREADS: ISO C Threads

4528 *call_once()*, *cnd_broadcast()*, *cnd_signal()*, *cnd_destroy()*, *cnd_init()*,

4529 *cnd_timedwait()*, *cnd_wait()*, *mtx_destroy()*, *mtx_init()*, *mtx_lock()*, *mtx_timedlock()*,

4530 *mtx_trylock()*, *mtx_unlock()*, *thrd_create()*, *thrd_current()*, *thrd_detach()*,

4531 *thrd_equal()*, *thrd_exit()*, *thrd_join()*, *thrd_sleep()*, *thrd_yield()*, *tss_create()*,

4532 *tss_delete()*, *tss_get()*, *tss_set()*

4533 POSIX_C_LANG_UCHAR: ISO C Unicode Utilities

4534 *c16rtomb()*, *c32rtomb()*, *mbrtoc16()*, *mbrtoc32()*