

# 1 **TODO**

2 | **Check for overlaps with Mantis bugs: 374 and 1218 (once resolved; NB 374 may also affect**  
3 | **aligned\_alloc()), and any that get tagged tc3 or issue8 after ~~2020-10-29~~2021-08-12.**

## 4 **Introduction**

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

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

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

## 12 **Global Change**

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

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

## 17 **Changes to XBD**

18 Ref G.1 para 1

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

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

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

23 Ref (none)

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

25 the ISO/IEC 9899: 1999 standard

26 to:

27 the ISO C standard

28 Ref 6.2.8

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

30 See also the ISO C standard, Section B3.

31 to:

32 See also the ISO C standard, Section 6.2.8.

33 Ref 5.1.2.4

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

### 35 **3.31 Atomic Operation**

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

40 Ref 7.26.3

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

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

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

49 Ref 5.1.2.4

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

### 51 **3.125 Data Race**

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

55 Ref 5.1.2.4

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

### 57 **3.215 Lock-Free Operation**

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

60 Ref 7.26.5.1

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

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

64 to:

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

67 Ref 7.26.4

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

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

73 **Note:** The *pthread\_mutex\_init()* and *mtx\_init()* functions are defined in detail in the System  
74 Interfaces volume of POSIX.1-20xx.

75 Ref 7.26.5.5

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

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

79 to:

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

82 Ref 7.26.5.1

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

84 if the process attempts to create additional threads using *pthread\_create()* or  
85 SIGEV\_THREAD notifications

86 to:

87 if the process attempts to create additional threads using *pthread\_create()*, *thrd\_create()*, or  
88 SIGEV\_THREAD notifications

89 Ref 5.1.2.4

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

### 91 **3.382 Synchronization Operation**

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

93 Ref 7.26.5.1

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

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

97 to:

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

- 100 • The initial thread.

- 101 • A thread created using *pthread\_create()*.
- 102 • A thread created using *thrd\_create()*.
- 103 • A thread created via a SIGEV\_THREAD notification.

104 **Note:** Since *pthread\_create()* returns an ID of type **pthread\_t** and *thrd\_create()* returns an ID of  
105 type **thrd\_t**, this uniqueness requirement necessitates that these two types are defined as the  
106 same underlying type because calls to *pthread\_self()* and *thrd\_current()* from the initial  
107 thread need to return the same thread ID. The *pthread\_create()*, *pthread\_self()*, *thrd\_create()*  
108 and *thrd\_current()* functions and SIGEV\_THREAD notifications are defined in detail in the  
109 System Interfaces volume of POSIX.1-20xx.

110 Ref 5.1.2.4

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

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

114 to:

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

117 Ref 5.1.2.4

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

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

122 Ref 7.26.6

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

124 A process global handle of type **pthread\_key\_t** which is used for naming thread-specific  
125 data.

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

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

131 to:

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

137 Although the same key value can be used by different threads, the values bound to the key  
138 by *pthread\_setspecific()* for keys of type **pthread\_key\_t**, and by *tss\_set()* for keys of type  
139 **tss\_t**, are maintained on a per-thread basis and persist for the life of the calling thread.

140           **Note:** The *pthread\_key\_create()*, *pthread\_setspecific()*, *tss\_create()* and *tss\_set()* functions are  
141           defined in detail in the System Interfaces volume of POSIX.1-20xx.

142 Ref 5.1.2.4, 7.17.3

143 | On page 111 line 3060 section 4.12 Memory Synchronization, [after applying bug 1426](#) change:

## 144 **4.12 Memory Synchronization**

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

151 to:

## 152 **4.12 Memory Ordering and Synchronization**

### 153 **4.12.1 Memory Ordering**

#### 154 *4.12.1.1 Data Races*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#### 278 4.12.1.2 Memory Order and Consistency

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

283 For `memory_order_relaxed`, no operation orders memory.

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

#### 336 4.12.2 Memory Synchronization

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

343 Ref 7.26.3, 7.26.4

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

346 |            `cond_broadcast()`                    ~~`mtx_lock()`~~                    ~~`thrd_create()`~~  
347 |            `cond_signal()`                    ~~`mtx_timedlock()`~~                    ~~`thrd_join()`~~  
348 |            ~~`end_timedwait()`~~                    ~~`mtx_trylock()`~~  
349 |            ~~`end_wait()`~~                    ~~`mtx_unlock()`~~

350 | Ref 7.26.2.1, 7.26.4

351 | On page 111 line 3076 section 4.12 Memory Synchronization, [after applying bugs 1216 and 1426](#)  
352 | change:

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

358 |            The `pthread_mutex_clocklock()`, `pthread_mutex lock()`,  
359 |            [RPP|TPP]`pthread_mutex setprioceiling()`, [TPP|TPP] `pthread_mutex timedlock()`, and  
360 |            `pthread_mutex trylock()` functions shall synchronize memory on all calls that acquire the  
361 |            mutex, including those that return [EOWNERDEAD]. The `pthread_mutex unlock()` function  
362 |            shall synchronize memory on all calls that release the mutex.

363 |            **Note:** If the mutex type is PTHREAD\_MUTEX\_RECURSIVE, calls to the locking functions do  
364 |            not acquire the mutex if the calling thread already owns it, and calls to  
365 |            `pthread_mutex unlock()` do not release the mutex if it has a lock count greater than one.

366 |            The `pthread_cond clockwait()`, `pthread_cond wait()`, and `pthread_cond timedwait()`  
367 |            functions shall synchronize memory on all calls that release and re-acquire the specified  
368 |            mutex, including calls that return [EOWNERDEAD], both when the mutex is released and  
369 |            when it is re-acquired.

370 |            **Note:** If the mutex type is PTHREAD\_MUTEX\_RECURSIVE, calls to `pthread_cond clockwait()`,  
371 |            `pthread_cond wait()`, and `pthread_cond timedwait()` do not release and re-acquire the mutex  
372 |            if it has a lock count greater than one.

373 |            ~~The `pthread_mutex_lock()` function need not synchronize memory if the mutex type if~~  
374 |            ~~PTHREAD\_MUTEX\_RECURSIVE and the calling thread already owns the mutex. The~~  
375 |            ~~`pthread_mutex_unlock()` function need not synchronize memory if the mutex type is~~  
376 |            ~~PTHREAD\_MUTEX\_RECURSIVE and the mutex has a lock count greater than one.~~

377 | to:

378 |            The `pthread_once()` and `call_once()` functions shall synchronize memory for the first  
379 |            **successful** call in each thread for a given **pthread\_once\_t** or **once\_flag** object, respectively.  
380 |            If the `init_routine` called by `pthread_once()` or `call_once()` is a cancellation point and is  
381 |            canceled, a **successful** call to `pthread_once()` for the same **pthread\_once\_t** object, or to  
382 |            `call_once()` for the same **once\_flag** object, made from a cancellation cleanup handler shall  
383 |            also synchronize memory.

384 |            The `pthread_mutex clocklock()`, `pthread_mutex lock()`,  
385 |            [RPP|TPP]`pthread_mutex setprioceiling()`, [TPP|TPP] `pthread_mutex timedlock()`, and  
386 |            `pthread_mutex trylock()` functions shall synchronize memory on all calls that acquire the  
387 |            mutex, including those that return [EOWNERDEAD]. The `pthread_mutex unlock()` function  
388 |            shall synchronize memory on all calls that release the mutex.

389 Note: If the mutex type is PTHREAD\_MUTEX\_RECURSIVE, calls to the locking functions do  
390 not acquire the mutex if the calling thread already owns it, and calls to  
391 *pthread\_mutex\_unlock()* do not release the mutex if it has a lock count greater than one.

392 The *pthread\_cond\_clockwait()*, *pthread\_cond\_wait()*, and *pthread\_cond\_timedwait()*  
393 functions shall synchronize memory on all calls that release and re-acquire the specified  
394 mutex, including calls that return [EOWNERDEAD], both when the mutex is released and  
395 when it is re-acquired.

396 Note: If the mutex type is PTHREAD\_MUTEX\_RECURSIVE, calls to *pthread\_cond\_clockwait()*,  
397 *pthread\_cond\_wait()*, and *pthread\_cond\_timedwait()* do not release and re-acquire the mutex  
398 if it has a lock count greater than one.

399 The *mtx\_lock()*, *mtx\_timedlock()*, and *mtx\_trylock()* functions shall synchronize memory on  
400 all calls that acquire the mutex. The *mtx\_unlock()* function shall synchronize memory on all  
401 calls that release the mutex.

402 Note: If the mutex is a recursive mutex, calls to the locking functions do not acquire the mutex if  
403 the calling thread already owns it, and calls to *mtx\_unlock()* do not release the mutex if it has  
404 a lock count greater than one.

405 The *cond\_wait()* and *cond\_timedwait()* functions shall synchronize memory on all calls that  
406 release and re-acquire the specified mutex, both when the mutex is released and when it is  
407 re-acquired.

408 Note: If the mutex is a recursive mutex, calls to *cond\_wait()* and *cond\_timedwait()* do not release and  
409 re-acquire the mutex if it has a lock count greater than one.

410 ~~The *pthread\_mutex\_lock()* and *thrd\_lock()* functions, and their related “timed” and “try”~~  
411 ~~variants, need not synchronize memory if the mutex is a recursive mutex and the calling~~  
412 ~~thread already owns the mutex. The *pthread\_mutex\_unlock()* and *thrd\_unlock()* functions~~  
413 ~~need not synchronize memory if the mutex is a recursive mutex and has a lock count greater~~  
414 ~~than one.~~

415 Ref 7.26.4

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

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

420 Ref 7.12.1 para 7

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

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

424 to:

425 The error conditions defined for all functions in the <math.h> header are domain, pole and  
426 range errors, described below. If a domain, pole, or range error occurs and the integer  
427 expression (*math\_errhandling* & MATH\_ERRNO) is zero, then *errno* shall either be set to  
428 the value corresponding to the error, as specified below, or be left unmodified. If no such

429 error occurs, *errno* shall be left unmodified regardless of the setting of *math\_errhandling*.

430 Ref 7.12.1 para 3

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

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

434 to:

435 A ``pole error'' shall occur if the mathematical result of the function has an exact infinite  
436 result as the finite input argument(s) are approached in the limit (for example,  $\log(0.0)$ ). The  
437 description of each function lists any required pole errors; an implementation may define  
438 additional pole errors, provided that such errors are consistent with the mathematical  
439 definition of the function.

440 Ref 7.12.1 para 4

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

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

444 add:

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

448 Ref 7.29.1 para 5

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

450 Arguments to the functions declared in the `<wchar.h>` header can point to arrays containing  
451 **wchar\_t** values that do not correspond to valid wide character codes according to the  
452 *LC\_CTYPE* category of the locale being used. Such values shall be processed according to  
453 the specified semantics for the function in the System Interfaces volume of POSIX.1-20xx,  
454 except that it is unspecified whether an encoding error occurs if such a value appears in the  
455 format string of a function that has a format string as a parameter and the specified  
456 semantics do not require that value to be processed as if by *wcrtomb()*.

457 Ref 7.3.1 para 2

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

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

464 Ref G.6 para 1

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

466 The macros `imaginary` and `_Imaginary_I` shall be defined if and only if the implementation

467 supports imaginary types.

468 add:

469 [MXC]Implementations that support the IEC 60559 Complex Floating-Point option shall  
470 define the macros `imaginary` and `_Imaginary_I`, and the macro `I` shall expand to  
471 `_Imaginary_I`. [MXC]

472 Ref 7.3.9.3

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

474 The following shall be defined as macros.

```
475 double complex      CMPLX(double x, double y);  
476 float complex      CMPLXF(float x, float y);  
477 long double complex CMPLXL(long double x, long double y);
```

478 Ref 7.3.1 para 2

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

480 The `<complex.h>` header is optional in the ISO C standard but is mandated by POSIX.1-  
481 20xx. Note however that subprofiles can choose to make this header optional (see [xref to  
482 2.1.5.1 Subprofiling Considerations]), and therefore application portability to subprofile  
483 implementations would benefit from checking whether `__STDC_NO_COMPLEX__` is  
484 defined before inclusion of `<complex.h>`.

485 Ref 7.3.9.3

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

487 Ref 7.5 para 2

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

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

492 to:

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

495 Ref (none)

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

497 the ISO/IEC 9899: 1999 standard

498 to:

499 the ISO C standard

500 Ref 5.2.4.2.2 para 11

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

502 The presence or absence of subnormal numbers is characterized by the implementation-

503 defined values of FLT\_HAS\_SUBNORM , DBL\_HAS\_SUBNORM , and  
504 LDBL\_HAS\_SUBNORM :

-1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

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

509 Ref 5.2.4.2.2 para 12

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

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

514 [math stuff]

515 FLT\_DECIMAL\_DIG 6

516 DBL\_DECIMAL\_DIG 10

517 LDBL\_DECIMAL\_DIG 10

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

520 Ref 5.2.4.2.2 para 14

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

522 Minimum positive floating-point number.

523 FLT\_TRUE\_MIN 1E-37

524 DBL\_TRUE\_MIN 1E-37

525 LDBL\_TRUE\_MIN 1E-37

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

529 Ref (none)

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

531 the ISO/IEC 9899: 1999 standard

532 to:

533 the ISO C standard

534 Ref 7.22.4.3

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

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

537 to:

538 Maximum number of functions that can be registered with *atexit()* or *at\_quick\_exit()*. The  
539 limit shall apply independently to each function.

540 Ref 5.2.4.2.1 para 2

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

542 If the value of an object of type **char** is treated as a signed integer when used in an  
543 expression, the value of {CHAR\_MIN} is the same as that of {SCHAR\_MIN} and the value  
544 of {CHAR\_MAX} is the same as that of {SCHAR\_MAX}. Otherwise, the value of  
545 {CHAR\_MIN} is 0 and the value of {CHAR\_MAX} is the same as that of  
546 {UCHAR\_MAX}.

547 to:

548 If an object of type **char** can hold negative values, the value of {CHAR\_MIN} shall be the  
549 same as that of {SCHAR\_MIN} and the value of {CHAR\_MAX} shall be the same as that  
550 of {SCHAR\_MAX}. Otherwise, the value of {CHAR\_MIN} shall be 0 and the value of  
551 {CHAR\_MAX} shall be the same as that of {UCHAR\_MAX}.

552 Ref (none)

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

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

555 to:

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

557 Ref 7.26.5.5

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

559 void pthread\_exit(void \*);

560 to:

561 \_Noreturn void pthread\_exit(void \*);

562 Ref 7.13.2.1 para 1

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

564 void longjmp(jmp\_buf, int);

565 [CX]void siglongjmp(sigjmp\_buf, int);[/CX]

566 to:

```
567     _Noreturn void longjmp(jmp_buf, int);  
568     [CX]_Noreturn void siglongjmp(sigjmp_buf, int);[/CX]
```

569 Ref 7.15

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

571 **NAME**

572 stdalign.h — alignment macros

573 **SYNOPSIS**

574 #include <stdalign.h>

575 **DESCRIPTION**

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

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

580 alignas Expands to **\_Alignas**

581 alignof Expands to **\_Alignof**

582 \_\_alignas\_is\_defined

583 Expands to the integer constant 1

584 \_\_alignof\_is\_defined

585 Expands to the integer constant 1

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

588 **APPLICATION USAGE**

589 None.

590 **RATIONALE**

591 None.

592 **FUTURE DIRECTIONS**

593 None.

594 **SEE ALSO**

595 None.

596 **CHANGE HISTORY**

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

598 Ref 7.17, 7.31.8 para 2

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



600 **NAME**

601           stdatomic.h — atomics

602 **SYNOPSIS**

603           #include <stdatomic.h>

604 **DESCRIPTION**

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

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

610           The <stdatomic.h> header shall define the **atomic\_flag** type as a structure type. This type  
611           provides the classic test-and-set functionality. It shall have two states, set and clear.  
612           Operations on an object of type **atomic\_flag** shall be lock free.

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

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

Atomic type name	Direct type
<b>atomic_bool</b>	<b>_Atomic_Bool</b>
<b>atomic_char</b>	<b>_Atomic char</b>
<b>atomic_schar</b>	<b>_Atomic signed char</b>
<b>atomic_uchar</b>	<b>_Atomic unsigned char</b>
<b>atomic_short</b>	<b>_Atomic short</b>
<b>atomic_ushort</b>	<b>_Atomic unsigned short</b>
<b>atomic_int</b>	<b>_Atomic int</b>
<b>atomic_uint</b>	<b>_Atomic unsigned int</b>
<b>atomic_long</b>	<b>_Atomic long</b>
<b>atomic_ulong</b>	<b>_Atomic unsigned long</b>
<b>atomic_llong</b>	<b>_Atomic long long</b>
<b>atomic_ullong</b>	<b>_Atomic unsigned long long</b>
<b>atomic_char16_t</b>	<b>_Atomic char16_t</b>
<b>atomic_char32_t</b>	<b>_Atomic char32_t</b>
<b>atomic_wchar_t</b>	<b>_Atomic wchar_t</b>
<b>atomic_int_least8_t</b>	<b>_Atomic int_least8_t</b>
<b>atomic_uint_least8_t</b>	<b>_Atomic uint_least8_t</b>
<b>atomic_int_least16_t</b>	<b>_Atomic int_least16_t</b>
<b>atomic_uint_least16_t</b>	<b>_Atomic uint_least16_t</b>
<b>atomic_int_least32_t</b>	<b>_Atomic int_least32_t</b>
<b>atomic_uint_least32_t</b>	<b>_Atomic uint_least32_t</b>
<b>atomic_int_least64_t</b>	<b>_Atomic int_least64_t</b>
<b>atomic_uint_least64_t</b>	<b>_Atomic uint_least64_t</b>
<b>atomic_int_fast8_t</b>	<b>_Atomic int_fast8_t</b>
<b>atomic_uint_fast8_t</b>	<b>_Atomic uint_fast8_t</b>

<code>atomic_int_fast16_t</code>	<code>_Atomic int_fast16_t</code>
<code>atomic_uint_fast16_t</code>	<code>_Atomic uint_fast16_t</code>
<code>atomic_int_fast32_t</code>	<code>_Atomic int_fast32_t</code>
<code>atomic_uint_fast32_t</code>	<code>_Atomic uint_fast32_t</code>
<code>atomic_int_fast64_t</code>	<code>_Atomic int_fast64_t</code>
<code>atomic_uint_fast64_t</code>	<code>_Atomic uint_fast64_t</code>
<code>atomic_intptr_t</code>	<code>_Atomic intptr_t</code>
<code>atomic_uintptr_t</code>	<code>_Atomic uintptr_t</code>
<code>atomic_size_t</code>	<code>_Atomic size_t</code>
<code>atomic_ptrdiff_t</code>	<code>_Atomic ptrdiff_t</code>
<code>atomic_intmax_t</code>	<code>_Atomic intmax_t</code>
<code>atomic_uintmax_t</code>	<code>_Atomic uintmax_t</code>

618 The `<stdatomic.h>` header shall define the **memory\_order** type as an enumerated type  
619 whose enumerators shall include at least the following:

```
620 memory_order_relaxed
621 memory_order_consume
622 memory_order_acquire
623 memory_order_release
624 memory_order_acq_rel
625 memory_order_seq_cst
```

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

```
627 ATOMIC_BOOL_LOCK_FREE
628 ATOMIC_CHAR_LOCK_FREE
629 ATOMIC_CHAR16_T_LOCK_FREE
630 ATOMIC_CHAR32_T_LOCK_FREE
631 ATOMIC_WCHAR_T_LOCK_FREE
632 ATOMIC_SHORT_LOCK_FREE
633 ATOMIC_INT_LOCK_FREE
634 ATOMIC_LONG_LOCK_FREE
635 ATOMIC_LLONG_LOCK_FREE
636 ATOMIC_POINTER_LOCK_FREE
```

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

642 The `<stdatomic.h>` header shall define the macro `ATOMIC_FLAG_INIT` which shall  
643 expand to an initializer for an object of type **atomic\_flag**. This macro shall initialize an  
644 **atomic\_flag** to the clear state. An **atomic\_flag** that is not explicitly initialized with  
645 `ATOMIC_FLAG_INIT` is initially in an indeterminate state.

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

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

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

```
656 _Bool    atomic_compare_exchange_strong(volatile A *, C *, C);
657 _Bool    atomic_compare_exchange_strong_explicit(volatile A *,
658          C *, C, memory_order, memory_order);
659 _Bool    atomic_compare_exchange_weak(volatile A *, C *, C);
660 _Bool    atomic_compare_exchange_weak_explicit(volatile A *, C *,
661          C, memory_order, memory_order);
662 C        atomic_exchange(volatile A *, C);
663 C        atomic_exchange_explicit(volatile A *, C, memory_order);
664 C        atomic_fetch_add(volatile A *, M);
665 C        atomic_fetch_add_explicit(volatile A *, M,
666          memory_order);
667 C        atomic_fetch_and(volatile A *, M);
668 C        atomic_fetch_and_explicit(volatile A *, M,
669          memory_order);
670 C        atomic_fetch_or(volatile A *, M);
671 C        atomic_fetch_or_explicit(volatile A *, M, memory_order);
672 C        atomic_fetch_sub(volatile A *, M);
673 C        atomic_fetch_sub_explicit(volatile A *, M,
674          memory_order);
675 C        atomic_fetch_xor(volatile A *, M);
676 C        atomic_fetch_xor_explicit(volatile A *, M,
677          memory_order);
678 void     atomic_init(volatile A *, C);
679 _Bool    atomic_is_lock_free(const volatile A *);
680 C        atomic_load(const volatile A *);
681 C        atomic_load_explicit(const volatile A *, memory_order);
682 void     atomic_store(volatile A *, C);
683 void     atomic_store_explicit(volatile A *, C, memory_order);
```

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

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

```
690 void     atomic_flag_clear(volatile atomic_flag *);
691 void     atomic_flag_clear_explicit(volatile atomic_flag *,
692          memory_order);
693 _Bool    atomic_flag_test_and_set(volatile atomic_flag *);
694 _Bool    atomic_flag_test_and_set_explicit(
695          volatile atomic_flag *, memory_order);
696 void     atomic_signal_fence(memory_order);
697 void     atomic_thread_fence(memory_order);
```

## 698 APPLICATION USAGE

699 None.

700 **RATIONALE**

701 Since operations on the **atomic\_flag** type are lock free, the operations should also be  
702 address-free. No other type requires lock-free operations, so the **atomic\_flag** type is the  
703 minimum hardware-implemented type needed to conform to this standard. The remaining  
704 types can be emulated with **atomic\_flag**, though with less than ideal properties.

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

708 **FUTURE DIRECTIONS**

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

711 **SEE ALSO**

712 Section 4.12.1

713 *XSH `atomic_compare_exchange_strong()`, `atomic_compare_exchange_weak()`,*  
714 *`atomic_exchange()`, `atomic_fetch_key()`, `atomic_flag_clear()`, `atomic_flag_test_and_set()`,*  
715 *`atomic_init()`, `atomic_is_lock_free()`, `atomic_load()`, `atomic_signal_fence()`, `atomic_store()`,*  
716 *`atomic_thread_fence()`, `kill_dependency()`.*

717 **CHANGE HISTORY**

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

719 Ref 7.31.9

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

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

722 Ref 7.19 para 2

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

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

725 Ref (none)

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

727 the ISO/IEC 9899: 1999 standard

728 to:

729 the ISO C standard

730 Ref 7.20.1.1 para 1

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

732 denotes a signed integer type

733 to:

734 denotes such a signed integer type

735 Ref 7.20.1.1 para 2

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

737 ... designates an unsigned integer type with width *N*. Thus, **uint24\_t** denotes an unsigned  
738 integer type ...

739 to:

740 ... designates an unsigned integer type with width *N* and no padding bits. Thus, **uint24\_t**  
741 denotes such an unsigned integer type ...

742 Ref 7.21.1 para 2

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

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

746 to:

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

749 Ref 7.21.1 para 3

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

751 There is a conflict between the ISO C standard and the POSIX definition of the  
752 {TMP\_MAX} macro that is addressed by ISO/IEC 9899: 1999 standard, Defect Report 336.  
753 The POSIX standard is in alignment with the public record of the response to the Defect  
754 Report. This change has not yet been published as part of the ISO C standard.

755 to:

756 None.

757 Ref 7.22.4.5 para 1

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

759 void \_Exit(int);

760 to:

761 \_Noreturn void \_Exit(int);

762 Ref 7.22.4.1 para 1

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

764 void abort(void);

765 to:

766 `_Noreturn void abort(void);`

767 Ref 7.22.3.1, 7.22.4.3

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

769 `void *aligned_alloc(size_t, size_t);`  
770 `int at_quick_exit(void (*)(void));`

771 Ref 7.22.4.4 para 1

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

773 `void exit(int);`

774 to:

775 `_Noreturn void exit(int);`

776 Ref 7.22.4.7

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

778 `_Noreturn void quick_exit(int);`

779 Ref 7.23

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

## 781 **NAME**

782 `stdnoreturn.h` — noreturn macro

## 783 **SYNOPSIS**

784 `#include <stdnoreturn.h>`

## 785 **DESCRIPTION**

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

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

## 791 **APPLICATION USAGE**

792 None.

## 793 **RATIONALE**

794 None.

## 795 **FUTURE DIRECTIONS**

796 None.

## 797 **SEE ALSO**

798 None.

## 799 **CHANGE HISTORY**

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

801 Ref G.7

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

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

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

812 *cos(iy)* = *cosh(y)*  
813 *sin(iy)* = *i sinh(y)*  
814 *tan(iy)* = *i tanh(y)*  
815 *cosh(iy)* = *cos(y)*  
816 *sinh(iy)* = *i sin(y)*  
817 *tanh(iy)* = *i tan(y)*  
818 *asin(iy)* = *i asinh(y)*  
819 *atan(iy)* = *i atanh(y)*  
820 *asinh(iy)* = *i asin(y)*  
821 *atanh(iy)* = *i atan(y)*  
822 [/MXC]

823 Ref (none)

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

825 the ISO/IEC 9899: 1999 standard

826 to:

827 the ISO C standard

828 Ref 7.26

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

830 **NAME**

831 threads.h — ISO C threads

832 **SYNOPSIS**

833 #include <threads.h>

834 **DESCRIPTION**

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

838 [CX] Implementations shall not define the macro `__STDC_NO_THREADS__`, except for  
839 profile implementations that define `_POSIX_SUBPROFILE` (see [xref to 2.1.5.1

840 Subprofiling Considerations]) in `<unistd.h>`, which may define `__STDC_NO_THREADS__`  
841 and, if they do so, need not provide this header nor support any of its facilities. [/CX]

842 The `<threads.h>` header shall define the following macros:

843 `thread_local` Expands to `_Thread_local`.

844 `ONCE_FLAG_INIT` Expands to a value that can be used to initialize an object of  
845 type `once_flag`.

846 `TSS_DTOR_ITERATIONS` Expands to an integer constant expression representing the  
847 maximum number of times that destructors will be called  
848 when a thread terminates and shall be suitable for use in `#if`  
849 preprocessing directives.

850 [CX]If `{PTHREAD_DESTRUCTOR_ITERATIONS}` is defined in `<limits.h>`, the value of  
851 `TSS_DTOR_ITERATIONS` shall be equal to  
852 `{PTHREAD_DESTRUCTOR_ITERATIONS}`; otherwise, the value of  
853 `TSS_DTOR_ITERATIONS` shall be greater than or equal to the value of  
854 `{_POSIX_THREAD_DESTRUCTOR_ITERATIONS}` and shall be less than or equal to the  
855 maximum positive value that can be returned by a call to  
856 `sysconf(_SC_THREAD_DESTRUCTOR_ITERATIONS)` in any process. [/CX]

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

861 The `<threads.h>` header shall define the enumeration constants `mtx_plain`,  
862 `mtx_recursive`, `mtx_timed`, `thrd_busy`, `thrd_error`, `thrd_nomem`, `thrd_success`  
863 and `thrd_timedout`.

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

```
866 void      call_once(once_flag *, void (*)(void));
867 int       cnd_broadcast(cnd_t *);
868 void      cnd_destroy(cnd_t *);
869 int       cnd_init(cnd_t *);
870 int       cnd_signal(cnd_t *);
871 int       cnd_timedwait(cnd_t * restrict, mtx_t * restrict,
872                        const struct timespec * restrict);
873 int       cnd_wait(cnd_t *, mtx_t *);
874 void      mtx_destroy(mtx_t *);
875 int       mtx_init(mtx_t *, int);
876 int       mtx_lock(mtx_t *);
877 int       mtx_timedlock(mtx_t * restrict,
878                        const struct timespec * restrict);
879 int       mtx_trylock(mtx_t *);
880 int       mtx_unlock(mtx_t *);
881 int       thrd_create(thrd_t *, thrd_start_t, void *);
882 thrd_t    thrd_current(void);
883 int       thrd_detach(thrd_t);
884 int       thrd_equal(thrd_t, thrd_t);
```



```

885     _Noreturn void   thrd_exit(int);
886     int             thrd_join(thrd_t, int *);
887     int             thrd_sleep(const struct timespec *,
888                             struct timespec *);
889     void            thrd_yield(void);
890     int             tss_create(tss_t *, tss_dtor_t);
891     void            tss_delete(tss_t);
892     void            *tss_get(tss_t);
893     int             tss_set(tss_t, void *);

```

894 Inclusion of the `<threads.h>` header shall make symbols defined in the header `<time.h>`  
895 visible.

## 896 APPLICATION USAGE

897 The `<threads.h>` header is optional in the ISO C standard but is mandated by POSIX.1-  
898 20xx. Note however that subprofiles can choose to make this header optional (see [xref to  
899 2.1.5.1 Subprofiling Considerations]), and therefore application portability to subprofile  
900 implementations would benefit from checking whether `__STDC_NO_THREADS__` is  
901 defined before inclusion of `<threads.h>`.

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

## 908 RATIONALE

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

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

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

931 **FUTURE DIRECTIONS**

932 None.

933 **SEE ALSO**

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

935 XSH Section 2.9, *call\_once()*, *cnd\_broadcast()*, *cnd\_destroy()*, *cnd\_timedwait()*,  
936 *mtx\_destroy()*, *mtx\_lock()*, *sysconf()*, *thrd\_create()*, *thrd\_current()*, *thrd\_detach()*,  
937 *thrd\_equal()*, *thrd\_exit()*, *thrd\_join()*, *thrd\_sleep()*, *thrd\_yield()*, *tss\_create()*, *tss\_delete()*,  
938 *tss\_get()*.

939 **CHANGE HISTORY**

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

941 Ref 7.27.1 para 4

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

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

945	<code>time_t</code>	<code>tv_sec</code>	Seconds.
946	<code>long</code>	<code>tv_nsec</code>	Nanoseconds.

947 and change the members to:

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

950 Ref 7.27.1 para 2

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

952	<code>TIME_UTC</code>	An integer constant greater than 0 that designates the UTC time base 953 in calls to <i>timespec_get()</i> . The value shall be suitable for use in <b>#if</b> 954 preprocessing directives.
-----	-----------------------	--

955 Ref 7.27.2.5

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

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

958 Ref 7.28

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

960 **NAME**

961 `uchar.h` — Unicode character handling

962 **SYNOPSIS**

963 `#include <uchar.h>`

964 **DESCRIPTION**

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

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

968 The `<uchar.h>` header shall define the following types:

969 **mbstate\_t** As described in `<wchar.h>`.

970 **size\_t** As described in `<stddef.h>`.

971 **char16\_t** The same type as **uint\_least16\_t**, described in `<stdint.h>`.

972 **char32\_t** The same type as **uint\_least32\_t**, described in `<stdint.h>`.

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

```
975 size_t    c16rtomb(char *restrict, char16_t,  
976             mbstate_t *restrict);  
977 size_t    c32rtomb(char *restrict, char32_t,  
978             mbstate_t *restrict);  
979 size_t    mbrtoc16(char16_t *restrict, const char *restrict,  
980             size_t, mbstate_t *restrict);  
981 size_t    mbrtoc32(char32_t *restrict, const char *restrict,  
982             size_t, mbstate_t *restrict);
```

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

## 985 APPLICATION USAGE

986 None.

## 987 RATIONALE

988 None.

## 989 FUTURE DIRECTIONS

990 None.

## 991 SEE ALSO

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

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

## 994 CHANGE HISTORY

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

996 Ref 7.22.4.5 para 1

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

```
998 void                _exit(int);
```

999 to:

```
1000 _Noreturn void    _exit(int);
```

1001 Ref 7.29.1 para 2  
1002 On page 458 line 15801 section <wchar.h>, change:

1003       **mbstate\_t**    An object type other than an array type ...

1004 to:

1005       **mbstate\_t**    A complete object type other than an array type ...

## 1006 **Changes to XSH**

1007 Ref 7.1.4 paras 5, 6  
1008 On page 471 line 16224 section 2.1.1 Use and Implementation of Functions, add two numbered list  
1009 items:

1010       6. Functions shall prevent data races as follows: A function shall not directly or indirectly  
1011       access objects accessible by threads other than the current thread unless the objects are  
1012       accessed directly or indirectly via the function's arguments. A function shall not directly or  
1013       indirectly modify objects accessible by threads other than the current thread unless the  
1014       objects are accessed directly or indirectly via the function's non-const arguments.  
1015       Implementations may share their own internal objects between threads if the objects are not  
1016       visible to applications and are protected against data races.

1017       7. Functions shall perform all operations solely within the current thread if those operations  
1018       have effects that are visible to applications.

1019 Ref K.3.1.1  
1020 On page 473 line 16283 section 2.2.1, add a new subsection:

1021       2.2.1.3 *The `__STDC_WANT_LIB_EXT1__` Feature Test Macro*

1022       A POSIX-conforming [XSI]or XSI-conforming[/XSI] application can define the feature test  
1023       macro `__STDC_WANT_LIB_EXT1__` before inclusion of any header.

1024       When an application includes a header described by POSIX.1-20xx, and when this feature  
1025       test macro is defined to have the value 1, the header may make visible those symbols  
1026       specified for the header in Annex K of the ISO C standard that are not already explicitly  
1027       permitted by POSIX.1-20xx to be made visible in the header. These symbols are listed in  
1028       [xref to 2.2.2].

1029       When an application includes a header described by POSIX.1-20xx, and when this feature  
1030       test macro is either undefined or defined to have the value 0, the header shall not make any  
1031       additional symbols visible that are not already made visible by the feature test macro  
1032       `_POSIX_C_SOURCE` [XSI]or `_XOPEN_SOURCE`[/XSI] as described above, except when  
1033       enabled by another feature test macro.

1034 Ref 7.31.8 para 1  
1035 On page 475 line 16347 section 2.2.2, insert a row in the table:

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

1036 Ref 7.31.15 para 1

1037 On page 476 line 16373 section 2.2.2, insert a row in the table:

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

1038 Ref 7.31.8 para 1

1039 On page 477 line 16410 section 2.2.2, insert a row in the table:

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

1040 Ref 7.31.14 para 1

1041 On page 477 line 16417 section 2.2.2, insert a row in the table:

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

1042 Ref K.3.4 - K.3.9

1043 On page 477 line 16436 section 2.2.2 The Name Space, add:

1044 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1  
1045 (see [xref to 2.2.1]), implementations may add symbols to the headers shown in the  
1046 following table provided the identifiers for those symbols have one of the corresponding  
1047 complete names in the table.

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

1048 When the feature test macro `__STDC_WANT_LIB_EXT1__` is defined with the value 1  
1049 (see [xref to 2.2.1]), if any header in the following table is included, macros with the  
1050 complete names shown may be defined.

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

1051 **Note:** The above two tables only include those symbols from Annex K of the ISO C standard that  
1052 are not already allowed to be visible by entries in earlier tables in this section.

1053 Ref 7.1.3 para 1

1054 On page 478 line 16438 section 2.2.2, change:

1055 With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that  
1056 begin with an `<underscore>` and either an uppercase letter or another `<underscore>` are  
1057 always reserved for any use by the implementation.

1058 to:

1059 With the exception of identifiers beginning with the prefix `_POSIX_` and those identifiers  
1060 which are lexically identical to keywords defined by the ISO C standard (for example  
1061 `_Bool`), all identifiers that begin with an `<underscore>` and either an uppercase letter or  
1062 another `<underscore>` are always reserved for any use by the implementation.

1063 Ref 7.1.3 para 1

1064 On page 478 line 16448 section 2.2.2, change:

1065 that have external linkage are always reserved

1066 to:

1067 that have external linkage and `errno` are always reserved

1068 Ref 7.1.3 para 1

1069 On page 479 line 16453 section 2.2.2, add the following in the appropriate place in the list:

1070	<code>aligned_alloc</code>	<code>c32rtomb</code>
1071	<code>at_quick_exit</code>	<code>call_once</code>
1072	<code>atomic_compare_exchange_strong</code>	<code>cnd_broadcast</code>
1073	<code>atomic_compare_exchange_strong_explicit</code>	<code>cnd_destroy</code>
1074	<code>atomic_compare_exchange_weak</code>	<code>cnd_init</code>
1075	<code>atomic_compare_exchange_weak_explicit</code>	<code>cnd_signal</code>
1076	<code>atomic_exchange</code>	<code>cnd_timedwait</code>
1077	<code>atomic_exchange_explicit</code>	<code>cnd_wait</code>
1078	<code>atomic_fetch_add</code>	<code>kill_dependency</code>
1079	<code>atomic_fetch_add_explicit</code>	<code>mbrtoc16</code>
1080	<code>atomic_fetch_and</code>	<code>mbrtoc32</code>
1081	<code>atomic_fetch_and_explicit</code>	<code>mtx_destroy</code>
1082	<code>atomic_fetch_or</code>	<code>mtx_init</code>
1083	<code>atomic_fetch_or_explicit</code>	<code>mtx_lock</code>
1084	<code>atomic_fetch_sub</code>	<code>mtx_timedlock</code>
1085	<code>atomic_fetch_sub_explicit</code>	<code>mtx_trylock</code>
1086	<code>atomic_fetch_xor</code>	<code>mtx_unlock</code>
1087	<code>atomic_fetch_xor_explicit</code>	<code>quick_exit</code>
1088	<code>atomic_flag_clear</code>	<code>thrd_create</code>
1089	<code>atomic_flag_clear_explicit</code>	<code>thrd_current</code>
1090	<code>atomic_flag_test_and_set</code>	<code>thrd_detach</code>
1091	<code>atomic_flag_test_and_set_explicit</code>	<code>thrd_equal</code>
1092	<code>atomic_init</code>	<code>thrd_exit</code>
1093	<code>atomic_is_lock_free</code>	<code>thrd_join</code>
1094	<code>atomic_load</code>	<code>thrd_sleep</code>

1095	atomic_load_explicit	thrd_yield
1096	atomic_signal_fence	timespec_get
1097	atomic_store	tss_create
1098	atomic_store_explicit	tss_delete
1099	atomic_thread_fence	tss_get
1100	c16rtomb	tss_set

1101 Ref 7.1.2 para 4

1102 On page 480 line 16551 section 2.2.2, change:

1103 Prior to the inclusion of a header, the application shall not define any macros with names  
1104 lexically identical to symbols defined by that header.

1105 to:

1106 Prior to the inclusion of a header, or when any macro defined in the header is expanded, the  
1107 application shall not define any macros with names lexically identical to symbols defined by  
1108 that header.

1109 Ref 7.26.5.1

1110 On page 490 line 16980 section 2.4.2 Realtime Signal Generation and Delivery, change:

1111 The function shall be executed in an environment as if it were the *start\_routine* for a newly  
1112 created thread with thread attributes specified by *sigev\_notify\_attributes*.

1113 to:

1114 The function shall be executed in a newly created thread as if it were the *start\_routine* for a  
1115 call to *pthread\_create()* with the thread attributes specified by *sigev\_notify\_attributes*.

1116 Ref 7.14.1.1 para 5

1117 On page 493 line 17088 section 2.4.3 Signal Actions, change:

1118 with static storage duration

1119 to:

1120 with static or thread storage duration that is not a lock-free atomic object

1121 Ref 7.14.1.1 para 5

1122 On page 493 line 17090 section 2.4.3 Signal Actions, after applying bug 711 change:

1123 other than one of the functions and macros listed in the following table

1124 to:

1125 other than one of the functions and macros specified below as being async-signal-safe

1126 Ref 7.14.1.1 para 5

1127 On page 494 line 17133 section 2.4.3 Signal Actions, add *quick\_exit()* to the table of async-signal-  
1128 safe functions.

1129 Ref 7.14.1.1 para 5  
1130 On page 494 line 17147 section 2.4.3 Signal Actions, change:

1131 Any function or function-like macro not in the above table may be unsafe with respect to  
1132 signals.

1133 to:

1134 In addition, the functions in `<stdatomic.h>` other than `atomic_init()` shall be async-signal-  
1135 safe when the atomic arguments are lock-free, and the `atomic_is_lock_free()` function shall  
1136 be async-signal-safe when called with an atomic argument.

1137 All other functions (including generic functions) and function-like macros may be unsafe  
1138 with respect to signals.

1139 Ref 7.21.2 para 7,8  
1140 On page 496 line 17228 section 2.5 Standard I/O Streams, add a new paragraph:

1141 Each stream shall have an associated lock that is used to prevent data races when multiple  
1142 threads of execution access a stream, and to restrict the interleaving of stream operations  
1143 performed by multiple threads. Only one thread can hold this lock at a time. The lock shall  
1144 be reentrant: a single thread can hold the lock multiple times at a given time. All functions  
1145 that read, write, position, or query the position of a stream, [CX]except those with names  
1146 ending `_unlocked`[/CX], shall lock the stream [CX] as if by a call to `flockfile()`[/CX] before  
1147 accessing it and release the lock [CX] as if by a call to `funlockfile()`[/CX] when the access is  
1148 complete.

1149 Ref (none)  
1150 On page 498 line 17312 section 2.5.2 Stream Orientation and Encoding Rules, change:

1151 For conformance to the ISO/IEC 9899: 1999 standard, the definition of a stream includes an  
1152 “orientation”.

1153 to:

1154 The definition of a stream includes an “orientation”.

1155 Ref 7.26.5.8  
1156 On page 508 line 17720 section 2.8.4 Process Scheduling, change:

1157 When a running thread issues the `sched_yield()` function

1158 to:

1159 When a running thread issues the `sched_yield()` or `thrd_yield()` function

1160 Ref 7.17.2.2 para 3, 7.22.2.2 para 3  
1161 On page 513 line 17907,17916 section 2.9.1 Thread-Safety, add `atomic_init()` and `srand()` to the list  
1162 of functions that need not be thread-safe.

1163 Ref 7.12.8.3, 7.22.4.8  
1164 On page 513 line 17907-17927 section 2.9.1 Thread-Safety, delete the following from the list of



1165 functions that need not be thread-safe:

1166 *lgamma()*, *lgammaf()*, *lgammal()*, *system()*

1167 [Note to reviewers: deletion of \*mblen\(\)\*, \*mbtowl\(\)\*, and \*wctomb\(\)\* from this list is the subject of](#)  
1168 [Mantis bug 708.](#)

1169 Ref 7.28.1 para 1

1170 On page 513 line 17928 section 2.9.1 Thread-Safety, change:

1171 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument.  
1172 The *mbrlen()*, *mbrtowc()*, *mbsnrtowcs()*, *mbsrtowcs()*, *wctomb()*, *wcsnrtombs()*, and  
1173 *wcsrtombs()* functions need not be thread-safe if passed a NULL *ps* argument.

1174 to:

1175 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a null pointer  
1176 argument. The *c16rtomb()*, *c32rtomb()*, *mbrlen()*, *mbrtoc16()*, *mbrtoc32()*, *mbrtowc()*,  
1177 *mbsnrtowcs()*, *mbsrtowcs()*, *wctomb()*, *wcsnrtombs()*, and *wcsrtombs()* functions need not  
1178 be thread-safe if passed a null *ps* argument. The *lgamma()*, *lgammaf()*, and *lgammal()*  
1179 functions shall be thread-safe [XSI]except that they need not avoid data races when storing a  
1180 value in the *siggam* variable[/XSI].

1181 Ref 7.1.4 para 5

1182 On page 513 line 17934 section 2.9.1 Thread-Safety, change:

1183 Implementations shall provide internal synchronization as necessary in order to satisfy this  
1184 requirement.

1185 to:

1186 Some functions that are not required to be thread-safe are nevertheless required to avoid data  
1187 races with either all or some other functions, as specified on their individual reference pages.

1188 Implementations shall provide internal synchronization as necessary in order to satisfy  
1189 thread-safety requirements.

1190 Ref 7.26.5

1191 On page 513 line 17944 section 2.9.2 Thread IDs, change:

1192 The lifetime of a thread ID ends after the thread terminates if it was created with the  
1193 *detachstate* attribute set to `PTHREAD_CREATE_DETACHED` or if *pthread\_detach()* or  
1194 *pthread\_join()* has been called for that thread.

1195 to:

1196 The lifetime of a thread ID ends after the thread terminates if it was created using  
1197 *pthread\_create()* with the *detachstate* attribute set to `PTHREAD_CREATE_DETACHED` or  
1198 if *pthread\_detach()*, *pthread\_join()*, *thrd\_detach()* or *thrd\_join()* has been called for that  
1199 thread.

1200 Ref 7.26.5

1201 On page 514 line 17950 section 2.9.2 Thread IDs, change:

1202 If a thread is detached, its thread ID is invalid for use as an argument in a call to  
1203 *pthread\_detach()* or *pthread\_join()*.

1204 to:

1205 If a thread is detached, its thread ID is invalid for use as an argument in a call to  
1206 *pthread\_detach()*, *pthread\_join()*, *thrd\_detach()* or *thrd\_join()*.

1207 Ref 7.26.4

1208 On page 514 line 17956 section 2.9.3 Thread Mutexes, change:

1209 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

1210 to:

1211 A thread shall become the owner of a mutex, *m*, of type **pthread\_mutex\_t** when one of the  
1212 following occurs:

1213 Ref 7.26.3, 7.26.4

1214 On page 514 line 17972 section 2.9.3 Thread Mutexes, add two new paragraphs and lists:

1215 A thread shall become the owner of a mutex, *m*, of type **mtx\_t** when one of the following  
1216 occurs:

- 1217 • It calls *mtx\_lock()* with *m* as the *mtx* argument and the call returns *thrd\_success*.
- 1218 • It calls *mtx\_trylock()* with *m* as the *mtx* argument and the call returns  
1219 *thrd\_success*.
- 1220 • It calls *mtx\_timedlock()* with *m* as the *mtx* argument and the call returns  
1221 *thrd\_success*.
- 1222 • It calls *cnd\_wait()* with *m* as the *mtx* argument and the call returns *thrd\_success*.
- 1223 • It calls *cnd\_timedwait()* with *m* as the *mtx* argument and the call returns  
1224 *thrd\_success* or *thrd\_timedout*.

1225 The thread shall remain the owner of *m* until one of the following occurs:

- 1226 • It executes *mtx\_unlock()* with *m* as the *mtx* argument.
- 1227 • It blocks in a call to *cnd\_wait()* with *m* as the *mtx* argument.
- 1228 • It blocks in a call to *cnd\_timedwait()* with *m* as the *mtx* argument.

1229 Ref 7.26.4

1230 On page 514 line 17980 section 2.9.3 Thread Mutexes, change:

1231 Robust mutexes provide a means to enable the implementation to notify other threads in the  
1232 event of a process terminating while one of its threads holds a mutex lock.

1233 to:

1234 Robust mutexes provide a means to enable the implementation to notify other threads in the  
1235 event of a process terminating while one of its threads holds a lock on a mutex of type  
1236 **pthread\_mutex\_t**.

1237 Ref 7.26.5  
1238 On page 517 line 18085 section 2.9.5 Thread Cancellation, change:

1239           The thread cancellation mechanism allows a thread to terminate the execution of any other  
1240           thread in the process in a controlled manner.

1241 to:

1242           The thread cancellation mechanism allows a thread to terminate the execution of any thread  
1243           in the process, except for threads created using *thrd\_create()*, in a controlled manner.

1244 Ref 7.26.3, 7.26.5.6  
1245 On page 518 line 18119-18137 section 2.9.5.2 Cancellation Points, add the following to the list of  
1246 functions that are required to be cancellation points:

1247           *cnd\_timedwait()*, *cnd\_wait()*, *thrd\_join()*, *thrd\_sleep()*

1248 Ref 7.26.5  
1249 On page 520 line 18225 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1250           Each thread maintains a list of cancellation cleanup handlers.

1251 to:

1252           Each thread that was not created using *thrd\_create()* maintains a list of cancellation cleanup  
1253           handlers.

1254 Ref 7.26.6.1  
1255 On page 521 line 18240 section 2.9.5.3 Thread Cancellation Cleanup Handlers, change:

1256           as described for *pthread\_key\_create()*

1257 to:

1258           as described for *pthread\_key\_create()* and *tss\_create()*

1259 Ref 7.26  
1260 On page 523 line 18337 section 2.9.9 Synchronization Object Copies and Alternative Mappings,  
1261 add a new sentence:

1262           For ISO C functions declared in **<threads.h>**, the above requirements shall apply as if  
1263           condition variables of type **cnd\_t** and mutexes of type **mtx\_t** have a process-shared attribute  
1264           that is set to **PTHREAD\_PROCESS\_PRIVATE**.

1265 Ref 7.26.3  
1266 On page 547 line 19279 section 2.12.1 Defined Types, change:

1267           **pthread\_cond\_t**

1268 to

1269           **pthread\_cond\_t, cnd\_t**

1270   Ref 7.26.6, 7.26.4

1271   On page 547 line 19281 section 2.12.1 Defined Types, change:

1272           **pthread\_key\_t**

1273           **pthread\_mutex\_t**

1274   to

1275           **pthread\_key\_t, tss\_t**

1276           **pthread\_mutex\_t, mtx\_t**

1277   Ref 7.26.2.1

1278   On page 547 line 19284 section 2.12.1 Defined Types, change:

1279           **pthread\_once\_t**

1280   to

1281           **pthread\_once\_t, once\_flag**

1282   Ref 7.26.5

1283   On page 547 line 19287 section 2.12.1 Defined Types, change:

1284           **pthread\_t**

1285   to

1286           **pthread\_t, thrd\_t**

1287   Ref 7.3.9.3

1288   On page 552 line 19370 insert a new CMPLX() section:

1289   **NAME**

1290       CMPLX — make a complex value

1291   **SYNOPSIS**

1292       #include <complex.h>

1293       double complex       CMPLX(double x, double y);

1294       float complex        CMPLXF(float x, float y);

1295       long double complex CMPLXL(long double x, long double y);

1296   **DESCRIPTION**

1297       [**CX**] The functionality described on this reference page is aligned with the ISO C standard.  
1298       Any conflict between the requirements described here and the ISO C standard is  
1299       unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/**CX**]

1300       The CMPLX macros shall expand to an expression of the specified complex type, with the  
1301       real part having the (converted) value of *x* and the imaginary part having the (converted)  
1302       value of *y*. The resulting expression shall be suitable for use as an initializer for an object  
1303       with static or thread storage duration, provided both arguments are likewise suitable.

1304 **RETURN VALUE**

1305 The CMPLX macros return the complex value  $x + iy$  (where  $i$  is the imaginary unit).

1306 These macros shall behave as if the implementation supported imaginary types and the  
1307 definitions were:

```
1308 #define CMPLX(x, y) ((double complex)((double)(x) + \  
1309 _Imaginary_I * (double)(y)))  
1310 #define CMPLXF(x, y) ((float complex)((float)(x) + \  
1311 _Imaginary_I * (float)(y)))  
1312 #define CMPLXL(x, y) ((long double complex)((long double)(x) + \  
1313 _Imaginary_I * (long double)(y)))
```

1314 **ERRORS**

1315 No errors are defined.

1316 **EXAMPLES**

1317 None.

1318 **APPLICATION USAGE**

1319 None.

1320 **RATIONALE**

1321 None.

1322 **FUTURE DIRECTIONS**

1323 None.

1324 **SEE ALSO**

1325 XBD <complex.h>

1326 **CHANGE HISTORY**

1327 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1328 Ref 7.22.4.5 para 1

1329 On page 553 line 19384 section `_Exit()`, change:

```
1330 void _Exit(int status);
```

```
1331 #include <unistd.h>
```

```
1332 void _exit(int status);
```

1333 to:

```
1334 _Noreturn void _Exit(int status);
```

```
1335 #include <unistd.h>
```

```
1336 _Noreturn void _exit(int status);
```

1337 Ref 7.22.4.5 para 2

1338 On page 553 line 19396 section `_Exit()`, change:

1339 shall not call functions registered with `atexit()` nor any registered signal handlers

1340 to:

1341 shall not call functions registered with `atexit()` nor `at_quick_exit()`, nor any registered signal  
1342 handlers

1343 Ref (none)

1344 On page 557 line 19562 section `_Exit()`, change:

1345 The ISO/IEC 9899: 1999 standard adds the `_Exit()` function

1346 to:

1347 The ISO/IEC 9899: 1999 standard added the `_Exit()` function

1348 Ref 7.22.4.3, 7.22.4.7

1349 On page 557 line 19568 section `_Exit()`, add `at_quick_exit` and `quick_exit` to the SEE ALSO section.

1350 Ref 7.22.4.1 para 1

1351 On page 565 line 19761 section `abort()`, change:

1352 `void abort(void);`

1353 to:

1354 `_Noreturn void abort(void);`

1355 Ref (none)

1356 On page 565 line 19785 section `abort()`, change:

1357 The ISO/IEC 9899: 1999 standard requires the `abort()` function to be async-signal-safe.

1358 to:

1359 The ISO/IEC 9899: 1999 standard required (and the current standard still requires) the  
1360 `abort()` function to be async-signal-safe.

1361 Ref 7.22.3.1

1362 On page 597 line 20771 insert the following new `aligned_alloc()` section:

1363 **NAME**

1364 `aligned_alloc` — allocate memory with a specified alignment

1365 **SYNOPSIS**

1366 `#include <stdlib.h>`

1367 `void *aligned_alloc(size_t alignment, size_t size);`

1368 **DESCRIPTION**

1369 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1370 Any conflict between the requirements described here and the ISO C standard is  
1371 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard. [/CX]

1372 The *aligned\_alloc()* function shall allocate unused space for an object whose alignment is  
1373 specified by *alignment*, whose size in bytes is specified by *size* and whose value is  
1374 indeterminate.

1375 The order and contiguity of storage allocated by successive calls to *aligned\_alloc()* is  
1376 unspecified. Each such allocation shall yield a pointer to an object disjoint from any other  
1377 object. The pointer returned shall point to the start (lowest byte address) of the allocated  
1378 space. If the value of *alignment* is not a valid alignment supported by the implementation, a  
1379 null pointer shall be returned. If the space cannot be allocated, a null pointer shall be  
1380 returned. If the size of the space requested is 0, the behavior is implementation-defined:  
1381 either a null pointer shall be returned to indicate an error, or the behavior shall be as if the  
1382 size were some non-zero value, except that the behavior is undefined if the returned pointer  
1383 is used to access an object.

1384 For purposes of determining the existence of a data race, *aligned\_alloc()* shall behave as  
1385 though it accessed only memory locations accessible through its arguments and not other  
1386 static duration storage. The function may, however, visibly modify the storage that it  
1387 allocates. Calls to *aligned\_alloc()*, *calloc()*, *free()*, *malloc()*,  
1388 [\[ADV\]posix\\_memalign\(\)](#), [\[/ADV\] \[CX\]reallocarray\(\)](#), [\[/CX\]](#) and *realloc()* that allocate or  
1389 deallocate a particular region of memory shall occur in a single total order (see [\[xref to XBD](#)  
1390 [4.12.1\]](#)), and each such deallocation call shall synchronize with the next allocation (if any)  
1391 in this order.

## 1392 RETURN VALUE

1393 Upon successful completion ~~with size not equal to 0~~, *aligned\_alloc()* shall return a pointer to  
1394 the allocated space; ~~if size is 0, either:~~

- 1395 • ~~A null pointer shall be returned [CX] and errno may be set to an implementation-~~  
1396 ~~defined value, [CX] or~~

1397 ~~A pointer to the allocated space shall be returned. T~~he application shall ensure that the  
1398 pointer is not used to access an object.

1399 Otherwise, it shall return a null pointer [CX] and set *errno* to indicate the error [CX].

## 1400 ERRORS

1401 The *aligned\_alloc()* function shall fail if:

1402 [CX][EINVAL] The value of *alignment* is not a valid alignment supported by the  
1403 implementation.

1404 [ENOMEM] Insufficient storage space is available. [/CX]

1405 The *aligned\_alloc()* function may fail if:

1406 [CX][EINVAL] *size* is 0 and the implementation does not support 0 sized allocations. [  
1407 CX]

1408 **EXAMPLES**

1409 None.

1410 **APPLICATION USAGE**

1411 None.

1412 **RATIONALE**

1413 ~~None.~~ See the [RATIONALE](#) for [\[xref to malloc\(\)\]](#).

1414 **FUTURE DIRECTIONS**

1415 None.

1416 **SEE ALSO**

1417 *calloc*, *free*, *getrlimit*, *malloc*, *posix\_memalign*, *realloc*

1418 XBD <stdlib.h>

1419 **CHANGE HISTORY**

1420 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1421 Ref 7.27.3, 7.1.4 para 5

1422 On page 600 line 20911 section *asctime()*, change:

1423 [CX]The *asctime()* function need not be thread-safe.[/CX]

1424 to:

1425 The *asctime()* function need not be thread-safe; however, *asctime()* shall avoid data races  
1426 with all functions other than itself, *ctime()*, *gmtime()* and *localtime()*.

1427 Ref 7.22.4.3

1428 On page 618 line 21380 insert the following new *at\_quick\_exit()* section:

1429 **NAME**

1430 *at\_quick\_exit* — register a function to be called from *quick\_exit()*

1431 **SYNOPSIS**

1432 `#include <stdlib.h>`

1433 `int at_quick_exit(void (*func)(void));`

1434 **DESCRIPTION**

1435 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1436 Any conflict between the requirements described here and the ISO C standard is  
1437 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1438 The *at\_quick\_exit()* function shall register the function pointed to by *func*, to be called  
1439 without arguments should *quick\_exit()* be called. It is unspecified whether a call to the  
1440 *at\_quick\_exit()* function that does not happen before the *quick\_exit()* function is called will  
1441 succeed.

1442 At least 32 functions can be registered with *at\_quick\_exit()*.



1443 [CX]After a successful call to any of the *exec* functions, any functions previously registered  
1444 by *at\_quick\_exit()* shall no longer be registered.[/CX]

1445 **RETURN VALUE**

1446 Upon successful completion, *at\_quick\_exit()* shall return 0; otherwise, it shall return a non-  
1447 zero value.

1448 **ERRORS**

1449 No errors are defined.

1450 **EXAMPLES**

1451 None.

1452 **APPLICATION USAGE**

1453 The *at\_quick\_exit()* function registrations are distinct from the *atexit()* registrations, so  
1454 applications might need to call both registration functions with the same argument.

1455 The functions registered by a call to *at\_quick\_exit()* must return to ensure that all registered  
1456 functions are called.

1457 The application should call *sysconf()* to obtain the value of {ATEXIT\_MAX}, the number of  
1458 functions that can be registered. There is no way for an application to tell how many  
1459 functions have already been registered with *at\_quick\_exit()*.

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

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

1466 **RATIONALE**

1467 None.

1468 **FUTURE DIRECTIONS**

1469 None.

1470 **SEE ALSO**

1471 *atexit*, *exec*, *exit*, *quick\_exit*, *sysconf*

1472 XBD <stdlib.h>

1473 **CHANGE HISTORY**

1474 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1475 Ref 7.22.4.3

1476 On page 618 line 21381 section *atexit()*, change:

1477 *atexit* — register a function to run at process termination

1478 to:

1479 `atexit` — register a function to be called from `exit()` or after return from `main()`

1480 Ref 7.22.4.2 para 2, 7.22.4.3

1481 On page 618 line 21389 section `atexit()`, change:

1482 The `atexit()` function shall register the function pointed to by `func`, to be called without  
1483 arguments at normal program termination. At normal program termination, all functions  
1484 registered by the `atexit()` function shall be called, in the reverse order of their registration,  
1485 except that a function is called after any previously registered functions that had already  
1486 been called at the time it was registered. Normal termination occurs either by a call to `exit()`  
1487 or a return from `main()`.

1488 to:

1489 The `atexit()` function shall register the function pointed to by `func`, to be called without  
1490 arguments from `exit()`, or after return from the initial call to `main()`, or on the last thread  
1491 termination. If the `exit()` function is called, it is unspecified whether a call to the `atexit()`  
1492 function that does not happen before `exit()` is called will succeed.

1493 [Note to reviewers: the part about all registered functions being called in reverse order is duplicated](#)  
1494 [on the `exit\(\)` page and is not needed here.](#)

1495 Ref 7.22.4.2 para 2

1496 On page 618 line 21405 section `atexit()`, insert a new first APPLICATION USAGE paragraph:

1497 The `atexit()` function registrations are distinct from the `at_quick_exit()` registrations, so  
1498 applications might need to call both registration functions with the same argument.

1499 Ref 7.22.4.3

1500 On page 618 line 21410 section `atexit()`, change:

1501 Since the behavior is undefined if the `exit()` function is called more than once, portable  
1502 applications calling `atexit()` must ensure that the `exit()` function is not called at normal  
1503 process termination when all functions registered by the `atexit()` function are called.

1504 All functions registered by the `atexit()` function are called at normal process termination,  
1505 which occurs by a call to the `exit()` function or a return from `main()` or on the last thread  
1506 termination, when the behavior is as if the implementation called `exit()` with a zero argument  
1507 at thread termination time.

1508 If, at normal process termination, a function registered by the `atexit()` function is called and a  
1509 portable application needs to stop further `exit()` processing, it must call the `_exit()` function  
1510 or the `_Exit()` function or one of the functions which cause abnormal process termination.

1511 to:

1512 Since the behavior is undefined if the `exit()` function is called more than once, portable  
1513 applications calling `atexit()` must ensure that the `exit()` function is not called when the  
1514 functions registered by the `atexit()` function are called.

1515 If a function registered by the `atexit()` function is called and a portable application needs to  
1516 stop further `exit()` processing, it must call the `_exit()` function or the `_Exit()` function or one  
1517 of the functions which cause abnormal process termination.

1518 Ref 7.22.4.3

1519 On page 619 line 21425 section `atexit()`, add `at_quick_exit` to the SEE ALSO section.

1520 Ref 7.16

1521 On page 624 line 21548 insert the following new `atomic_*` sections:

## 1522 NAME

1523 `atomic_compare_exchange_strong`, `atomic_compare_exchange_strong_explicit`,  
1524 `atomic_compare_exchange_weak`, `atomic_compare_exchange_weak_explicit` — atomically  
1525 compare and exchange the values of two objects

## 1526 SYNOPSIS

```
1527 #include <stdatomic.h>
1528 _Bool atomic_compare_exchange_strong(volatile A *object,
1529     C *expected, C desired);
1530 _Bool atomic_compare_exchange_strong_explicit(volatile A *object,
1531     C *expected, C desired, memory_order success,
1532     memory_order failure);
1533 _Bool atomic_compare_exchange_weak(volatile A *object,
1534     C *expected, C desired);
1535 _Bool atomic_compare_exchange_weak_explicit(volatile A *object,
1536     C *expected, C desired, memory_order success,
1537     memory_order failure);
```

## 1538 DESCRIPTION

1539 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1540 Any conflict between the requirements described here and the ISO C standard is  
1541 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1542 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1543 `<stdatomic.h>` header nor support these generic functions.

1544 The `atomic_compare_exchange_strong_explicit()` generic function shall atomically compare  
1545 the contents of the memory pointed to by `object` for equality with that pointed to by  
1546 `expected`, and if true, shall replace the contents of the memory pointed to by `object`  
1547 with `desired`, and if false, shall update the contents of the memory pointed to by `expected`  
1548 with that pointed to by `object`. This operation shall be an atomic read-modify-write operation  
1549 (see [xref to XBD 4.12.1]). If the comparison is true, memory shall be affected according to  
1550 the value of `success`, and if the comparison is false, memory shall be affected according to  
1551 the value of `failure`. The application shall ensure that `failure` is not  
1552 `memory_order_release` nor `memory_order_acq_rel`, and shall ensure that `failure` is  
1553 no stronger than `success`.

1554 The `atomic_compare_exchange_strong()` generic function shall be equivalent to  
1555 `atomic_compare_exchange_strong_explicit()` called with `success` and `failure` both set to  
1556 `memory_order_seq_cst`.

1557 The `atomic_compare_exchange_weak_explicit()` generic function shall be equivalent to  
1558 `atomic_compare_exchange_strong_explicit()`, except that the compare-and-exchange

1559 operation may fail spuriously. That is, even when the contents of memory referred to by  
1560 *expected* and *object* are equal, it may return zero and store back to *expected* the same  
1561 memory contents that were originally there.

1562 The `atomic_compare_exchange_weak()` generic function shall be equivalent to  
1563 `atomic_compare_exchange_weak_explicit()` called with *success* and *failure* both set to  
1564 `memory_order_seq_cst`.

#### 1565 **RETURN VALUE**

1566 These generic functions shall return the result of the comparison.

#### 1567 **ERRORS**

1568 No errors are defined.

#### 1569 **EXAMPLES**

1570 None.

#### 1571 **APPLICATION USAGE**

1572 A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will  
1573 be in a loop. For example:

```
1574 exp = atomic_load(&cur);  
1575 do {  
1576     des = function(exp);  
1577 } while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

1578 When a compare-and-exchange is in a loop, the weak version will yield better performance  
1579 on some platforms. When a weak compare-and-exchange would require a loop and a strong  
1580 one would not, the strong one is preferable.

#### 1581 **RATIONALE**

1582 None.

#### 1583 **FUTURE DIRECTIONS**

1584 None.

#### 1585 **SEE ALSO**

1586 XBD Section 4.12.1, `<stdatomic.h>`

#### 1587 **CHANGE HISTORY**

1588 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

#### 1589 **NAME**

1590 `atomic_exchange`, `atomic_exchange_explicit` — atomically exchange the value of an object

#### 1591 **SYNOPSIS**

```
1592 #include <stdatomic.h>  
1593 C atomic_exchange(volatile A *object, C desired);  
1594 C atomic_exchange_explicit(volatile A *object,  
1595     C desired, memory_order order);
```

#### 1596 **DESCRIPTION**

1597 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1598 Any conflict between the requirements described here and the ISO C standard is  
1599 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1600 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1601 `<stdatomic.h>` header nor support these generic functions.

1602 The `atomic_exchange_explicit()` generic function shall atomically replace the value pointed  
1603 to by *object* with *desired*. This operation shall be an atomic read-modify-write operation (see  
1604 [xref to XBD 4.12.1]). Memory shall be affected according to the value of *order*.

1605 The `atomic_exchange()` generic function shall be equivalent to `atomic_exchange_explicit()`  
1606 called with *order* set to `memory_order_seq_cst`.

#### 1607 RETURN VALUE

1608 These generic functions shall return the value pointed to by *object* immediately before the  
1609 effects.

#### 1610 ERRORS

1611 No errors are defined.

#### 1612 EXAMPLES

1613 None.

#### 1614 APPLICATION USAGE

1615 None.

#### 1616 RATIONALE

1617 None.

#### 1618 FUTURE DIRECTIONS

1619 None.

#### 1620 SEE ALSO

1621 XBD Section 4.12.1, `<stdatomic.h>`

#### 1622 CHANGE HISTORY

1623 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

#### 1624 NAME

1625 `atomic_fetch_add`, `atomic_fetch_add_explicit`, `atomic_fetch_and`,  
1626 `atomic_fetch_and_explicit`, `atomic_fetch_or`, `atomic_fetch_or_explicit`, `atomic_fetch_sub`,  
1627 `atomic_fetch_sub_explicit`, `atomic_fetch_xor`, `atomic_fetch_xor_explicit` — atomically  
1628 replace the value of an object with the result of a computation

#### 1629 SYNOPSIS

```
1630 #include <stdatomic.h>
1631 C atomic_fetch_add(volatile A *object, M operand);
1632 C atomic_fetch_add_explicit(volatile A *object, M operand,
1633 memory_order order);
1634 C atomic_fetch_and(volatile A *object, M operand);
1635 C atomic_fetch_and_explicit(volatile A *object, M operand,
```

```

1636         memory_order order);
1637 C   atomic_fetch_or(volatile A *object, M operand);
1638 C   atomic_fetch_or_explicit(volatile A *object, M operand,
1639         memory_order order);
1640 C   atomic_fetch_sub(volatile A *object, M operand);
1641 C   atomic_fetch_sub_explicit(volatile A *object, M operand,
1642         memory_order order);
1643 C   atomic_fetch_xor(volatile A *object, M operand);
1644 C   atomic_fetch_xor_explicit(volatile A *object, M operand,
1645         memory_order order);

```

1646 **DESCRIPTION**

1647 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
 1648 Any conflict between the requirements described here and the ISO C standard is  
 1649 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1650 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
 1651 `<stdatomic.h>` header nor support these generic functions.

1652 The `atomic_fetch_add_explicit()` generic function shall atomically replace the value pointed  
 1653 to by *object* with the result of adding *operand* to this value. This operation shall be an  
 1654 atomic read-modify-write operation (see [xref to XBD 4.12.1]). Memory shall be affected  
 1655 according to the value of *order*.

1656 The `atomic_fetch_add()` generic function shall be equivalent to `atomic_fetch_add_explicit()`  
 1657 called with *order* set to `memory_order_seq_cst`.

1658 The other `atomic_fetch_*`() generic functions shall be equivalent to  
 1659 `atomic_fetch_add_explicit()` if their name ends with *explicit*, or to `atomic_fetch_add()` if it  
 1660 does not, respectively, except that they perform the computation indicated in their name,  
 1661 instead of addition:

```

1662 sub   subtraction
1663 or    bitwise inclusive OR
1664 xor   bitwise exclusive OR
1665 and   bitwise AND

```

1666 For addition and subtraction, the application shall ensure that **A** is an atomic integer type or  
 1667 an atomic pointer type and is not **atomic\_bool**. For the other operations, the application  
 1668 shall ensure that **A** is an atomic integer type and is not **atomic\_bool**.

1669 For signed integer types, the computation shall silently wrap around on overflow; there are  
 1670 no undefined results. For pointer types, the result can be an undefined address, but the  
 1671 computations otherwise have no undefined behavior.

1672 **RETURN VALUE**

1673 These generic functions shall return the value pointed to by *object* immediately before the  
 1674 effects.

1675 **ERRORS**

1676 No errors are defined.

1677 **EXAMPLES**

1678 None.

1679 **APPLICATION USAGE**

1680 The operation of these generic functions is nearly equivalent to the operation of the  
1681 corresponding compound assignment operators +=, -=, etc. The only differences are that the  
1682 compound assignment operators are not guaranteed to operate atomically, and the value  
1683 yielded by a compound assignment operator is the updated value of the object, whereas the  
1684 value returned by these generic functions is the previous value of the atomic object.

1685 **RATIONALE**

1686 None.

1687 **FUTURE DIRECTIONS**

1688 None.

1689 **SEE ALSO**

1690 XBD Section 4.12.1, <stdatomic.h>

1691 **CHANGE HISTORY**

1692 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1693 **NAME**

1694 `atomic_flag_clear`, `atomic_flag_clear_explicit` — clear an atomic flag

1695 **SYNOPSIS**

```
1696 #include <stdatomic.h>  
1697 void atomic_flag_clear(volatile atomic_flag *object);  
1698 void atomic_flag_clear_explicit(  
1699     volatile atomic_flag *object, memory_order order);
```

1700 **DESCRIPTION**

1701 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1702 Any conflict between the requirements described here and the ISO C standard is  
1703 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[CX]

1704 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1705 `<stdatomic.h>` header nor support these functions.

1706 The `atomic_flag_clear_explicit()` function shall atomically place the atomic flag pointed to  
1707 by `object` into the clear state. Memory shall be affected according to the value of `order`,  
1708 which the application shall ensure is not `memory_order_acquire` nor  
1709 `memory_order_acq_rel`.

1710 The `atomic_flag_clear()` function shall be equivalent to `atomic_flag_clear_explicit()` called  
1711 with `order` set to `memory_order_seq_cst`.

1712 **RETURN VALUE**

1713 These functions shall not return a value.

1714 **ERRORS**

1715 No errors are defined.

1716 **EXAMPLES**

1717       None.

1718 **APPLICATION USAGE**

1719       None.

1720 **RATIONALE**

1721       None.

1722 **FUTURE DIRECTIONS**

1723       None.

1724 **SEE ALSO**

1725       XBD Section 4.12.1, <**stdatomic.h**>

1726 **CHANGE HISTORY**

1727       First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1728 **NAME**

1729       atomic\_flag\_test\_and\_set, atomic\_flag\_test\_and\_set\_explicit — test and set an atomic flag

1730 **SYNOPSIS**

```
1731       #include <stdatomic.h>
1732       _Bool atomic_flag_test_and_set(volatile atomic_flag *object);
1733       _Bool atomic_flag_test_and_set_explicit(
1734           volatile atomic_flag *object, memory_order order);
```

1735 **DESCRIPTION**

1736       [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1737       Any conflict between the requirements described here and the ISO C standard is  
1738       unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1739       Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1740       <**stdatomic.h**> header nor support these functions.

1741       The `atomic_flag_test_and_set_explicit()` function shall atomically place the atomic flag  
1742       pointed to by `object` into the set state and return the value corresponding to the immediately  
1743       preceding state. This operation shall be an atomic read-modify-write operation (see [xref to  
1744       XBD 4.12.1]). Memory shall be affected according to the value of `order`.

1745       The `atomic_flag_test_and_set()` function shall be equivalent to  
1746       `atomic_flag_test_and_set_explicit()` called with `order` set to `memory_order_seq_cst`.

1747 **RETURN VALUE**

1748       These functions shall return the value that corresponds to the state of the atomic flag  
1749       immediately before the effects. The return value true shall correspond to the set state and the  
1750       return value false shall correspond to the clear state.

1751 **ERRORS**

1752       No errors are defined.

1753 **EXAMPLES**



1754 None.

1755 **APPLICATION USAGE**

1756 None.

1757 **RATIONALE**

1758 None.

1759 **FUTURE DIRECTIONS**

1760 None.

1761 **SEE ALSO**

1762 XBD Section 4.12.1, <**stdatomic.h**>

1763 **CHANGE HISTORY**

1764 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1765 **NAME**

1766 `atomic_init` — initialize an atomic object

1767 **SYNOPSIS**

```
1768 #include <stdatomic.h>
1769 void atomic_init(volatile A *obj, C value);
```

1770 **DESCRIPTION**

1771 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1772 Any conflict between the requirements described here and the ISO C standard is  
1773 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1774 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1775 `<stdatomic.h>` header nor support this generic function.

1776 The `atomic_init()` generic function shall initialize the atomic object pointed to by `obj` to the  
1777 value `value`, while also initializing any additional state that the implementation might need  
1778 to carry for the atomic object.

1779 Although this function initializes an atomic object, it does not avoid data races; concurrent  
1780 access to the variable being initialized, even via an atomic operation, constitutes a data race.

1781 **RETURN VALUE**

1782 The `atomic_init()` generic function shall not return a value.

1783 **ERRORS**

1784 No errors are defined.

1785 **EXAMPLES**

```
1786 atomic_int guide;
1787 atomic_init(&guide, 42);
```

1788 **APPLICATION USAGE**

1789 None.

1790 **RATIONALE**

1791 None.

1792 **FUTURE DIRECTIONS**

1793 None.

1794 **SEE ALSO**

1795 XBD <stdatomic.h>

1796 **CHANGE HISTORY**

1797 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1798 **NAME**

1799 `atomic_is_lock_free` — indicate whether or not atomic operations are lock-free

1800 **SYNOPSIS**

```
1801 #include <stdatomic.h>
1802 _Bool atomic_is_lock_free(const volatile A *obj);
```

1803 **DESCRIPTION**

1804 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1805 Any conflict between the requirements described here and the ISO C standard is  
1806 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1807 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1808 `<stdatomic.h>` header nor support this generic function.

1809 The `atomic_is_lock_free()` generic function shall indicate whether or not atomic operations  
1810 on objects of the type pointed to by `obj` are lock-free; `obj` can be a null pointer.

1811 **RETURN VALUE**

1812 The `atomic_is_lock_free()` generic function shall return a non-zero value if and only if  
1813 atomic operations on objects of the type pointed to by `obj` are lock-free. During the lifetime  
1814 of the calling process, the result of the lock-free query shall be consistent for all pointers of  
1815 the same type.

1816 **ERRORS**

1817 No errors are defined.

1818 **EXAMPLES**

1819 None.

1820 **APPLICATION USAGE**

1821 None.

1822 **RATIONALE**

1823 Operations that are lock-free should also be address-free. That is, atomic operations on the  
1824 same memory location via two different addresses will communicate atomically. The  
1825 implementation should not depend on any per-process state. This restriction enables  
1826 communication via memory mapped into a process more than once and memory shared  
1827 between two processes.

1828 **FUTURE DIRECTIONS**

1829       None.

1830 **SEE ALSO**

1831       XBD <stdatomic.h>

1832 **CHANGE HISTORY**

1833       First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1834 **NAME**

1835       atomic\_load, atomic\_load\_explicit — atomically obtain the value of an object

1836 **SYNOPSIS**

```
1837       #include <stdatomic.h>
1838       C atomic_load(const volatile A *object);
1839       C atomic_load_explicit(const volatile A *object,
1840                               memory_order order);
```

1841 **DESCRIPTION**

1842       [**CX**] The functionality described on this reference page is aligned with the ISO C standard.  
1843       Any conflict between the requirements described here and the ISO C standard is  
1844       unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/**CX**]

1845       Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1846       <**stdatomic.h**> header nor support these generic functions.

1847       The `atomic_load_explicit()` generic function shall atomically obtain the value pointed to by  
1848       *object*. Memory shall be affected according to the value of *order*, which the application shall  
1849       ensure is not `memory_order_release` nor `memory_order_acq_rel`.

1850       The `atomic_load()` generic function shall be equivalent to `atomic_load_explicit()` called with  
1851       *order* set to `memory_order_seq_cst`.

1852 **RETURN VALUE**

1853       These generic functions shall return the value pointed to by *object*.

1854 **ERRORS**

1855       No errors are defined.

1856 **EXAMPLES**

1857       None.

1858 **APPLICATION USAGE**

1859       None.

1860 **RATIONALE**

1861       None.

1862 **FUTURE DIRECTIONS**

1863       None.

1864 **SEE ALSO**

1865 XBD Section 4.12.1, <stdatomic.h>

## 1866 CHANGE HISTORY

1867 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

## 1868 NAME

1869 `atomic_signal_fence`, `atomic_thread_fence` — fence operations

## 1870 SYNOPSIS

```
1871 #include <stdatomic.h>
1872 void atomic_signal_fence(memory_order order);
1873 void atomic_thread_fence(memory_order order);
```

## 1874 DESCRIPTION

1875 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
1876 Any conflict between the requirements described here and the ISO C standard is  
1877 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1878 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1879 <stdatomic.h> header nor support these functions.

1880 The `atomic_signal_fence()` and `atomic_thread_fence()` functions provide synchronization  
1881 primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A  
1882 fence with acquire semantics is called an *acquire fence*; a fence with release semantics is  
1883 called a *release fence*.

1884 A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X*  
1885 and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X*  
1886 modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* or a value written  
1887 by any side effect in the hypothetical release sequence *X* would head if it were a release  
1888 operation.

1889 A release fence *A* synchronizes with an atomic operation *B* that performs an acquire  
1890 operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is  
1891 sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by  
1892 any side effect in the hypothetical release sequence *X* would head if it were a release  
1893 operation.

1894 An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with  
1895 an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced  
1896 before *B* and reads the value written by *A* or a value written by any side effect in the release  
1897 sequence headed by *A*.

1898 Depending on the value of *order*, the operation performed by `atomic_thread_fence()` shall:

- 1899 • have no effects, if *order* is equal to `memory_order_relaxed`;
- 1900 • be an acquire fence, if *order* is equal to `memory_order_acquire` or  
1901 `memory_order_consume`;
- 1902 • be a release fence, if *order* is equal to `memory_order_release`;

1903           • be both an acquire fence and a release fence, if *order* is equal to  
1904           memory\_order\_acq\_rel;

1905           • be a sequentially consistent acquire and release fence, if *order* is equal to  
1906           memory\_order\_seq\_cst.

1907           The *atomic\_signal\_fence()* function shall be equivalent to *atomic\_thread\_fence()*, except  
1908           that the resulting ordering constraints shall be established only between a thread and a signal  
1909           handler executed in the same thread.

#### 1910 **RETURN VALUE**

1911           These functions shall not return a value.

#### 1912 **ERRORS**

1913           No errors are defined.

#### 1914 **EXAMPLES**

1915           None.

#### 1916 **APPLICATION USAGE**

1917           The *atomic\_signal\_fence()* function can be used to specify the order in which actions  
1918           performed by the thread become visible to the signal handler. Implementation reorderings of  
1919           loads and stores are inhibited in the same way as with *atomic\_thread\_fence()*, but the  
1920           hardware fence instructions that *atomic\_thread\_fence()* would have inserted are not  
1921           emitted.

#### 1922 **RATIONALE**

1923           None.

#### 1924 **FUTURE DIRECTIONS**

1925           None.

#### 1926 **SEE ALSO**

1927           XBD Section 4.12.1, <stdatomic.h>

#### 1928 **CHANGE HISTORY**

1929           First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

#### 1930 **NAME**

1931           atomic\_store, atomic\_store\_explicit — atomically store a value in an object

#### 1932 **SYNOPSIS**

```
1933       #include <stdatomic.h>  
1934       void atomic_store(volatile A *object, C desired);  
1935       void atomic_store_explicit(volatile A *object, C desired,  
1936                                   memory_order order);
```

#### 1937 **DESCRIPTION**

1938           [*CX*] The functionality described on this reference page is aligned with the ISO C standard.  
1939           Any conflict between the requirements described here and the ISO C standard is

1940 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1941 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
1942 `<stdatomic.h>` header nor support these generic functions.

1943 The `atomic_store_explicit()` generic function shall atomically replace the value pointed to by  
1944 *object* with the value of *desired*. Memory shall be affected according to the value of *order*,  
1945 which the application shall ensure is not `memory_order_acquire`,  
1946 `memory_order_consume`, nor `memory_order_acq_rel`.

1947 The `atomic_store()` generic function shall be equivalent to `atomic_store_explicit()` called  
1948 with *order* set to `memory_order_seq_cst`.

1949 **RETURN VALUE**

1950 These generic functions shall not return a value.

1951 **ERRORS**

1952 No errors are defined.

1953 **EXAMPLES**

1954 None.

1955 **APPLICATION USAGE**

1956 None.

1957 **RATIONALE**

1958 None.

1959 **FUTURE DIRECTIONS**

1960 None.

1961 **SEE ALSO**

1962 XBD Section 4.12.1, `<stdatomic.h>`

1963 **CHANGE HISTORY**

1964 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

1965 Ref 7.28.1, 7.1.4 para 5

1966 On page 633 line 21891 insert a new `c16rtomb()` section:

1967 **NAME**

1968 `c16rtomb`, `c32rtomb` — convert a Unicode character code to a character (restartable)

1969 **SYNOPSIS**

1970 `#include <uchar.h>`

1971 `size_t c16rtomb(char *restrict s, char16_t c16,`  
1972 `mbstate_t *restrict ps);`

1973 `size_t c32rtomb(char *restrict s, char32_t c32,`  
1974 `mbstate_t *restrict ps);`

1975 **DESCRIPTION**

1976 [CX] The functionality described on this reference page is aligned with the ISO C standard.

1977 Any conflict between the requirements described here and the ISO C standard is  
1978 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

1979 If *s* is a null pointer, the *c16rtomb()* function shall be equivalent to the call:

1980 `c16rtomb(buf, L'\0', ps)`

1981 where *buf* is an internal buffer.

1982 If *s* is not a null pointer, the *c16rtomb()* function shall determine the number of bytes needed  
1983 to represent the character that corresponds to the wide character given by *c16* (including any  
1984 shift sequences), and store the resulting bytes in the array whose first element is pointed to  
1985 by *s*. At most {MB\_CUR\_MAX} bytes shall be stored. If *c16* is a null wide character, a null  
1986 byte shall be stored, preceded by any shift sequence needed to restore the initial shift state;  
1987 the resulting state described shall be the initial conversion state.

1988 If *ps* is a null pointer, the *c16rtomb()* function shall use its own internal **mbstate\_t** object,  
1989 which shall be initialized at program start-up to the initial conversion state. Otherwise, the  
1990 **mbstate\_t** object pointed to by *ps* shall be used to completely describe the current  
1991 conversion state of the associated character sequence.

1992 The behavior of this function is affected by the *LC\_CTYPE* category of the current locale.

1993 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

1994 The *c32rtomb()* function shall behave the same way as *c16rtomb()* except that the second  
1995 parameter shall be an object of type **char32\_t** instead of **char16\_t**. References to *c16* in the  
1996 above description shall apply as if they were *c32* when they are being read as describing  
1997 *c32rtomb()*.

1998 If called with a null *ps* argument, the *c16rtomb()* function need not be thread-safe; however,  
1999 such calls shall avoid data races with calls to *c16rtomb()* with a non-null argument and with  
2000 calls to all other functions.

2001 If called with a null *ps* argument, the *c32rtomb()* function need not be thread-safe; however,  
2002 such calls shall avoid data races with calls to *c32rtomb()* with a non-null argument and with  
2003 calls to all other functions.

2004 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx  
2005 calls *c16rtomb()* or *c32rtomb()* with a null pointer for *ps*.

## 2006 RETURN VALUE

2007 These functions shall return the number of bytes stored in the array object (including any  
2008 shift sequences). When *c16* or *c32* is not a valid wide character, an encoding error shall  
2009 occur. In this case, the function shall store the value of the macro [EILSEQ] in *errno* and  
2010 shall return (**size\_t**)-1; the conversion state is unspecified.

## 2011 ERRORS

2012 These function shall fail if:

2013 [EILSEQ] An invalid wide-character code is detected.

2014 These functions may fail if:

- 2015 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]
- 2016 **EXAMPLES**
- 2017 None.
- 2018 **APPLICATION USAGE**
- 2019 None.
- 2020 **RATIONALE**
- 2021 None.
- 2022 **FUTURE DIRECTIONS**
- 2023 None.
- 2024 **SEE ALSO**
- 2025 *mbrtoc16*
- 2026 XBD <**uchar.h**>
- 2027 **CHANGE HISTORY**
- 2028 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.
- 2029 Ref G.6 para 6, F.10.4.3, F.10.4.2, F.10 para 11
- 2030 On page 633 line 21905 section *cabs()*, add:
- 2031 [MXC]*cabs(x + iy)*, *cabs(y + ix)*, and *cabs(x - iy)* shall return exactly the same value.
- 2032 If  $z$  is  $\pm 0 \pm i0$ ,  $+0$  shall be returned.
- 2033 If the real or imaginary part of  $z$  is  $\pm\text{Inf}$ ,  $+\text{Inf}$  shall be returned, even if the other part is NaN.
- 2034 If the real or imaginary part of  $z$  is NaN and the other part is not  $\pm\text{Inf}$ , NaN shall be returned.
- 2035 [/MXC]
- 2036 Ref G.6.1.1
- 2037 On page 634 line 21935 section *cacos()*, add:
- 2038 [MXC]*cacos(conj(z))*, *cacosf(conjf(z))* and *cacosl(conjl(z))* shall return exactly the same
- 2039 value as *conj(cacos(z))*, *conjf(cacosf(z))* and *conjl(cacosl(z))*, respectively, including for the
- 2040 special values of  $z$  below.
- 2041 If  $z$  is  $\pm 0 + i0$ ,  $\pi/2 - i0$  shall be returned.
- 2042 If  $z$  is  $\pm 0 + i\text{NaN}$ ,  $\pi/2 + i\text{NaN}$  shall be returned.
- 2043 If  $z$  is  $x + i\text{Inf}$  where  $x$  is finite,  $\pi/2 - i\text{Inf}$  shall be returned.
- 2044 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
- 2045 floating-point exception may be raised.
- 2046 If  $z$  is  $-\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $\pi - i\text{Inf}$  shall be returned.



- 2047 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+0 - i\text{Inf}$  shall be returned.
- 2048 If  $z$  is  $-\text{Inf} + i\text{Inf}$ ,  $3\pi/4 - i\text{Inf}$  shall be returned.
- 2049 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $\pi/4 - i\text{Inf}$  shall be returned.
- 2050 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  
2051 is unspecified.
- 2052 If  $z$  is  $\text{NaN} + iy$  where  $y$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2053 point exception may be raised.
- 2054 If  $z$  is  $\text{NaN} + i\text{Inf}$ ,  $\text{NaN} - i\text{Inf}$  shall be returned.
- 2055 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} - i\text{NaN}$  shall be returned.[/MXC]
- 2056 Ref G.6.2.1
- 2057 On page 635 line 21966 section `cacosh()`, add:
- 2058 [MXC]`cacosh(conj(z))`, `cacoshf(conjf(z))` and `cacoshl(conjl(z))` shall return exactly the same  
2059 value as `conj(cacosh(z))`, `conjf(cacoshf(z))` and `conjl(cacoshl(z))`, respectively, including for  
2060 the special values of  $z$  below.
- 2061 If  $z$  is  $\pm 0 + i0$ ,  $+0 + i\pi/2$  shall be returned.
- 2062 If  $z$  is  $x + i\text{Inf}$  where  $x$  is finite,  $+\text{Inf} + i\pi/2$  shall be returned.
- 2063 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  
2064 unspecified.
- 2065 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  
2066 floating-point exception may be raised.
- 2067 If  $z$  is  $-\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+\text{Inf} + i\pi$  shall be returned.
- 2068 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+\text{Inf} + i0$  shall be returned.
- 2069 If  $z$  is  $-\text{Inf} + i\text{Inf}$ ,  $+\text{Inf} + i3\pi/4$  shall be returned.
- 2070 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $+\text{Inf} + i\pi/4$  shall be returned.
- 2071 If  $z$  is  $\pm\text{Inf} + i\text{NaN}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.
- 2072 If  $z$  is  $\text{NaN} + iy$  where  $y$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2073 point exception may be raised.
- 2074 If  $z$  is  $\text{NaN} + i\text{Inf}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.
- 2075 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]
- 2076 Ref 7.26.2.1

2077 On page 637 line 21989 insert the following new `call_once()` section:

2078 **NAME**

2079 `call_once` — dynamic package initialization

2080 **SYNOPSIS**

2081 `#include <threads.h>`

2082 `void call_once(once_flag *flag, void (*init_routine)(void));`  
2083 `once_flag flag = ONCE_FLAG_INIT;`

2084 **DESCRIPTION**

2085 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
2086 Any conflict between the requirements described here and the ISO C standard is  
2087 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2088 The `call_once()` function shall use the **once\_flag** pointed to by `flag` to ensure that  
2089 `init_routine` is called exactly once, the first time the `call_once()` function is called with that  
2090 value of `flag`. Completion of an effective call to the `call_once()` function shall synchronize  
2091 with all subsequent calls to the `call_once()` function with the same value of `flag`.

2092 [CX]The `call_once()` function is not a cancellation point. However, if `init_routine` is a  
2093 cancellation point and is canceled, the effect on `flag` shall be as if `call_once()` was never  
2094 called.

2095 If the call to `init_routine` is terminated by a call to `longjmp()` or `siglongjmp()`, the behavior is  
2096 undefined.

2097 The behavior of `call_once()` is undefined if `flag` has automatic storage duration or is not  
2098 initialized by `ONCE_FLAG_INIT`.

2099 The `call_once()` function shall not be affected if the calling thread executes a signal handler  
2100 during the call.[/CX]

2101 **RETURN VALUE**

2102 The `call_once()` function shall not return a value.

2103 **ERRORS**

2104 No errors are defined.

2105 **EXAMPLES**

2106 None.

2107 **APPLICATION USAGE**

2108 If `init_routine` recursively calls `call_once()` with the same `flag`, the recursive call will not call  
2109 the specified `init_routine`, and thus the specified `init_routine` will not complete, and thus the  
2110 recursive call to `call_once()` will not return. Use of `longjmp()` or `siglongjmp()` within an  
2111 `init_routine` to jump to a point outside of `init_routine` prevents `init_routine` from returning.

2112 **RATIONALE**

2113 For dynamic library initialization in a multi-threaded process, if an initialization flag is used  
2114 the flag needs to be protected against modification by multiple threads simultaneously  
2115 calling into the library. This can be done by using a statically-initialized mutex. However,

2116 the better solution is to use *call\_once()* or *pthread\_once()* which are designed for exactly  
2117 this purpose, for example:

```
2118 #include <threads.h>
2119 static once_flag random_is_initialized = ONCE_FLAG_INIT;
2120 extern void initialize_random(void);

2121 int random_function()
2122 {
2123     call_once(&random_is_initialized, initialize_random);
2124     ...
2125     /* Operations performed after initialization. */
2126 }
```

2127 The *call\_once()* function is not affected by signal handlers for the reasons stated in [xref to  
2128 XRAT B.2.3].

### 2129 FUTURE DIRECTIONS

2130 None.

### 2131 SEE ALSO

2132 *pthread\_once*

2133 XBD Section 4.12.2, <**threads.h**>

### 2134 CHANGE HISTORY

2135 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2136 Ref 7.22.3 para 1

2137 On page 637 line 22002 section *calloc()*, change:

2138 a pointer to any type of object

2139 to:

2140 a pointer to any type of object with a fundamental alignment requirement

2141 ~~Ref 7.22.3 para 1~~

2142 ~~On page 637 line 22007 section *calloc()*, change:~~

2143 ~~either a null pointer shall be returned, or ...~~

2144 ~~to:~~

2145 ~~either a null pointer shall be returned to indicate an error, or ...~~

2146 Ref 7.22.3 para 2

2147 On page 637 line 22008 section *calloc()*, add a new paragraph:

2148 For purposes of determining the existence of a data race, *calloc()* shall behave as though it  
2149 accessed only memory locations accessible through its arguments and not other static

2150 duration storage. The function may, however, visibly modify the storage that it allocates.  
2151 Calls to *aligned\_alloc()*, *calloc()*, *free()*, *malloc()*, [ADV]*posix\_memalign()*,[/ADV]  
2152 [CX]*reallocarray()*,[/CX] and *realloc()* that allocate or deallocate a particular region of  
2153 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such  
2154 deallocation call shall synchronize with the next allocation (if any) in this order.

2155 Ref 7.22.3.1

2156 On page 637 line 22029 section *calloc()*, add *aligned\_alloc* to the SEE ALSO section.

2157 Ref G.6 para 6, F.10.1.4, F.10 para 11

2158 On page 639 line 22055 section *carg()*, add:

2159 [MXC]If  $z$  is  $-0 \pm i0$ ,  $\pm\pi$  shall be returned.

2160 If  $z$  is  $+0 \pm i0$ ,  $\pm 0$  shall be returned.

2161 If  $z$  is  $x \pm i0$  where  $x$  is negative,  $\pm\pi$  shall be returned.

2162 If  $z$  is  $x \pm i0$  where  $x$  is positive,  $\pm 0$  shall be returned.

2163 If  $z$  is  $\pm 0 + iy$  where  $y$  is negative,  $-\pi/2$  shall be returned.

2164 If  $z$  is  $\pm 0 + iy$  where  $y$  is positive,  $\pi/2$  shall be returned.

2165 If  $z$  is  $-\text{Inf} \pm iy$  where  $y$  is positive and finite,  $\pm\pi$  shall be returned.

2166 If  $z$  is  $+\text{Inf} \pm iy$  where  $y$  is positive and finite,  $\pm 0$  shall be returned.

2167 If  $z$  is  $x \pm i\text{Inf}$  where  $x$  is finite,  $\pm\pi/2$  shall be returned.

2168 If  $z$  is  $-\text{Inf} \pm i\text{Inf}$ ,  $\pm 3\pi/4$  shall be returned.

2169 If  $z$  is  $+\text{Inf} \pm i\text{Inf}$ ,  $\pm\pi/4$  shall be returned.

2170 If the real or imaginary part of  $z$  is NaN, NaN shall be returned.[/MXC]

2171 Ref G.6 para 7, G.6.2.2

2172 On page 640 line 22086 section *casin()*, add:

2173 [MXC]*casin(conj(iz))*, *casinf(conjf(iz))* and *casinl(conjl(iz))* shall return exactly the same  
2174 value as *conj(casin(iz))*, *conjf(casinf(iz))* and *conjl(casinl(iz))*, respectively, and *casin(-iz)*,  
2175 *casinf(-iz)* and *casinl(-iz)* shall return exactly the same value as  $-casin(iz)$ ,  $-casinf(iz)$  and  
2176  $-casinl(iz)$ , respectively, including for the special values of  $iz$  below.

2177 If  $iz$  is  $+0 + i0$ ,  $-i$  ( $0 + i0$ ) shall be returned.

2178 If  $iz$  is  $x + i\text{Inf}$  where  $x$  is positive-signed and finite,  $-i$  ( $+\text{Inf} + i\pi/2$ ) shall be returned.

2179 If  $iz$  is  $x + i\text{NaN}$  where  $x$  is finite,  $-i$  ( $\text{NaN} + i\text{NaN}$ ) shall be returned and the invalid  
2180 floating-point exception may be raised.

2181 If  $iz$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $-i$  ( $+\text{Inf} + i0$ ) shall be returned.

2182 If  $iz$  is  $+\text{Inf} + i\text{Inf}$ ,  $-i (+\text{Inf} + i\pi/4)$  shall be returned.

2183 If  $iz$  is  $+\text{Inf} + i\text{NaN}$ ,  $-i (+\text{Inf} + i\text{NaN})$  shall be returned.

2184 If  $iz$  is  $\text{NaN} + i0$ ,  $-i (\text{NaN} + i0)$  shall be returned.

2185 If  $iz$  is  $\text{NaN} + iy$  where  $y$  is non-zero and finite,  $-i (\text{NaN} + i\text{NaN})$  shall be returned and the  
2186 invalid floating-point exception may be raised.

2187 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  
2188 result is unspecified.

2189 If  $iz$  is  $\text{NaN} + i\text{NaN}$ ,  $-i (\text{NaN} + i\text{NaN})$  shall be returned.[/MXC]

2190 Ref G.6 para 7  
2191 On page 640 line 22094 section `casin()`, change RATIONALE from:

2192 None.

2193 to:

2194 The MXC special cases for `casin()` are derived from those for `casinh()` by applying the  
2195 formula  $\text{casin}(z) = -i \text{casinh}(iz)$ .

2196 Ref G.6.2.2  
2197 On page 641 line 22118 section `casinh()`, add:

2198 [MXC]`casinh(conj(z))`, `casinhf(conjf(z))` and `casinhl(conjl(z))` shall return exactly the same  
2199 value as `conj(casinh(z))`, `conjf(casinhf(z))` and `conjl(casinhl(z))`, respectively, and `casinh(-z)`,  
2200 `casinhf(-z)` and `casinhl(-z)` shall return exactly the same value as  $-\text{casinh}(z)$ ,  $-\text{casinhf}(z)$   
2201 and  $-\text{casinhl}(z)$ , respectively, including for the special values of  $z$  below.

2202 If  $z$  is  $+0 + i0$ ,  $0 + i0$  shall be returned.

2203 If  $z$  is  $x + i\text{Inf}$  where  $x$  is positive-signed and finite,  $+\text{Inf} + i\pi/2$  shall be returned.

2204 If  $z$  is  $x + i\text{NaN}$  where  $x$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2205 point exception may be raised.

2206 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+\text{Inf} + i0$  shall be returned.

2207 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $+\text{Inf} + i\pi/4$  shall be returned.

2208 If  $z$  is  $+\text{Inf} + i\text{NaN}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.

2209 If  $z$  is  $\text{NaN} + i0$ ,  $\text{NaN} + i0$  shall be returned.

2210 If  $z$  is  $\text{NaN} + iy$  where  $y$  is non-zero and finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid  
2211 floating-point exception may be raised.

2212 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

- 2213 unspecified.
- 2214 If  $z$  is NaN + iNaN, NaN + iNaN shall be returned.[/MXC]
- 2215 Ref G.6 para 7, G.6.2.3
- 2216 On page 643 line 22157 section `catan`, add:
- 2217 [MXC]`catan(conj(iz))`, `catanf(conjf(iz))` and `catanl(conjl(iz))` shall return exactly the same  
 2218 value as `conj(catan(iz))`, `conjf(catanf(iz))` and `conjl(catanl(iz))`, respectively, and `catan(-iz)`,  
 2219 `catanf(-iz)` and `catanl(-iz)` shall return exactly the same value as `-catan(iz)`, `-catanf(iz)` and  
 2220 `-catanl(iz)`, respectively, including for the special values of  $iz$  below.
- 2221 If  $iz$  is  $+0 + i0$ ,  $-i (+0 + i0)$  shall be returned.
- 2222 If  $iz$  is  $+0 + iNaN$ ,  $-i (+0 + iNaN)$  shall be returned.
- 2223 If  $iz$  is  $+1 + i0$ ,  $-i (+Inf + i0)$  shall be returned and the divide-by-zero floating-point  
 2224 exception shall be raised.
- 2225 If  $iz$  is  $x + iInf$  where  $x$  is positive-signed and finite,  $-i (+0 + i\pi/2)$  shall be returned.
- 2226 If  $iz$  is  $x + iNaN$  where  $x$  is non-zero and finite,  $-i (NaN + iNaN)$  shall be returned and the  
 2227 invalid floating-point exception may be raised.
- 2228 If  $iz$  is  $+Inf + iy$  where  $y$  is positive-signed and finite,  $-i (+0 + i\pi/2)$  shall be returned.
- 2229 If  $iz$  is  $+Inf + iInf$ ,  $-i (+0 + i\pi/2)$  shall be returned.
- 2230 If  $iz$  is  $+Inf + iNaN$ ,  $-i (+0 + iNaN)$  shall be returned.
- 2231 If  $iz$  is  $NaN + iy$  where  $y$  is finite,  $-i (NaN + iNaN)$  shall be returned and the invalid  
 2232 floating-point exception may be raised.
- 2233 If  $iz$  is  $NaN + iInf$ ,  $-i (\pm 0 + i\pi/2)$  shall be returned; the sign of the imaginary part of the  
 2234 result is unspecified.
- 2235 If  $iz$  is  $NaN + iNaN$ ,  $-i (NaN + iNaN)$  shall be returned.[/MXC]
- 2236 Ref G.6 para 7
- 2237 On page 643 line 22165 section `catan()`, change RATIONALE from:
- 2238 None.
- 2239 to:
- 2240 The MXC special cases for `catan()` are derived from those for `catanh()` by applying the  
 2241 formula  $catan(z) = -i catanh(iz)$ .
- 2242 Ref G.6.2.3
- 2243 On page 644 line 22189 section `catanh`, add:
- 2244 [MXC]`catanh(conj(z))`, `catanhf(conjf(z))` and `catanhl(conjl(z))` shall return exactly the same

- 2245 value as  $\text{conj}(\text{catanh}(z))$ ,  $\text{conjf}(\text{catanhf}(z))$  and  $\text{conjl}(\text{catanhl}(z))$ , respectively, and  
 2246  $\text{catanh}(-z)$ ,  $\text{catanhf}(-z)$  and  $\text{catanhl}(-z)$  shall return exactly the same value as  $-\text{catanh}(z)$ ,  
 2247  $-\text{catanhf}(z)$  and  $-\text{catanhl}(z)$ , respectively, including for the special values of  $z$  below.
- 2248 If  $z$  is  $+0 + i0$ ,  $+0 + i0$  shall be returned.
- 2249 If  $z$  is  $+0 + i\text{NaN}$ ,  $+0 + i\text{NaN}$  shall be returned.
- 2250 If  $z$  is  $+1 + i0$ ,  $+\text{Inf} + i0$  shall be returned and the divide-by-zero floating-point exception  
 2251 shall be raised.
- 2252 If  $z$  is  $x + i\text{Inf}$  where  $x$  is positive-signed and finite,  $+0 + i\pi/2$  shall be returned.
- 2253 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  
 2254 floating-point exception may be raised.
- 2255 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+0 + i\pi/2$  shall be returned.
- 2256 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $+0 + i\pi/2$  shall be returned.
- 2257 If  $z$  is  $+\text{Inf} + i\text{NaN}$ ,  $+0 + i\text{NaN}$  shall be returned.
- 2258 If  $z$  is  $\text{NaN} + iy$  where  $y$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
 2259 point exception may be raised.
- 2260 If  $z$  is  $\text{NaN} + i\text{Inf}$ ,  $\pm 0 + i\pi/2$  shall be returned; the sign of the real part of the result is  
 2261 unspecified.
- 2262 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]
- 2263 Ref G.6 para 7, G.6.2.4  
 2264 On page 652 line 22426 section  $\text{ccos}()$ , add:
- 2265 [MXC] $\text{ccos}(\text{conj}(iz))$ ,  $\text{ccosf}(\text{conjf}(iz))$  and  $\text{ccosl}(\text{conjl}(iz))$  shall return exactly the same value  
 2266 as  $\text{conj}(\text{ccos}(iz))$ ,  $\text{conjf}(\text{ccosf}(iz))$  and  $\text{conjl}(\text{ccosl}(iz))$ , respectively, and  $\text{ccos}(-iz)$ ,  $\text{ccosf}(-iz)$   
 2267 and  $\text{ccosl}(-iz)$  shall return exactly the same value as  $\text{ccos}(iz)$ ,  $\text{ccosf}(iz)$  and  $\text{ccosl}(iz)$ ,  
 2268 respectively, including for the special values of  $iz$  below.
- 2269 If  $iz$  is  $+0 + i0$ ,  $1 + i0$  shall be returned.
- 2270 If  $iz$  is  $+0 + i\text{Inf}$ ,  $\text{NaN} \pm i0$  shall be returned and the invalid floating-point exception shall be  
 2271 raised; the sign of the imaginary part of the result is unspecified.
- 2272 If  $iz$  is  $+0 + i\text{NaN}$ ,  $\text{NaN} \pm i0$  shall be returned; the sign of the imaginary part of the result is  
 2273 unspecified.
- 2274 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  
 2275 floating-point exception shall be raised.
- 2276 If  $iz$  is  $x + i\text{NaN}$  where  $x$  is non-zero and finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the  
 2277 invalid floating-point exception may be raised.

- 2278 If  $iz$  is  $+\text{Inf} + i0$ ,  $+\text{Inf} + i0$  shall be returned.
- 2279 If  $iz$  is  $+\text{Inf} + iy$  where  $y$  is non-zero and finite,  $+\text{Inf} (\cos(y) + i\sin(y))$  shall be returned.
- 2280 If  $iz$  is  $+\text{Inf} + i\text{Inf}$ ,  $\pm\text{Inf} + i\text{NaN}$  shall be returned and the invalid floating-point exception  
2281 shall be raised; the sign of the real part of the result is unspecified.
- 2282 If  $iz$  is  $+\text{Inf} + i\text{NaN}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.
- 2283 If  $iz$  is  $\text{NaN} + i0$ ,  $\text{NaN} \pm i0$  shall be returned; the sign of the imaginary part of the result is  
2284 unspecified.
- 2285 If  $iz$  is  $\text{NaN} + iy$  where  $y$  is any non-zero number,  $\text{NaN} + i\text{NaN}$  shall be returned and the  
2286 invalid floating-point exception may be raised.
- 2287 If  $iz$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]
- 2288 Ref G.6 para 7  
2289 On page 652 line 22434 section `ccos()`, change RATIONALE from:
- 2290 None.
- 2291 to:
- 2292 The MXC special cases for `ccos()` are derived from those for `ccosh()` by applying the  
2293 formula  $\text{ccos}(z) = \text{ccosh}(iz)$ .
- 2294 Ref G.6.2.4  
2295 On page 653 line 22455 section `ccosh()`, add:
- 2296 [MXC]`ccosh(conj(z))`, `ccoshf(conjf(z))` and `ccoshl(conjl(z))` shall return exactly the same  
2297 value as `conj(ccosh(z))`, `conjf(ccoshf(z))` and `conjl(ccoshl(z))`, respectively, and `ccosh(-z)`,  
2298 `ccoshf(-z)` and `ccoshl(-z)` shall return exactly the same value as `ccosh(z)`, `ccoshf(z)` and  
2299 `ccoshl(z)`, respectively, including for the special values of  $z$  below.
- 2300 If  $z$  is  $+0 + i0$ ,  $1 + i0$  shall be returned.
- 2301 If  $z$  is  $+0 + i\text{Inf}$ ,  $\text{NaN} \pm i0$  shall be returned and the invalid floating-point exception shall be  
2302 raised; the sign of the imaginary part of the result is unspecified.
- 2303 If  $z$  is  $+0 + i\text{NaN}$ ,  $\text{NaN} \pm i0$  shall be returned; the sign of the imaginary part of the result is  
2304 unspecified.
- 2305 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  
2306 floating-point exception shall be raised.
- 2307 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  
2308 floating-point exception may be raised.
- 2309 If  $z$  is  $+\text{Inf} + i0$ ,  $+\text{Inf} + i0$  shall be returned.
- 2310 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is non-zero and finite,  $+\text{Inf} (\cos(y) + i\sin(y))$  shall be returned.



2311 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $\pm\text{Inf} + i\text{NaN}$  shall be returned and the invalid floating-point exception  
2312 shall be raised; the sign of the real part of the result is unspecified.

2313 If  $z$  is  $+\text{Inf} + i\text{NaN}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.

2314 If  $z$  is  $\text{NaN} + i0$ ,  $\text{NaN} \pm i0$  shall be returned; the sign of the imaginary part of the result is  
2315 unspecified.

2316 If  $z$  is  $\text{NaN} + iy$  where  $y$  is any non-zero number,  $\text{NaN} + i\text{NaN}$  shall be returned and the  
2317 invalid floating-point exception may be raised.

2318 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]

2319 Ref F.10.6.1 para 4  
2320 On page 655 line 22489 section `ceil()`, add a new paragraph:

2321 [MX]These functions may raise the inexact floating-point exception for finite non-integer  
2322 arguments.[/MX]

2323 Ref F.10.6.1 para 2  
2324 On page 655 line 22491 section `ceil()`, change:

2325 [MX]The result shall have the same sign as  $x$ .[/MX]

2326 to:

2327 [MX]The returned value shall be independent of the current rounding direction mode and  
2328 shall have the same sign as  $x$ .[/MX]

2329 Ref F.10.6.1 para 4  
2330 On page 655 line 22504 section `ceil()`, delete from APPLICATION USAGE:

2331 These functions may raise the inexact floating-point exception if the result differs in value  
2332 from the argument.

2333 Ref G.6.3.1  
2334 On page 657 line 22539 section `cexp()`, add:

2335 [MXC]`cexp(conj(z))`, `cexpf(conjf(z))` and `cexpl(conjl(z))` shall return exactly the same value  
2336 as `conj(cexp(z))`, `conjf(cexpf(z))` and `conjl(cexpl(z))`, respectively, including for the special  
2337 values of  $z$  below.

2338 If  $z$  is  $\pm 0 + i0$ ,  $1 + i0$  shall be returned.

2339 If  $z$  is  $x + i\text{Inf}$  where  $x$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-point  
2340 exception shall be raised.

2341 If  $z$  is  $x + i\text{NaN}$  where  $x$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2342 point exception may be raised.

2343 If  $z$  is  $+\text{Inf} + i0$ ,  $+\text{Inf} + i0$  shall be returned.

- 2344 If  $z$  is  $-\text{Inf} + iy$  where  $y$  is finite,  $+0 (\cos(y) + i\sin(y))$  shall be returned.
- 2345 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is non-zero and finite,  $+\text{Inf} (\cos(y) + i\sin(y))$  shall be returned.
- 2346 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  
2347 result are unspecified.
- 2348 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $\pm\text{Inf} + i\text{NaN}$  shall be returned and the invalid floating-point exception  
2349 shall be raised; the sign of the real part of the result is unspecified.
- 2350 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  
2351 result are unspecified.
- 2352 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  
2353 unspecified.
- 2354 If  $z$  is  $\text{NaN} + i0$ ,  $\text{NaN} + i0$  shall be returned.
- 2355 If  $z$  is  $\text{NaN} + iy$  where  $y$  is any non-zero number,  $\text{NaN} + i\text{NaN}$  shall be returned and the  
2356 invalid floating-point exception may be raised.
- 2357 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]
- 2358 Ref 7.26.5.7  
2359 On page 679 line 23268 section `clock_getres()`, change:
- 2360 including the `nanosleep()` function
- 2361 to:
- 2362 including the `nanosleep()` and `thrd_sleep()` functions
- 2363 Ref G.6.3.2  
2364 On page 687 line 23495 section `clog()`, add:
- 2365 [MXC]`clog(conj(z))`, `clogf(conjf(z))` and `clogl(conjl(z))` shall return exactly the same value as  
2366 `conj(clog(z))`, `conjf(clogf(z))` and `conjl(clogl(z))`, respectively, including for the special  
2367 values of  $z$  below.
- 2368 If  $z$  is  $-0 + i0$ ,  $-\text{Inf} + i\pi$  shall be returned and the divide-by-zero floating-point exception  
2369 shall be raised.
- 2370 If  $z$  is  $+0 + i0$ ,  $-\text{Inf} + i0$  shall be returned and the divide-by-zero floating-point exception  
2371 shall be raised.
- 2372 If  $z$  is  $x + i\text{Inf}$  where  $x$  is finite,  $+\text{Inf} + i\pi/2$  shall be returned.
- 2373 If  $z$  is  $x + i\text{NaN}$  where  $x$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2374 point exception may be raised.
- 2375 If  $z$  is  $-\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+\text{Inf} + i\pi$  shall be returned.

2376 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+\text{Inf} + i0$  shall be returned.

2377 If  $z$  is  $-\text{Inf} + i\text{Inf}$ ,  $+\text{Inf} + i3\pi/4$  shall be returned.

2378 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $+\text{Inf} + i\pi/4$  shall be returned.

2379 If  $z$  is  $\pm\text{Inf} + i\text{NaN}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.

2380 If  $z$  is  $\text{NaN} + iy$  where  $y$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2381 point exception may be raised.

2382 If  $z$  is  $\text{NaN} + i\text{Inf}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.

2383 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]

2384 Ref 7.26.3

2385 On page 698 line 23854 insert the following new `cond_*`() sections:

2386 [Note to reviewers: changes to `cond\_broadcast` and `cond\_signal` may be needed depending on the](#)  
2387 [outcome of Mantis bug 609.](#)

## 2388 NAME

2389 `cond_broadcast`, `cond_signal` — broadcast or signal a condition

## 2390 SYNOPSIS

2391 `#include <threads.h>`

2392 `int cond_broadcast(cond_t *cond);`

2393 `int cond_signal(cond_t *cond);`

## 2394 DESCRIPTION

2395 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
2396 Any conflict between the requirements described here and the ISO C standard is  
2397 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2398 The `cond_broadcast()` function shall unblock all of the threads that are blocked on the  
2399 condition variable pointed to by `cond` at the time of the call.

2400 The `cond_signal()` function shall unblock one of the threads that are blocked on the condition  
2401 variable pointed to by `cond` at the time of the call (if any threads are blocked on `cond`).

2402 If no threads are blocked on the condition variable pointed to by `cond` at the time of the call,  
2403 these functions shall have no effect and shall return `thrd_success`.

2404 [CX]If more than one thread is blocked on a condition variable, the scheduling policy shall  
2405 determine the order in which threads are unblocked. When each thread unblocked as a result  
2406 of a `cond_broadcast()` or `cond_signal()` returns from its call to `cond_wait()` or `cond_timedwait()`,  
2407 the thread shall own the mutex with which it called `cond_wait()` or `cond_timedwait()`. The  
2408 thread(s) that are unblocked shall contend for the mutex according to the scheduling policy  
2409 (if applicable), and as if each had called `mtx_lock()`.

2410 The `cond_broadcast()` and `cond_signal()` functions can be called by a thread whether or not it

2411 currently owns the mutex that threads calling *cnd\_wait()* or *cnd\_timedwait()* have associated  
2412 with the condition variable during their waits; however, if predictable scheduling behavior is  
2413 required, then that mutex shall be locked by the thread calling *cnd\_broadcast()* or  
2414 *cnd\_signal()*.

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

2417 The behavior is undefined if the value specified by the *cond* argument to *cnd\_broadcast()* or  
2418 *cnd\_signal()* does not refer to an initialized condition variable.

#### 2419 **RETURN VALUE**

2420 These functions shall return *thrd\_success* on success, or *thrd\_error* if the request  
2421 could not be honored.

#### 2422 **ERRORS**

2423 No errors are defined.

#### 2424 **EXAMPLES**

2425 None.

#### 2426 **APPLICATION USAGE**

2427 See the APPLICATION USAGE section for *pthread\_cond\_broadcast()*, substituting  
2428 *cnd\_broadcast()* for *pthread\_cond\_broadcast()* and *cnd\_signal()* for *pthread\_cond\_signal()*.

#### 2429 **RATIONALE**

2430 As for *pthread\_cond\_broadcast()* and *pthread\_cond\_signal()*, spurious wakeups may occur  
2431 with *cnd\_broadcast()* and *cnd\_signal()*, necessitating that applications code a predicate-  
2432 testing-loop around the condition wait. (See the RATIONALE section for  
2433 *pthread\_cond\_broadcast()*.)

2434 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT  
2435 B.2.3].

#### 2436 **FUTURE DIRECTIONS**

2437 None.

#### 2438 **SEE ALSO**

2439 *cnd\_destroy*, *cnd\_timedwait*, *pthread\_cond\_broadcast*

2440 XBD Section 4.12.2, <**threads.h**>

#### 2441 **CHANGE HISTORY**

2442 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

#### 2443 **NAME**

2444 *cnd\_destroy*, *cnd\_init* — destroy and initialize condition variables

#### 2445 **SYNOPSIS**

2446 `#include <threads.h>`

2447 `void cnd_destroy(cnd_t *cond);`

2448 `int cnd_init(cnd_t *cond);`

2449 **DESCRIPTION**

2450 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
2451 Any conflict between the requirements described here and the ISO C standard is  
2452 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2453 The `cnd_destroy()` function shall release all resources used by the condition variable pointed  
2454 to by `cond`. It shall be safe to destroy an initialized condition variable upon which no threads  
2455 are currently blocked. Attempting to destroy a condition variable upon which other threads  
2456 are currently blocked results in undefined behavior. A destroyed condition variable object  
2457 can be reinitialized using `cnd_init()`; the results of otherwise referencing the object after it  
2458 has been destroyed are undefined. The behavior is undefined if the value specified by the  
2459 `cond` argument to `cnd_destroy()` does not refer to an initialized condition variable.

2460 The `cnd_init()` function shall initialize a condition variable. If it succeeds it shall set the  
2461 variable pointed to by `cond` to a value that uniquely identifies the newly initialized condition  
2462 variable. Attempting to initialize an already initialized condition variable results in  
2463 undefined behavior. A thread that calls `cnd_wait()` on a newly initialized condition variable  
2464 shall block.

2465 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for  
2466 further requirements.

2467 These functions shall not be affected if the calling thread executes a signal handler during  
2468 the call.[/CX]

2469 **RETURN VALUE**

2470 The `cnd_destroy()` function shall not return a value.

2471 The `cnd_init()` function shall return `thrd_success` on success, or `thrd_nomem` if no  
2472 memory could be allocated for the newly created condition, or `thrd_error` if the request  
2473 could not be honored.

2474 **ERRORS**

2475 See RETURN VALUE.

2476 **EXAMPLES**

2477 None.

2478 **APPLICATION USAGE**

2479 None.

2480 **RATIONALE**

2481 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT  
2482 B.2.3].

2483 **FUTURE DIRECTIONS**

2484 None.

2485 **SEE ALSO**

2486 `cnd_broadcast`, `cnd_timedwait`

2487 XBD <threads.h>

2488 **CHANGE HISTORY**

2489 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2490 **NAME**

2491 `cnd_timedwait`, `cnd_wait` — wait on a condition

2492 **SYNOPSIS**

```
2493 #include <threads.h>
2494 int cnd_timedwait(cnd_t * restrict cond, mtx_t * restrict mtx,
2495                  const struct timespec * restrict ts);
2496 int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

2497 **DESCRIPTION**

2498 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
2499 Any conflict between the requirements described here and the ISO C standard is  
2500 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

2501 The `cnd_timedwait()` function shall atomically unlock the mutex pointed to by `mtx` and block  
2502 until the condition variable pointed to by `cond` is signaled by a call to `cnd_signal()` or to  
2503 `cnd_broadcast()`, or until after the `TIME_UTC`-based calendar time pointed to by `ts`, or until  
2504 it is unblocked due to an unspecified reason.

2505 The `cnd_wait()` function shall atomically unlock the mutex pointed to by `mtx` and block until  
2506 the condition variable pointed to by `cond` is signaled by a call to `cnd_signal()` or to  
2507 `cnd_broadcast()`, or until it is unblocked due to an unspecified reason.

2508 [CX]Atomically here means "atomically with respect to access by another thread to the  
2509 mutex and then the condition variable". That is, if another thread is able to acquire the mutex  
2510 after the about-to-block thread has released it, then a subsequent call to `cnd_broadcast()` or  
2511 `cnd_signal()` in that thread shall behave as if it were issued after the about-to-block thread  
2512 has blocked.[/CX]

2513 When the calling thread becomes unblocked, these functions shall lock the mutex pointed to  
2514 by `mtx` before they return. The application shall ensure that the mutex pointed to by `mtx` is  
2515 locked by the calling thread before it calls these functions.

2516 When using condition variables there is always a Boolean predicate involving shared  
2517 variables associated with each condition wait that is true if the thread should proceed.  
2518 Spurious wakeups from the `cnd_timedwait()` and `cnd_wait()` functions may occur. Since the  
2519 return from `cnd_timedwait()` or `cnd_wait()` does not imply anything about the value of this  
2520 predicate, the predicate should be re-evaluated upon such return.

2521 When a thread waits on a condition variable, having specified a particular mutex to either  
2522 the `cnd_timedwait()` or the `cnd_wait()` operation, a dynamic binding is formed between that  
2523 mutex and condition variable that remains in effect as long as at least one thread is blocked  
2524 on the condition variable. During this time, the effect of an attempt by any thread to wait on  
2525 that condition variable using a different mutex is undefined. Once all waiting threads have  
2526 been unblocked (as by the `cnd_broadcast()` operation), the next wait operation on  
2527 that condition variable shall form a new dynamic binding with the mutex specified by that

2528 wait operation. Even though the dynamic binding between condition variable and mutex  
2529 might be removed or replaced between the time a thread is unblocked from a wait on the  
2530 condition variable and the time that it returns to the caller or begins cancellation cleanup, the  
2531 unblocked thread shall always re-acquire the mutex specified in the condition wait operation  
2532 call from which it is returning.

2533 [CX]A condition wait (whether timed or not) is a cancellation point. When the cancelability  
2534 type of a thread is set to PTHREAD\_CANCEL\_DEFERRED, a side-effect of acting upon a  
2535 cancellation request while in a condition wait is that the mutex is (in effect) re-acquired  
2536 before calling the first cancellation cleanup handler. The effect is as if the thread were  
2537 unblocked, allowed to execute up to the point of returning from the call to *cond\_timedwait()*  
2538 or *cond\_wait()*, but at that point notices the cancellation request and instead of returning to  
2539 the caller of *cond\_timedwait()* or *cond\_wait()*, starts the thread cancellation activities, which  
2540 includes calling cancellation cleanup handlers.

2541 A thread that has been unblocked because it has been canceled while blocked in a call to  
2542 *cond\_timedwait()* or *cond\_wait()* shall not consume any condition signal that may be directed  
2543 concurrently at the condition variable if there are other threads blocked on the condition  
2544 variable.[/CX]

2545 When *cond\_timedwait()* times out, it shall nonetheless release and re-acquire the mutex  
2546 referenced by mutex, and may consume a condition signal directed concurrently at the  
2547 condition variable.

2548 [CX]These functions shall not be affected if the calling thread executes a signal handler  
2549 during the call, except that if a signal is delivered to a thread waiting for a condition  
2550 variable, upon return from the signal handler either the thread shall resume waiting for the  
2551 condition variable as if it was not interrupted, or it shall return *thrd\_success* due to  
2552 spurious wakeup.[/CX]

2553 The behavior is undefined if the value specified by the *cond* or *mtx* argument to these  
2554 functions does not refer to an initialized condition variable or an initialized mutex object,  
2555 respectively.

#### 2556 **RETURN VALUE**

2557 The *cond\_timedwait()* function shall return *thrd\_success* upon success, or  
2558 *thrd\_timedout* if the time specified in the call was reached without acquiring the  
2559 requested resource, or *thrd\_error* if the request could not be honored.

2560 The *cond\_wait()* function shall return *thrd\_success* upon success or *thrd\_error* if the  
2561 request could not be honored.

#### 2562 **ERRORS**

2563 See RETURN VALUE.

#### 2564 **EXAMPLES**

2565 None.

#### 2566 **APPLICATION USAGE**

2567 None.

#### 2568 **RATIONALE**

2569 These functions are not affected by signal handlers (except as stated in the DESCRIPTION)  
2570 for the reasons stated in [xref to XRAT B.2.3].

#### 2571 **FUTURE DIRECTIONS**

2572 None.

#### 2573 **SEE ALSO**

2574 *cmd\_broadcast*, *cmd\_destroy*, *timespec\_get*

2575 XBD Section 4.12.2, <**threads.h**>

#### 2576 **CHANGE HISTORY**

2577 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

2578 Ref F.10.8.1 para 2

2579 On page 705 line 24155 section *copysign()*, add a new paragraph:

2580 [MX]The returned value shall be exact and shall be independent of the current rounding  
2581 direction mode.[/MX]

2582 Ref G.6.4.1 para 1

2583 On page 711 line 24308 section *cpow()*, add a new paragraph:

2584 [MXC]These functions shall raise floating-point exceptions if appropriate for the calculation  
2585 of the parts of the result, and may also raise spurious floating-point exceptions.[/MXC]

2586 Ref G.6.4.1 footnote 386

2587 On page 711 line 24318 section *cpow()*, change RATIONALE from:

2588 None.

2589 to:

2590 Permitting spurious floating-point exceptions allows *cpow(z, c)* to be implemented as *cexp(c*  
2591 *clog(z))* without precluding implementations that treat special cases more carefully.

2592 Ref G.6 para 7, G.6.2.5

2593 On page 718 line 24545 section *csin()*, add:

2594 [MXC]*csin(conj(iz))*, *csinf(conjf(iz))* and *csinl(conjl(iz))* shall return exactly the same value  
2595 as *conj(csin(iz))*, *conjf(csinf(iz))* and *conjl(csinl(iz))*, respectively, and *csin(-iz)*, *csinf(-iz)*  
2596 and *csinl(-iz)* shall return exactly the same value as *-csin(iz)*, *-csinf(iz)* and *-csinl(iz)*,  
2597 respectively, including for the special values of *iz* below.

2598 If *iz* is  $+0 + i0$ ,  $-i (+0 + i0)$  shall be returned.

2599 If *iz* is  $+0 + i\text{Inf}$ ,  $-i (\pm 0 + i\text{NaN})$  shall be returned and the invalid floating-point exception  
2600 shall be raised; the sign of the imaginary part of the result is unspecified.

2601 If *iz* is  $+0 + i\text{NaN}$ ,  $-i (\pm 0 + i\text{NaN})$  shall be returned; the sign of the imaginary part of the  
2602 result is unspecified.



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

2605 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  
2606 invalid floating-point exception may be raised.

2607 If  $iz$  is  $+\text{Inf} + i0$ ,  $-i(+\text{Inf} + i0)$  shall be returned.

2608 If  $iz$  is  $+\text{Inf} + iy$  where  $y$  is positive and finite,  $-i\text{Inf}(\cos(y) + i\sin(y))$  shall be returned.

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

2611 If  $iz$  is  $+\text{Inf} + i\text{NaN}$ ,  $-i(\pm\text{Inf} + i\text{NaN})$  shall be returned; the sign of the imaginary part of the  
2612 result is unspecified.

2613 If  $iz$  is  $\text{NaN} + i0$ ,  $-i(\text{NaN} + i0)$  shall be returned.

2614 If  $iz$  is  $\text{NaN} + iy$  where  $y$  is any non-zero number,  $-i(\text{NaN} + i\text{NaN})$  shall be returned and the  
2615 invalid floating-point exception may be raised.

2616 If  $iz$  is  $\text{NaN} + i\text{NaN}$ ,  $-i(\text{NaN} + i\text{NaN})$  shall be returned.[/MXC]

2617 Ref G.6 para 7  
2618 On page 718 line 24553 section `csin()`, change RATIONALE from:

2619 None.

2620 to:

2621 The MXC special cases for `csin()` are derived from those for `csinh()` by applying the formula  
2622  $csin(z) = -i csinh(iz)$ .

2623 Ref G.6.2.5  
2624 On page 719 line 24574 section `csinh()`, add:

2625 [MXC]`csinh(conj(z))`, `csinhf(conj(z))` and `csinhl(conj(z))` shall return exactly the same  
2626 value as `conj(csinh(z))`, `conjf(csinhf(z))` and `conjl(csinhl(z))`, respectively, and `csinh(-z)`,  
2627 `csinhf(-z)` and `csinhl(-z)` shall return exactly the same value as  $-csinh(z)$ ,  $-csinhf(z)$  and  
2628  $-csinhl(z)$ , respectively, including for the special values of  $z$  below.

2629 If  $z$  is  $+0 + i0$ ,  $+0 + i0$  shall be returned.

2630 If  $z$  is  $+0 + i\text{Inf}$ ,  $\pm 0 + i\text{NaN}$  shall be returned and the invalid floating-point exception shall be  
2631 raised; the sign of the real part of the result is unspecified.

2632 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  
2633 unspecified.

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

- 2636 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  
2637 floating-point exception may be raised.
- 2638 If  $z$  is  $+\text{Inf} + i0$ ,  $+\text{Inf} + i0$  shall be returned.
- 2639 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive and finite,  $+\text{Inf} (\cos(y) + i\sin(y))$  shall be returned.
- 2640 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $\pm\text{Inf} + i\text{NaN}$  shall be returned and the invalid floating-point exception  
2641 shall be raised; the sign of the real part of the result is unspecified.
- 2642 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  
2643 unspecified.
- 2644 If  $z$  is  $\text{NaN} + i0$ ,  $\text{NaN} + i0$  shall be returned.
- 2645 If  $z$  is  $\text{NaN} + iy$  where  $y$  is any non-zero number,  $\text{NaN} + i\text{NaN}$  shall be returned and the  
2646 invalid floating-point exception may be raised.
- 2647 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]
- 2648 Ref G.6.4.2  
2649 On page 721 line 24612 section `csqrt()`, add:
- 2650 [MXC]`csqrt(conj(z))`, `csqrtf(conjf(z))` and `csqrtl(conjl(z))` shall return exactly the same value  
2651 as `conj(csqrt(z))`, `conjf(csqrtf(z))` and `conjl(csqrtl(z))`, respectively, including for the special  
2652 values of  $z$  below.
- 2653 If  $z$  is  $\pm 0 + i0$ ,  $+0 + i0$  shall be returned.
- 2654 If the imaginary part of  $z$  is  $\text{Inf}$ ,  $+\text{Inf} + i\text{Inf}$ , shall be returned.
- 2655 If  $z$  is  $x + i\text{NaN}$  where  $x$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2656 point exception may be raised.
- 2657 If  $z$  is  $-\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+0 + i\text{Inf}$  shall be returned.
- 2658 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $+\text{Inf} + i0$  shall be returned.
- 2659 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  
2660 is unspecified.
- 2661 If  $z$  is  $+\text{Inf} + i\text{NaN}$ ,  $+\text{Inf} + i\text{NaN}$  shall be returned.
- 2662 If  $z$  is  $\text{NaN} + iy$  where  $y$  is finite,  $\text{NaN} + i\text{NaN}$  shall be returned and the invalid floating-  
2663 point exception may be raised.
- 2664 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]
- 2665 Ref G.6 para 7, G.6.2.6  
2666 On page 722 line 24641 section `ctan()`, add:

2667 [MXC] $ctan(conj(iz))$ ,  $ctanf(conjf(iz))$  and  $ctanl(conjl(iz))$  shall return exactly the same value  
2668 as  $conj(ctan(iz))$ ,  $conjf(ctanf(iz))$  and  $conjl(ctanl(iz))$ , respectively, and  $ctan(-iz)$ ,  $ctanf(-iz)$   
2669 and  $ctanl(-iz)$  shall return exactly the same value as  $-ctan(iz)$ ,  $-ctanf(iz)$  and  $-ctanl(iz)$ ,  
2670 respectively, including for the special values of  $iz$  below.

2671 If  $iz$  is  $+0 + i0$ ,  $-i (+0 + i0)$  shall be returned.

2672 If  $iz$  is  $0 + iInf$ ,  $-i (0 + iNaN)$  shall be returned and the invalid floating-point exception shall  
2673 be raised.

2674 If  $iz$  is  $x + iInf$  where  $x$  is non-zero and finite,  $-i (NaN + iNaN)$  shall be returned and the  
2675 invalid floating-point exception shall be raised.

2676 If  $iz$  is  $0 + iNaN$ ,  $-i (0 + iNaN)$  shall be returned.

2677 If  $iz$  is  $x + iNaN$  where  $x$  is non-zero and finite,  $-i (NaN + iNaN)$  shall be returned and the  
2678 invalid floating-point exception may be raised.

2679 If  $iz$  is  $+Inf + iy$  where  $y$  is positive-signed and finite,  $-i (1 + i0 \sin(2y))$  shall be returned.

2680 If  $iz$  is  $+Inf + iInf$ ,  $-i (1 \pm i0)$  shall be returned; the sign of the real part of the result is  
2681 unspecified.

2682 If  $iz$  is  $+Inf + iNaN$ ,  $-i (1 \pm i0)$  shall be returned; the sign of the real part of the result is  
2683 unspecified.

2684 If  $iz$  is  $NaN + i0$ ,  $-i (NaN + i0)$  shall be returned.

2685 If  $iz$  is  $NaN + iy$  where  $y$  is any non-zero number,  $-i (NaN + iNaN)$  shall be returned and the  
2686 invalid floating-point exception may be raised.

2687 If  $iz$  is  $NaN + iNaN$ ,  $-i (NaN + iNaN)$  shall be returned.[/MXC]

2688 Ref G.6 para 7  
2689 On page 722 line 24649 section  $ctan()$ , change RATIONALE from:

2690 None.

2691 to:

2692 The MXC special cases for  $ctan()$  are derived from those for  $ctanh()$  by applying the  
2693 formula  $ctan(z) = -i ctanh(iz)$ .

2694 Ref G.6.2.6  
2695 On page 723 line 24670 section  $ctanh()$ , add:

2696 [MXC] $ctanh(conj(z))$ ,  $ctanhf(conjf(z))$  and  $ctanhl(conjl(z))$  shall return exactly the same  
2697 value as  $conj(ctanh(z))$ ,  $conjf(ctanhf(z))$  and  $conjl(ctanhl(z))$ , respectively, and  $ctanh(-z)$ ,  
2698  $ctanhf(-z)$  and  $ctanhl(-z)$  shall return exactly the same value as  $-ctanh(z)$ ,  $-ctanhf(z)$  and  
2699  $-ctanhl(z)$ , respectively, including for the special values of  $z$  below.

2700 If  $z$  is  $+0 + i0$ ,  $+0 + i0$  shall be returned.

2701 If  $z$  is  $0 + i\text{Inf}$ ,  $0 + i\text{NaN}$  shall be returned and the invalid floating-point exception shall be  
2702 raised.

2703 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  
2704 floating-point exception shall be raised.

2705 If  $z$  is  $0 + i\text{NaN}$ ,  $0 + i\text{NaN}$  shall be returned.

2706 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  
2707 floating-point exception may be raised.

2708 If  $z$  is  $+\text{Inf} + iy$  where  $y$  is positive-signed and finite,  $1 + i0 \sin(2y)$  shall be returned.

2709 If  $z$  is  $+\text{Inf} + i\text{Inf}$ ,  $1 \pm i0$  shall be returned; the sign of the imaginary part of the result is  
2710 unspecified.

2711 If  $z$  is  $+\text{Inf} + i\text{NaN}$ ,  $1 \pm i0$  shall be returned; the sign of the imaginary part of the result is  
2712 unspecified.

2713 If  $z$  is  $\text{NaN} + i0$ ,  $\text{NaN} + i0$  shall be returned.

2714 If  $z$  is  $\text{NaN} + iy$  where  $y$  is any non-zero number,  $\text{NaN} + i\text{NaN}$  shall be returned and the  
2715 invalid floating-point exception may be raised.

2716 If  $z$  is  $\text{NaN} + i\text{NaN}$ ,  $\text{NaN} + i\text{NaN}$  shall be returned.[/MXC]

2717 Ref 7.27.3, 7.1.4 para 5  
2718 On page 727 line 24774 section `ctime()`, change:

2719 [CX]The `ctime()` function need not be thread-safe.[/CX]

2720 to:  
2721 The `ctime()` function need not be thread-safe; however, `ctime()` shall avoid data races with all  
2722 functions other than itself, `asctime()`, `gmtime()` and `localtime()`.

2723 Ref 7.5 para 2  
2724 On page 781 line 26447 section `errno`, change:

2725 The lvalue `errno` is used by many functions to return error values.

2726 to:  
2727 The lvalue to which the macro `errno` expands is used by many functions to return error  
2728 values.

2729 Ref 7.5 para 3  
2730 On page 781 line 26449 section `errno`, change:

2731 The value of `errno` shall be defined only after a call to a function for which it is explicitly  
2732 stated to be set and until it is changed by the next function call or if the application assigns it  
2733 a value.

2734 to:

2735 The value of *errno* in the initial thread shall be zero at program startup (the initial value of  
2736 *errno* in other threads is an indeterminate value) and shall otherwise be defined only after a  
2737 call to a function for which it is explicitly stated to be set and until it is changed by the next  
2738 function call or if the application assigns it a value.

2739 Ref 7.5 para 2  
2740 On page 781 line 26456 section *errno*, delete:

2741 It is unspecified whether *errno* is a macro or an identifier declared with external linkage.

2742 Ref 7.22.4.4 para 2  
2743 On page 796 line 27057 section *exit()*, add a new (unshaded) paragraph:

2744 The *exit()* function shall cause normal process termination to occur. No functions registered  
2745 by the *at\_quick\_exit()* function shall be called. If a process calls the *exit()* function more  
2746 than once, or calls the *quick\_exit()* function in addition to the *exit()* function, the behavior is  
2747 undefined.

2748 Ref 7.22.4.4 para 2  
2749 On page 796 line 27068 section *exit()*, delete:

2750 If *exit()* is called more than once, the behavior is undefined.

2751 Ref 7.22.4.3, 7.22.4.7  
2752 On page 796 line 27086 section *exit()*, add *at\_quick\_exit* and *quick\_exit* to the SEE ALSO section.

2753 Ref F.10.4.2 para 2  
2754 On page 804 line 27323 section *fabs()*, add a new paragraph:

2755 [MX]The returned value shall be exact and shall be independent of the current rounding  
2756 direction mode.[/MX]

2757 Ref 7.21.2 para 7,8  
2758 On page 874 line 29483 section *flockfile()*, change:

2759 These functions shall provide for explicit application-level locking of stdio (**FILE \***)  
2760 objects.

2761 to:

2762 These functions shall provide for explicit application-level locking of the locks associated  
2763 with standard I/O streams (see [xref to 2.5]).

2764 Ref 7.21.2 para 7,8  
2765 On page 874 line 29499 section *flockfile()*, delete:

2766 All functions that reference (**FILE \***) objects, except those with names ending in *\_unlocked*,  
2767 shall behave as if they use *flockfile()* and *funlockfile()* internally to obtain ownership of these  
2768 (**FILE \***) objects.

2769 Ref F.10.6.2 para 3  
2770 On page 876 line 29560 section floor(), add a new paragraph:

2771 [MX]These functions may raise the inexact floating-point exception for finite non-integer  
2772 arguments.[/MX]

2773 Ref F.10.6.2 para 2  
2774 On page 876 line 29562 section floor(), change:

2775 [MX]The result shall have the same sign as  $x$ .[/MX]

2776 to:

2777 [MX]The returned value shall be independent of the current rounding direction mode and  
2778 shall have the same sign as  $x$ .[/MX]

2779 Ref F.10.6.2 para 3  
2780 On page 876 line 29576 section floor(), delete from APPLICATION USAGE:

2781 These functions may raise the inexact floating-point exception if the result differs in value  
2782 from the argument.

2783 Ref F.10.9.2 para 2  
2784 On page 880 line 29695 section fmax(), add a new paragraph:

2785 [MX]The returned value shall be exact and shall be independent of the current rounding  
2786 direction mode.[/MX]

2787 Ref F.10.9.3 para 2  
2788 On page 884 line 29844 section fmin(), add a new paragraph:

2789 [MX]The returned value shall be exact and shall be independent of the current rounding  
2790 direction mode.[/MX]

2791 Ref F.10.7.1 para 2  
2792 On page 885 line 29892 section fmod(), change:

2793 [MXX]If the correct value would cause underflow, and is representable, a range error may  
2794 occur and the correct value shall be returned.[/MXX]

2795 to:

2796 [MX]When subnormal results are supported, the returned value shall be exact and shall be  
2797 independent of the current rounding direction mode.[/MX]

2798 Ref 7.21.5.3 para 5  
2799 On page 892 line 30117 section fopen(), change:

2800 [CX]The functionality described on this reference page is aligned with the ISO C standard.  
2801 Any conflict between the requirements described here and the ISO C standard is  
2802 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2803 to:

2804 [CX]Except for the “exclusive access” requirement (see below), the functionality described  
2805 on this reference page is aligned with the ISO C standard. Any other conflict between the  
2806 requirements described here and the ISO C standard is unintentional. This volume of  
2807 POSIX.1-202x defers to the ISO C standard for all *fopen()* functionality except in relation to  
2808 “exclusive access”.[/CX]

2809 Ref 7.21.5.3 para 5

2810 On page 892 line 30132 section *fopen()*, after applying bug 411, change:

2811 'x' If specified with a prefix beginning with 'w' [CX]or 'a'[/CX], then the function shall  
2812 fail if the file already exists, [CX]as if by the O\_EXCL flag to *open()*. If specified  
2813 with a prefix beginning with 'r', this modifier shall have no effect.[/CX]

2814 to:

2815 'x' If specified with a prefix beginning with 'w' [CX]or 'a'[/CX], then the function shall  
2816 fail if the file already exists or cannot be created; if the file does not exist and can be  
2817 created, it shall be created with [CX]an implementation-defined form of[/CX]  
2818 exclusive (also known as non-shared) access, [CX]if supported by the underlying file  
2819 system, provided the resulting file permissions are the same as they would be without  
2820 the 'x' modifier. If specified with a prefix beginning with 'r', this modifier shall have  
2821 no effect.[/CX]

2822 **Note:** The ISO C standard requires exclusive access “to the extent that the underlying file  
2823 system supports exclusive access”, but does not define what it means by this. Taken  
2824 at face value—that systems must do whatever they are capable of, at the file system  
2825 level, in order to exclude access by others—this would require POSIX.1 systems to  
2826 set the file permissions in a way that prevents access by other users and groups.  
2827 Consequently, this volume of POSIX.1-202x does not defer to the ISO C standard as  
2828 regards the “exclusive access” requirement.

2829 [Note to reviewers: This “exclusive access” requirement may be clarified in C2x, in which case the](#)  
2830 [above text may be changed to match the proposed C2x text.](#)

2831 Ref 7.21.5.3 para 3

2832 On page 892 line 30144 section *fopen()*, change:

2833 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and ...

2834 to:

2835 If the first character in *mode* is *w* or *a*, and ...

2836 Ref 7.21.5.3 para 3,5

2837 On page 892 line 30148 section *fopen()*, change:

2838 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously  
2839 exist, the *fopen()* function shall create a file as if it called the *creat()* function with a value  
2840 appropriate for the *path* argument interpreted from *pathname* and a value of S\_IRUSR |

2841 S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH for the *mode* argument.

2842 to:

2843 If the first character in *mode* is *w* or *a*, and the file did not previously exist, the *fopen()*  
2844 function shall create a file as if it called the *open()* function with a value appropriate for the  
2845 *path* argument interpreted from *pathname*, a value for the *oflag* argument as specified below,  
2846 and a value of S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH for  
2847 the third argument.

2848 Ref 7.21.5.3 para 5

2849 On page 893 line 30158 section *fopen()*, change:

2850 The file descriptor ...

2851 to:

2852 If the first character in *mode* is *r*, or the suffix of *mode* does not include *x*, the file descriptor  
2853 ...

2854 Ref (none; see bug 411)

2855 On page 893 line 30160 section *fopen()*, change the first column heading from:

2856 ***fopen()* Mode**

2857 to:

2858 ***fopen()* Mode Without Suffix**

2859 and add the following text after the table:

2860 with the addition of the O\_CLOEXEC flag if the suffix of *mode* includes *e*.

2861 Ref 7.21.5.3 para 5

2862 On page 893 line 30166 section *fopen()*, add the following new paragraphs:

2863 [CX]If the first character in *mode* is *w* or *a*, the suffix of *mode* includes *x*, and the underlying  
2864 file system does not support exclusive access, then the file descriptor associated with the  
2865 opened stream shall be allocated and opened as if by a call to *open()* with the following  
2866 flags:

<b><i>fopen()</i> Mode Without Suffix</b>	<b><i>open()</i> Flags</b>
[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

2867 with the addition of the O\_CLOEXEC flag if the suffix of *mode* includes *e*.

2868 If the first character in *mode* is *w* or *a*, the suffix of *mode* includes *x*, and the underlying file



2869 system supports exclusive access, then the file descriptor associated with the opened stream  
2870 shall be allocated and opened as if by a call to *open()* with the above flags or with the above  
2871 flags ORed with an implementation-defined file creation flag if necessary to enable  
2872 exclusive access (see above).[/CX]

2873 [Note to reviewers: The above change may need to be updated depending on whether WG14 clarify](#)  
2874 [the “exclusive access” requirement.](#)

2875 Ref 7.21.5.3 para 5  
2876 On page 895 line 30236 section *fopen()*, change APPLICATION USAGE from:

2877 None.

2878 to:

2879 If an application needs to create a file in a way that fails if the file already exists, and either  
2880 requires that it does not have exclusive access to the file or does not need exclusive access, it  
2881 should use *open()* with the *O\_CREAT* and *O\_EXCL* flags instead of using *fopen()* with an *x*  
2882 in the *mode*. A stream can then be created, if needed, by calling *fdopen()* on the file  
2883 descriptor returned by *open()*.

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

2886 Ref 7.21.5.3 para 5  
2887 On page 895 line 30238 section *fopen()*, after applying bug 411, change:

2888 The *x* mode suffix character was added by C1x only for files opened with a mode string  
2889 beginning with *w*.

2890 to:

2891 The *x* mode suffix character is specified by the ISO C standard only for files opened with a  
2892 mode string beginning with *w*.

2893 and then add two new paragraphs after the one that starts with the above text:

2894 When the last character in *mode* is *x*, the ISO C standard requires that the file is created with  
2895 exclusive access to the extent that the underlying system supports exclusive access.  
2896 Although POSIX.1 does not specify any method of enabling exclusive access, it allows for  
2897 the existence of an implementation-defined file creation flag that enables it. Note that it must  
2898 be a file creation flag, not a file access mode flag (that is, one that is included in  
2899 *O\_ACCMODE*) or a file status flag, so that it does not affect the value returned by *fcntl()*  
2900 with *F\_GETFL*. On implementations that have such a flag, if support for it is file system  
2901 dependent and exclusive access is requested when using *fopen()* to create a file on a file  
2902 system that does not support it, the flag must not be used if it would cause *fopen()* to fail.

2903 Some implementations support mandatory file locking as a means of enabling exclusive  
2904 access to a file. Locks are set in the normal way, but instead of only preventing others from  
2905 setting conflicting locks they prevent others from accessing the contents of the locked part  
2906 of the file in a way that conflicts with the lock. However, unless the implementation has a  
2907 way of setting a whole-file write lock on file creation, this does not satisfy the requirement

2908 in the ISO C standard that the file is “created with exclusive access to the extent that the  
2909 underlying system supports exclusive access”. (Having *fopen()* create the file and set a lock  
2910 on the file as two separate operations is not the same, and it would introduce a race  
2911 condition whereby another process could open the file and write to it (or set a lock) in  
2912 between the two operations.) However, on all implementations that support mandatory file  
2913 locking, its use is discouraged; therefore, it is recommended that implementations which  
2914 support mandatory file locking do **not** add a means of creating a file with a whole-file  
2915 exclusive lock set, so that *fopen()* is not required to enable mandatory file locking in order to  
2916 conform to the ISO C standard. Note also that, since mandatory file locking is enabled via a  
2917 file permissions change, the requirement that the 'x' modifier does not alter the permissions  
2918 means that this standard does not allow mandatory file locking to be enabled. An  
2919 implementation that has a means of creating a file with a whole-file exclusive lock set would  
2920 need to provide a way to change the behavior of *fopen()* depending on whether the calling  
2921 process is executing in a POSIX.1 conforming environment or an ISO C conforming  
2922 environment.

2923 [Note to reviewers: The above change may need to be updated depending on whether WG14 clarify](#)  
2924 [the “exclusive access” requirement.](#)

2925 Ref 7.22.3.3 para 2  
2926 | On page 933 line 31673 section *free()*, [after applying bug 1218](#) change:

2927 Otherwise, if the argument does not match a pointer earlier returned by a function in  
2928 POSIX.1-2017 that allocates memory as if by *malloc()*, or if the space has been deallocated  
2929 | by a call to *free()*, [realloc\(\)](#), [\[CX\]reallocarray\(\)](#), [\[/CX\]](#) the behavior is undefined.

2930 to:

2931 Otherwise, if the argument does not match a pointer earlier returned by *aligned\_alloc()*,  
2932 | *calloc()*, *malloc()*, [\[ADV\]posix\\_memalign\(\)](#), [\[/ADV\] realloc\(\)](#), [\[CX\]reallocarray\(\)](#), or a  
2933 | function in POSIX.1-20xx that allocates memory as if by *malloc()*, [\[/CX\]](#) or if the space has  
2934 | been deallocated by a call to *free()*, [\[CX\]reallocarray\(\)](#), [\[/CX\]](#) or *realloc()*, the behavior is  
2935 | undefined.

2936 Ref 7.22.3 para 2  
2937 | On page 933 line 31677 section *free()*, add a new paragraph:

2938 For purposes of determining the existence of a data race, *free()* shall behave as though it  
2939 | accessed only memory locations accessible through its argument and not other static  
2940 | duration storage. The function may, however, visibly modify the storage that it deallocates.  
2941 | Calls to *aligned\_alloc()*, *calloc()*, *free()*, *malloc()*, [\[ADV\]posix\\_memalign\(\)](#), [\[/ADV\]](#)  
2942 | [\[CX\]reallocarray\(\)](#), [\[/CX\]](#) and *realloc()* that allocate or deallocate a particular region of  
2943 | memory shall occur in a single total order (see [\[xref to XBD 4.12.1\]](#)), and each such  
2944 | deallocation call shall synchronize with the next allocation (if any) in this order.

2945 Ref 7.22.3.1  
2946 | On page 933 line 31691 section *free()*, add *aligned\_alloc* to the SEE ALSO section.

2947 Ref 7.21.5.3 para 5  
2948 | On page 942 line 31988 section *freopen()*, change:

2949 | [\[CX\]](#)The functionality described on this reference page is aligned with the ISO C standard.

2950 Any conflict between the requirements described here and the ISO C standard is  
2951 unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.[/CX]

2952 to:

2953 [CX]Except for the “exclusive access” requirement (see [xref to fopen()]), the functionality  
2954 described on this reference page is aligned with the ISO C standard. Any other conflict  
2955 between the requirements described here and the ISO C standard is unintentional. This  
2956 volume of POSIX.1-202x defers to the ISO C standard for all *freopen()* functionality except  
2957 in relation to “exclusive access”.[/CX]

2958 Ref 7.21.5.3 para 3,5; 7.21.5.4 para 2

2959 On page 942 line 32010 section *freopen()*, replace the following text:

2960 shall be allocated and opened as if by a call to *open()* with the following flags:

2961 and the table that follows it, and the paragraph added by bug 411 after the table, with:

2962 shall be allocated and opened as if by a call to *open()* with the flags specified for *fopen()*  
2963 with the same *mode* argument.

2964 Ref (none)

2965 On page 944 line 32094 section *freopen()*, change:

2966 It is possible that these side-effects are an unintended consequence of the way the feature is  
2967 specified in the ISO/IEC 9899: 1999 standard, but unless or until the ISO C standard is  
2968 changed, ...

2969 to:

2970 It is possible that these side-effects are an unintended consequence of the way the feature  
2971 was specified in the ISO/IEC 9899: 1999 standard (and still is in the current standard), but  
2972 unless or until the ISO C standard is changed, ...

2973 [Note to reviewers: if the APPLICATION USAGE and RATIONALE additions for fopen\(\) are](#)  
2974 [retained, changes should be added here to make the equivalent sections for freopen\(\) refer to those](#)  
2975 [for fopen\(\)](#).

2976 Ref (none)

2977 On page 944 line 32102 section *freopen()*, after applying bug 411 change:

2978 The *x* mode suffix character was added by C1x only for files opened with a *mode* string  
2979 beginning with *w*.

2980 to:

2981 The *x* mode suffix character is specified by the ISO C standard only for files opened with a  
2982 mode string beginning with *w*.

2983 Ref 7.12.6.4 para 3

2984 On page 947 line 32161 section *frexp()*, change:

2985           The integer exponent shall be stored in the **int** object pointed to by *exp*.

2986 to:

2987           The integer exponent shall be stored in the **int** object pointed to by *exp*; if the integer  
2988 exponent is outside the range of **int**, the results are unspecified.

2989 Ref F.10.3.4 para 3  
2990 On page 947 line 32164 section `frexp()`, add a new paragraph:

2991           [MX]When the radix of the argument is a power of 2, the returned value shall be exact and  
2992 shall be independent of the current rounding direction mode.[/MX]

2993 Ref 7.21.6.2 para 4  
2994 On page 950 line 32239 section `fscanf()`, change:

2995           If a directive fails, as detailed below, the function shall return.

2996 to:

2997           When all directives have been executed, or if a directive fails (as detailed below), the  
2998 function shall return.

2999 Ref 7.21.6.2 para 5  
3000 On page 950 line 32242 section `fscanf()`, after applying bug 1163 change:

3001           A directive composed of one or more white-space bytes shall be executed by reading input  
3002 until no more valid input can be read, or up to the first non-white-space byte , which remains  
3003 unread.

3004 to:

3005           A directive composed of one or more white-space bytes shall be executed by reading input  
3006 up to the first non-white-space byte, which shall remain unread, or until no more bytes can  
3007 be read. The directive shall never fail.

3008 Ref (none)  
3009 On page 955 line 32471 section `fscanf()`, change:

3010           This function is aligned with the ISO/IEC 9899: 1999 standard, and in doing so a few  
3011 “obvious” things were not included. Specifically, the set of characters allowed in a scanset is  
3012 limited to single-byte characters. In other similar places, multi-byte characters have been  
3013 permitted, but for alignment with the ISO/IEC 9899: 1999 standard, it has not been done  
3014 here.

3015 to:

3016           The set of characters allowed in a scanset is limited to single-byte characters. In other  
3017 similar places, multi-byte characters have been permitted, but for alignment with the ISO C  
3018 standard, it has not been done here.

3019 Ref 7.29.2.2 para 4

3020 On page 1004 line 34144 section `fwscanf()`, change:

3021 If a directive fails, as detailed below, the function shall return.

3022 to:

3023 When all directives have been executed, or if a directive fails (as detailed below), the  
3024 function shall return.

3025 Ref 7.29.2.2 para 5

3026 On page 1004 line 34147 section `fwscanf()`, change:

3027 A directive composed of one or more white-space wide characters is executed by reading  
3028 input until no more valid input can be read, or up to the first wide character which is not a  
3029 white-space wide character, which remains unread.

3030 to:

3031 A directive composed of one or more white-space wide characters shall be executed by  
3032 reading input up to the first wide character that is not a white-space wide character, which  
3033 shall remain unread, or until no more wide characters can be read. The directive shall never  
3034 fail.

3035 Ref 7.27.3, 7.1.4 para 5

3036 On page 1113 line 37680 section `gmtime()`, change:

3037 [CX]The `gmtime()` function need not be thread-safe.[/CX]

3038 to:

3039 The `gmtime()` function need not be thread-safe; however, `gmtime()` shall avoid data races  
3040 with all functions other than itself, `asctime()`, `ctime()` and `localtime()`.

3041 Ref F.10.3.5 para 1

3042 On page 1133 line 38281 section `ilogb()`, add a new paragraph:

3043 [MX]When the correct result is representable in the range of the return type, the returned  
3044 value shall be exact and shall be independent of the current rounding direction mode.[/MX]

3045 Ref F.10.3.5 para 3

3046 On page 1133 line 38282,38285,38288 section `ilogb()`, change:

3047 [XSI]On XSI-conformant systems, a domain error shall occur[/XSI]

3048 to:

3049 [XSI|MX]On XSI-conformant systems and on systems that support the IEC 60559 Floating-  
3050 Point option, a domain error shall occur[/XSI|MX]

3051 Ref 7.12.6.5 para 2

3052 On page 1133 line 38291 section `ilogb()`, change:

3053 If the correct value is greater than `{INT_MAX}`, [MX]a domain error shall occur and[/MX]

3054 an unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error  
3055 shall occur and {INT\_MAX} shall be returned.[/XSI]

3056 If the correct value is less than {INT\_MIN}, [MX]a domain error shall occur and[/MX] an  
3057 unspecified value shall be returned. [XSI]On XSI-conformant systems, a domain error shall  
3058 occur and {INT\_MIN} shall be returned.[/XSI]

3059 to:

3060 If the correct value is greater than {INT\_MAX} or less than {INT\_MIN}, an unspecified  
3061 value shall be returned. [XSI]On XSI-conformant systems, a domain error shall occur and  
3062 {INT\_MAX} or {INT\_MIN}, respectively, shall be returned;[/XSI] [MX]if the IEC 60559  
3063 Floating-Point option is supported, a domain error shall occur;[/MX] otherwise, a domain  
3064 error or range error may occur.

3065 Ref F.10.3.5 para 3

3066 On page 1133 line 38300 section `ilogb()`, change:

3067 [XSI]The *x* argument is zero, NaN, or  $\pm\text{Inf}$ .[/XSI]

3068 to:

3069 [XSI|MX]The *x* argument is zero, NaN, or  $\pm\text{Inf}$ .[/XSI|MX]

3070 Ref F.10.11 para 1

3071 On page 1174 line 39604 section `isgreater()`,

3072 and page 1175 line 39642 section `isgreaterequal()`,

3073 and page 1177 line 39708 section `isless()`,

3074 and page 1178 line 39746 section `islessequal()`,

3075 and page 1179 line 39784 section `islessgreater()`, add a new paragraph:

3076 [MX]Relational operators and their corresponding comparison macros shall produce  
3077 equivalent result values, even if argument values are represented in wider formats. Thus,  
3078 comparison macro arguments represented in formats wider than their semantic types shall  
3079 not be converted to the semantic types, unless the wide evaluation method converts operands  
3080 of relational operators to their semantic types. The standard wide evaluation methods  
3081 characterized by `FLT_EVAL_METHOD` equal to 1 or 2 (see [xref to <float.h>]) do not  
3082 convert operands of relational operators to their semantic types.[/MX]

3083 (The editors may wish to merge the pages for the above interfaces to reduce duplication – they have  
3084 duplicate APPLICATION USAGE as well.)

3085 Ref 7.30.2.2.1 para 4

3086 On page 1202 line 40411 section `iswctype()`, remove the CX shading from:

3087 If *charclass* is (`wctype_t`)0, these functions shall return 0.

3088 Ref 7.17.3.1

3089 On page 1229 line 41126 insert a new `kill_dependency()` section:

3090 **NAME**

3091 `kill_dependency` — terminate a dependency chain

3092 **SYNOPSIS**  
3093       #include <stdatomic.h>  
3094       type kill\_dependency(type y);

3095 **DESCRIPTION**  
3096       [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3097       Any conflict between the requirements described here and the ISO C standard is  
3098       unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3099       Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide the  
3100       <stdatomic.h> header nor support this macro.

3101       The `kill_dependency()` macro shall terminate a dependency chain (see [xref to XBD 4.12.1  
3102       Memory Ordering]). The argument shall not carry a dependency to the return value.

3103 **RETURN VALUE**  
3104       The `kill_dependency()` macro shall return the value of *y*.

3105 **ERRORS**  
3106       No errors are defined.

3107 **EXAMPLES**  
3108       None.

3109 **APPLICATION USAGE**  
3110       None.

3111 **RATIONALE**  
3112       None.

3113 **FUTURE DIRECTIONS**  
3114       None.

3115 **SEE ALSO**  
3116       XBD Section 4.12.1, <stdatomic.h>

3117 **CHANGE HISTORY**  
3118       First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3119       Ref 7.12.8.3, 7.1.4 para 5  
3120       On page 1241 line 41433 section `lgamma()`, change:  
3121       [CX]These functions need not be thread-safe.[/CX]

3122       to:

3123       [XSI]If concurrent calls are made to these functions, the value of *signgam* is indeterminate.[/  
3124       XSI]

3125       Ref 7.12.8.3, 7.1.4 para 5

3126 On page 1242 line 41464 section `lgamma()`, add a new paragraph to APPLICATION USAGE:

3127 If the value of *signgam* will be obtained after a call to *lgamma()*, *lgammaf()*, or *lgammal()*,  
3128 in order to ensure that the value will not be altered by another call in a different thread,  
3129 applications should either restrict calls to these functions to be from a single thread or use a  
3130 lock such as a mutex or spin lock to protect a critical section starting before the function call  
3131 and ending after the value of *signgam* has been obtained.

3132 Ref 7.12.8.3, 7.1.4 para 5

3133 On page 1242 line 41466 section `lgamma()`, change RATIONALE from:

3134 None.

3135 to:

3136 Earlier versions of this standard did not require *lgamma()*, *lgammaf()*, and *lgammal()* to be  
3137 thread-safe because *signgam* was a global variable. They are now required to be thread-safe  
3138 to align with the ISO C standard (which, since the introduction of threads in 2011, requires  
3139 that they avoid data races), with the exception that they need not avoid data races when  
3140 storing a value in the *signgam* variable. Since *signgam* is not specified by the ISO C  
3141 standard, this exception is not a conflict with that standard.

3142 Ref 7.11.2.1, 7.1.4 para 5

3143 On page 1262 line 42124 section `localeconv()`, change:

3144 [CX]The *localeconv()* function need not be thread-safe.[/CX]

3145 to:

3146 The *localeconv()* function need not be thread-safe; however, *localeconv()* shall avoid data  
3147 races with all other functions.

3148 Ref 7.27.3, 7.1.4 para 5

3149 On page 1265 line 42217 section `localtime()`, change:

3150 [CX]The *localtime()* function need not be thread-safe.[/CX]

3151 to:

3152 The *localtime()* function need not be thread-safe; however, *localtime()* shall avoid data races  
3153 with all functions other than itself, *asctime()*, *ctime()* and *gmtime()*.

3154 Ref F.10.3.11 para 2

3155 On page 1280 line 42723 section `logb()`, add a new paragraph:

3156 [MX]The returned value shall be exact and shall be independent of the current rounding  
3157 direction mode.[/MX]

3158 Ref 7.13.2.1 para 1

3159 On page 1283 line 42780 section `longjmp()`, change:

3160 `void longjmp(jmp_buf env, int val);`



3161 to:

3162 `_Noreturn void longjmp(jmp_buf env, int val);`

3163 Ref 7.13.2.1 para 2

3164 On page 1283 line 42804 section `longjmp()`, remove the CX shading from:

3165 The effect of a call to `longjmp()` where initialization of the **jmp\_buf** structure was not  
3166 performed in the calling thread is undefined.

3167 Ref 7.13.2.1 para 4

3168 On page 1283 line 42807 section `longjmp()`, change:

3169 After `longjmp()` is completed, program execution continues ...

3170 to:

3171 After `longjmp()` is completed, thread execution shall continue ...

3172 Ref 7.22.3 para 1

3173 On page 1295 line 43144 section `malloc()`, change:

3174 a pointer to any type of object

3175 to:

3176 a pointer to any type of object with a fundamental alignment requirement

3177 ~~Ref 7.22.3 para 1~~

3178 ~~On page 1295 line 43148 section `malloc()`, change:~~

3179 ~~either a null pointer shall be returned, or ...~~

3180 ~~to:~~

3181 ~~either a null pointer shall be returned to indicate an error, or ...~~

3182 Ref 7.22.3 para 2

3183 On page 1295 line 43150 section `malloc()`, add a new paragraph:

3184 For purposes of determining the existence of a data race, `malloc()` shall behave as though it  
3185 accessed only memory locations accessible through its argument and not other static  
3186 duration storage. The function may, however, visibly modify the storage that it allocates.  
3187 Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `[ADV]posix_memalign()`, `[/ADV]`  
3188 `[CX]reallocarray()`, `[/CX]` and `realloc()` that allocate or deallocate a particular region of  
3189 memory shall occur in a single total order (see [xref to XBD 4.12.1]), and each such  
3190 deallocation call shall synchronize with the next allocation (if any) in this order.

3191 Ref 7.22.3.1

3192 On page 1295 line 43171 section `malloc()`, add `aligned_alloc` to the SEE ALSO section.

3193 Ref 7.22.7.1 para 2

3194 On page 1297 line 43194 section `mblen()`, change:

3195 `mbtowc((wchar_t *)0, s, n);`

3196 to:

3197 `mbtowc((wchar_t *)0, (const char *)0, 0);`

3198 `mbtowc((wchar_t *)0, s, n);`

3199 Ref 7.22.7 para 1

3200 On page 1297 line 43198 section `mblen()`, change:

3201 this function shall be placed into its initial state by a call for which

3202 to:

3203 this function shall be placed into its initial state at program startup and can be returned to  
3204 that state by a call for which

3205 Ref 7.22.7 para 1, 7.1.4 para 5

3206 On page 1297 line 43206 section `mblen()`, change:

3207 ~~[CX]The `mblen()` function need not be thread-safe.[/CX]~~

3208 to:

3209 The `mblen()` function need not be thread-safe; however, it shall avoid data races with all  
3210 other functions.

3211 Ref 7.29.6.3 para 1, 7.1.4 para 5

3212 On page 1299 line 43254 section `mbrlen()`, change:

3213 ~~[CX]The `mbrlen()` function need not be thread-safe if called with a NULL `ps`  
3214 argument.[/CX]~~

3215 to:

3216 If called with a null `ps` argument, the `mbrlen()` function need not be thread-safe; however,  
3217 such calls shall avoid data races with calls to `mbrlen()` with a non-null argument and with  
3218 calls to all other functions.

3219 Ref 7.28.1, 7.1.4 para 5

3220 On page 1301 line 43296 insert a new `mbrtoc16()` section:

3221 **NAME**

3222 `mbrtoc16`, `mbrtoc32` — convert a character to a Unicode character code (restartable)

3223 **SYNOPSIS**

3224 `#include <uchar.h>`

3225 `size_t mbrtoc16(char16_t *restrict pc16, const char *restrict s,`  
3226 `size_t n, mbstate_t *restrict ps);`

3227 `size_t mbrtoc32(char32_t *restrict pc32, const char *restrict s,`

3228 `size_t n, mbstate_t *restrict ps);`

3229 **DESCRIPTION**

3230 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3231 Any conflict between the requirements described here and the ISO C standard is  
3232 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3233 If *s* is a null pointer, the *mbrtoc16()* function shall be equivalent to the call:

3234 `mbrtoc16(NULL, "", 1, ps)`

3235 In this case, the values of the parameters *pc16* and *n* are ignored.

3236 If *s* is not a null pointer, the *mbrtoc16()* function shall inspect at most *n* bytes beginning with  
3237 the byte pointed to by *s* to determine the number of bytes needed to complete the next  
3238 character (including any shift sequences). If the function determines that the next character  
3239 is complete and valid, it shall determine the values of the corresponding wide characters and  
3240 then, if *pc16* is not a null pointer, shall store the value of the first (or only) such character in  
3241 the object pointed to by *pc16*. Subsequent calls shall store successive wide characters  
3242 without consuming any additional input until all the characters have been stored. If the  
3243 corresponding wide character is the null wide character, the resulting state described shall be  
3244 the initial conversion state.

3245 If *ps* is a null pointer, the *mbrtoc16()* function shall use its own internal **mbstate\_t** object,  
3246 which shall be initialized at program start-up to the initial conversion state. Otherwise, the  
3247 **mbstate\_t** object pointed to by *ps* shall be used to completely describe the current  
3248 conversion state of the associated character sequence.

3249 The behavior of this function is affected by the *LC\_CTYPE* category of the current locale.

3250 The *mbrtoc16()* function shall not change the setting of *errno* if successful.

3251 The *mbrtoc32()* function shall behave the same way as *mbrtoc16()* except that the first  
3252 parameter shall point to an object of type **char32\_t** instead of **char16\_t**. References to *pc16*  
3253 in the above description shall apply as if they were *pc32* when they are being read as  
3254 describing *mbrtoc32()*.

3255 If called with a null *ps* argument, the *mbrtoc16()* function need not be thread-safe; however,  
3256 such calls shall avoid data races with calls to *mbrtoc16()* with a non-null argument and with  
3257 calls to all other functions.

3258 If called with a null *ps* argument, the *mbrtoc32()* function need not be thread-safe; however,  
3259 such calls shall avoid data races with calls to *mbrtoc32()* with a non-null argument and with  
3260 calls to all other functions.

3261 The implementation shall behave as if no function defined in this volume of POSIX.1-20xx  
3262 calls *mbrtoc16()* or *mbrtoc32()* with a null pointer for *ps*.

3263 **RETURN VALUE**

3264 These functions shall return the first of the following that applies:

3265 0 If the next *n* or fewer bytes complete the character that corresponds to the null  
3266 wide character (which is the value stored).

3267 between 1 and *n* inclusive

3268 If the next *n* or fewer bytes complete a valid character (which is the value  
3269 stored); the value returned shall be the number of bytes that complete the  
3270 character.

3271 (size\_t)-3 If the next character resulting from a previous call has been stored, in which  
3272 case no bytes from the input shall be consumed by the call.

3273 (size\_t)-2 If the next *n* bytes contribute to an incomplete but potentially valid character,  
3274 and all *n* bytes have been processed (no value is stored). When *n* has at least  
3275 the value of the {MB\_CUR\_MAX} macro, this case can only occur if *s*  
3276 points at a sequence of redundant shift sequences (for implementations with  
3277 state-dependent encodings).

3278 (size\_t)-1 If an encoding error occurs, in which case the next *n* or fewer bytes do not  
3279 contribute to a complete and valid character (no value is stored). In this case,  
3280 [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

### 3281 ERRORS

3282 These function shall fail if:

3283 [EILSEQ] An invalid character sequence is detected. [CX]In the POSIX locale  
3284 an [EILSEQ] error cannot occur since all byte values are valid  
3285 characters.[/CX]

3286 These functions may fail if:

3287 [CX][EINVAL] *ps* points to an object that contains an invalid conversion state.[/CX]

### 3288 EXAMPLES

3289 None.

### 3290 APPLICATION USAGE

3291 None.

### 3292 RATIONALE

3293 None.

### 3294 FUTURE DIRECTIONS

3295 None.

### 3296 SEE ALSO

3297 *c16rtomb*

3298 XBD <uchar.h>

### 3299 CHANGE HISTORY

3300 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3301 Ref 7.29.6.3 para 1, 7.1.4 para 5

3302 On page 1301 line 43322 section *mbrtowc()*, change:

3303 [CX]The *mbrtowc()* function need not be thread-safe if called with a NULL *ps*  
3304 argument.[/CX]

3305 to:

3306 If called with a null *ps* argument, the *mbrtowc()* function need not be thread-safe; however,  
3307 such calls shall avoid data races with calls to *mbrtowc()* with a non-null argument and with  
3308 calls to all other functions.

3309 Ref 7.29.6.4 para 1, 7.1.4 para 5

3310 On page 1304 line 43451 section *mbsrtowcs()*, change:

3311 [CX]The *mbsnrtowcs()* and *mbsrtowcs()* functions need not be thread-safe if called with a  
3312 NULL *ps* argument.[/CX]

3313 to:

3314 [CX]If called with a null *ps* argument, the *mbsnrtowcs()* function need not be thread-safe;  
3315 however, such calls shall avoid data races with calls to *mbsnrtowcs()* with a non-null  
3316 argument and with calls to all other functions.[/CX]

3317 If called with a null *ps* argument, the *mbsrtowcs()* function need not be thread-safe;  
3318 however, such calls shall avoid data races with calls to *mbsrtowcs()* with a non-null  
3319 argument and with calls to all other functions.

3320 Ref 7.22.7 para 1

3321 On page 1308 line 43557 section *mbtowc()*, change:

3322 this function is placed into its initial state by a call for which

3323 to:

3324 this function shall be placed into its initial state at program startup and can be returned to  
3325 that state by a call for which

3326 Ref 7.22.7 para 1, 7.1.4 para 5

3327 On page 1308 line 43567 section *mbtowc()*, change:

3328 [CX]The *mbtowc()* function need not be thread-safe.[/CX]

3329 to:

3330 The *mbtowc()* function need not be thread-safe; however, it shall avoid data races with all  
3331 other functions.

3332 Ref 7.24.5.1 para 2

3333 On page 1311 line 43642 section *memchr()*, change:

3334 Implementations shall behave as if they read the memory byte by byte from the beginning of  
3335 the bytes pointed to by *s* and stop at the first occurrence of *c* (if it is found in the initial *n*  
3336 bytes).

3337 to:

3338 The implementation shall behave as if it reads the bytes sequentially and stops as soon as a  
3339 matching byte is found.

3340 Ref F.10.3.12 para 2

3341 On page 1346 line 44854 section `modf()`, add a new paragraph:

3342 [MX]The returned value shall be exact and shall be independent of the current rounding  
3343 direction mode.[/MX]

3344 Ref 7.26.4

3345 On page 1384 line 46032 insert the following new `mtx_*`() sections:

3346 **NAME**

3347 `mtx_destroy`, `mtx_init` — destroy and initialize a mutex

3348 **SYNOPSIS**

3349 `#include <threads.h>`

3350 `void mtx_destroy(mtx_t *mtx);`  
3351 `int mtx_init(mtx_t *mtx, int type);`

3352 **DESCRIPTION**

3353 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3354 Any conflict between the requirements described here and the ISO C standard is  
3355 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3356 The `mtx_destroy()` function shall release any resources used by the mutex pointed to by `mtx`.  
3357 A destroyed mutex object can be reinitialized using `mtx_init()`; the results of otherwise  
3358 referencing the object after it has been destroyed are undefined. It shall be safe to destroy an  
3359 initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that  
3360 another thread is attempting to lock, or a mutex that is being used in a `cond_timedwait()` or  
3361 `cond_wait()` call by another thread, results in undefined behavior. The behavior is undefined if  
3362 the value specified by the `mtx` argument to `mtx_destroy()` does not refer to an initialized  
3363 mutex.

3364 The `mtx_init()` function shall initialize a mutex object with properties indicated by `type`,  
3365 whose valid values include:

3366 `mtx_plain` for a simple non-recursive mutex,

3367 `mtx_timed` for a non-recursive mutex that supports timeout,

3368 `mtx_plain | mtx_recursive` for a simple recursive mutex, or

3369 `mtx_timed | mtx_recursive` for a recursive mutex that supports timeout.

3370 If the `mtx_init()` function succeeds, it shall set the mutex pointed to by `mtx` to a value that  
3371 uniquely identifies the newly initialized mutex. Upon successful initialization, the state of  
3372 the mutex becomes initialized and unlocked. Attempting to initialize an already initialized  
3373 mutex results in undefined behavior.

3374 [CX]See [xref to XSH 2.9.9 Synchronization Object Copies and Alternative Mappings] for  
3375 further requirements.

3376 These functions shall not be affected if the calling thread executes a signal handler during  
3377 the call.[/CX]

#### 3378 **RETURN VALUE**

3379 The *mtx\_destroy()* function shall not return a value.

3380 The *mtx\_init()* function shall return *thrd\_success* on success or *thrd\_error* if the  
3381 request could not be honored.

#### 3382 **ERRORS**

3383 No errors are defined.

#### 3384 **EXAMPLES**

3385 None.

#### 3386 **APPLICATION USAGE**

3387 A mutex can be destroyed immediately after it is unlocked. However, since attempting to  
3388 destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that  
3389 is being used in a *cond\_timedwait()* or *cond\_wait()* call by another thread results in undefined  
3390 behavior, care must be taken to ensure that no other thread may be referencing the mutex.

#### 3391 **RATIONALE**

3392 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT  
3393 B.2.3].

#### 3394 **FUTURE DIRECTIONS**

3395 None.

#### 3396 **SEE ALSO**

3397 *mtx\_lock*

3398 XBD <**threads.h**>

#### 3399 **CHANGE HISTORY**

3400 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

#### 3401 **NAME**

3402 *mtx\_lock*, *mtx\_timedlock*, *mtx\_trylock*, *mtx\_unlock* — lock and unlock a mutex

#### 3403 **SYNOPSIS**

3404 `#include <threads.h>`

```
3405 int mtx_lock(mtx_t *mtx);
3406 int mtx_timedlock(mtx_t * restrict mtx,
3407                  const struct timespec * restrict ts);
3408 int mtx_trylock(mtx_t *mtx);
3409 int mtx_unlock(mtx_t *mtx);
```

3410 **DESCRIPTION**

3411 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3412 Any conflict between the requirements described here and the ISO C standard is  
3413 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3414 The *mtx\_lock()* function shall block until it locks the mutex pointed to by *mtx*. If the mutex  
3415 is non-recursive, the application shall ensure that it is not already locked by the calling  
3416 thread.

3417 The *mtx\_timedlock()* function shall block until it locks the mutex pointed to by *mtx* or until  
3418 after the *TIME\_UTC* -based calendar time pointed to by *ts*. The application shall ensure that  
3419 the specified mutex supports timeout. [CX]Under no circumstance shall the function fail  
3420 with a timeout if the mutex can be locked immediately. The validity of the *ts* parameter need  
3421 not be checked if the mutex can be locked immediately.[/CX]

3422 The *mtx\_trylock()* function shall endeavor to lock the mutex pointed to by *mtx*. If the mutex  
3423 is already locked (by any thread, including the current thread), the function shall return  
3424 without blocking. If the mutex is recursive and the mutex is currently owned by the calling  
3425 thread, the mutex lock count (see below) shall be incremented by one and the *mtx\_trylock()*  
3426 function shall immediately return success.

3427 [CX]These functions shall not be affected if the calling thread executes a signal handler  
3428 during the call; if a signal is delivered to a thread waiting for a mutex, upon return from the  
3429 signal handler the thread shall resume waiting for the mutex as if it was not  
3430 interrupted.[/CX]

3431 If a call to *mtx\_lock()*, *mtx\_timedlock()* or *mtx\_trylock()* locks the mutex, prior calls to  
3432 *mtx\_unlock()* on the same mutex shall synchronize with this lock operation.

3433 The *mtx\_unlock()* function shall unlock the mutex pointed to by *mtx* . The application shall  
3434 ensure that the mutex pointed to by *mtx* is locked by the calling thread. [CX]If there are  
3435 threads blocked on the mutex object referenced by *mtx* when *mtx\_unlock()* is called,  
3436 resulting in the mutex becoming available, the scheduling policy shall determine which  
3437 thread shall acquire the mutex.[/CX]

3438 A recursive mutex shall maintain the concept of a lock count. When a thread successfully  
3439 acquires a mutex for the first time, the lock count shall be set to one. Every time a thread  
3440 relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks  
3441 the mutex, the lock count shall be decremented by one. When the lock count reaches zero,  
3442 the mutex shall become available for other threads to acquire.

3443 For purposes of determining the existence of a data race, mutex lock and unlock operations  
3444 on mutexes of type **mtx\_t** behave as atomic operations. All lock and unlock operations on a  
3445 particular mutex occur in some particular total order.

3446 If *mtx* does not refer to an initialized mutex object, the behavior of these functions is  
3447 undefined.

3448 **RETURN VALUE**

3449 The *mtx\_lock()* and *mtx\_unlock()* functions shall return *thrd\_success* on success, or  
3450 *thrd\_error* if the request could not be honored.



3451 The *mtx\_timedlock()* function shall return *thrd\_success* on success, or *thrd\_timedout*  
3452 if the time specified was reached without acquiring the requested resource, or *thrd\_error*  
3453 if the request could not be honored.

3454 The *mtx\_trylock()* function shall return *thrd\_success* on success, or *thrd\_busy* if the  
3455 resource requested is already in use, or *thrd\_error* if the request could not be honored.  
3456 The *mtx\_trylock()* function can spuriously fail to lock an unused resource, in which case it  
3457 shall return *thrd\_busy*.

#### 3458 **ERRORS**

3459 See RETURN VALUE.

#### 3460 **EXAMPLES**

3461 None.

#### 3462 **APPLICATION USAGE**

3463 None.

#### 3464 **RATIONALE**

3465 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT  
3466 B.2.3].

3467 Since **<pthread.h>** has no equivalent of the *mtx\_timed* mutex property, if the **<threads.h>**  
3468 interfaces are implemented as a thin wrapper around **<pthread.h>** interfaces (meaning  
3469 **mtx\_t** and **pthread\_mutex\_t** are the same type), all mutexes support timeout and  
3470 *mtx\_timedlock()* will not fail for a mutex that was not initialized with *mtx\_timed*.  
3471 Alternatively, implementations can use a less thin wrapper where **mtx\_t** contains additional  
3472 properties that are not held in **pthread\_mutex\_t** in order to be able to return a failure  
3473 indication from *mtx\_timedlock()* calls where the mutex was not initialized with  
3474 *mtx\_timed*.

#### 3475 **FUTURE DIRECTIONS**

3476 None.

#### 3477 **SEE ALSO**

3478 *mtx\_destroy*, *timespec\_get*

3479 XBD Section 4.12.2, **<threads.h>**

#### 3480 **CHANGE HISTORY**

3481 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3482 Ref F.10.8.2 para 2

3483 On page 1388 line 46143 section *nan()*, add a new paragraph:

3484 [MX]The returned value shall be exact and shall be independent of the current rounding  
3485 direction mode.[/MX]

3486 Ref F.10.8.3 para 2, F.10.8.4 para 2

3487 On page 1395 line 46388 section *nextafter()*, add a new paragraph:

3488 [MX]Even though underflow or overflow can occur, the returned value shall be independent  
3489 of the current rounding direction mode.[/MX]

3490 Ref 7.22.3 para 2

3491 On page 1448 line 48069 section `posix_memalign()`, add a new (unshaded) paragraph:

3492 For purposes of determining the existence of a data race, `posix_memalign()` shall behave as  
3493 though it accessed only memory locations accessible through its arguments and not other  
3494 static duration storage. The function may, however, visibly modify the storage that it  
3495 allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `posix_memalign()`, `realloc()`,  
3496 and `reallocarray()` that allocate or deallocate a particular region of memory shall occur in a  
3497 single total order (see [xref to XBD 4.12.1]), and each such deallocation call shall  
3498 synchronize with the next allocation (if any) in this order.

3499 Ref 7.22.3.1

3500 On page 1449 line 48107 section `posix_memalign()`, add `aligned_alloc` to the SEE ALSO section.

3501 Ref F.10.4.4 para 1

3502 On page 1548 line 50724 section `pow()`, change:

3503 On systems that support the IEC 60559 Floating-Point option, if  $x$  is  $\pm 0$ , a pole error shall  
3504 occur and `pow()`, `powf()`, and `powl()` shall return `±HUGE_VAL`, `±HUGE_VALF`, and  
3505 `±HUGE_VALL`, respectively if  $y$  is an odd integer, or `HUGE_VAL`, `HUGE_VALF`, and  
3506 `HUGE_VALL`, respectively if  $y$  is not an odd integer.

3507 to:

3508 On systems that support the IEC 60559 Floating-Point option, if  $x$  is  $\pm 0$ :

- 3509 • if  $y$  is an odd integer, a pole error shall occur and `pow()`, `powf()`, and `powl()` shall  
3510 return `±HUGE_VAL`, `±HUGE_VALF`, and `±HUGE_VALL`, respectively;
- 3511 • if  $y$  is finite and is not an odd integer, a pole error shall occur and `pow()`, `powf()`, and  
3512 `powl()` shall return `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively;
- 3513 • if  $y$  is `-Inf`, a pole error may occur and `pow()`, `powf()`, and `powl()` shall return  
3514 `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

3515 Ref 7.26

3516 On page 1603 line 52244 section `pthread_cancel()`, add a new paragraph:

3517 If *thread* refers to a thread that was created using `thrd_create()`, the behavior is undefined.

3518 Ref 7.26.5.6

3519 On page 1603 line 52277 section `pthread_cancel()`, add a new RATIONALE paragraph:

3520 Use of `pthread_cancel()` to cancel a thread that was created using `thrd_create()` is undefined  
3521 because `thrd_join()` has no way to indicate a thread was cancelled. The standard developers  
3522 considered adding a `thrd_cancelled` enumeration constant that `thrd_join()` would return in  
3523 this case. However, this return would be unexpected in code that is written to conform to the  
3524 ISO C standard, and it would also not solve the problem that threads which use only ISO C

3525 <**threads.h**> interfaces (such as ones created by third party libraries written to conform to  
3526 the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not  
3527 provide cancellation cleanup handlers.

3528 Ref 7.26.5.5  
3529 On page 1639 line 53422 section `pthread_exit()`, change:

3530 `void pthread_exit(void *value_ptr);`

3531 to:

3532 `_Noreturn void pthread_exit(void *value_ptr);`

3533 Ref 7.26.6  
3534 On page 1639 line 53427 section `pthread_exit()`, change:

3535 After all cancellation cleanup handlers have been executed, if the thread has any thread-  
3536 specific data, appropriate destructor functions shall be called in an unspecified order.

3537 to:

3538 After all cancellation cleanup handlers have been executed, if the thread has any thread-  
3539 specific data (whether associated with key type `tss_t` or `pthread_key_t`), appropriate  
3540 destructor functions shall be called in an unspecified order.

3541 Ref 7.26.5.5  
3542 On page 1639 line 53432 section `pthread_exit()`, change:

3543 An implicit call to `pthread_exit()` is made when a thread other than the thread in which  
3544 `main()` was first invoked returns from the start routine that was used to create it.

3545 to:

3546 An implicit call to `pthread_exit()` is made when a thread that was not created using  
3547 `thrd_create()`, and is not the thread in which `main()` was first invoked, returns from the start  
3548 routine that was used to create it.

3549 Ref 7.26.5.5  
3550 On page 1639 line 53451 section `pthread_exit()`, change APPLICATION USAGE from:

3551 None.

3552 to:

3553 Calls to `pthread_exit()` should not be made from threads created using `thrd_create()`, as their  
3554 exit status has a different type (**int** instead of **void \***). If `pthread_exit()` is called from the  
3555 initial thread and it is not the last thread to terminate, other threads should not try to obtain  
3556 its exit status using `thrd_join()`.

3557 Ref 7.26.5.5  
3558 On page 1639 line 53453 section `pthread_exit()`, change:

3559 The normal mechanism by which a thread terminates is to return from the routine that was  
3560 specified in the *pthread\_create()* call that started it.

3561 to:

3562 The normal mechanism by which a thread that was started using *pthread\_create()* terminates  
3563 is to return from the routine that was specified in the *pthread\_create()* call that started it.

3564 Ref 7.26.5.5, 7.26.6

3565 On page 1640 line 53470 section *pthread\_exit()*, add *pthread\_key\_create*, *thrd\_create*, *thrd\_exit* and  
3566 *tss\_create* to the SEE ALSO section.

3567 Ref 7.26.5.5

3568 On page 1649 line 53748 section *pthread\_join()*, add a new paragraph:

3569 If *thread* refers to a thread that was created using *thrd\_create()* and the thread terminates, or  
3570 has already terminated, by returning from its start routine, the behavior of *pthread\_join()* is  
3571 undefined. If *thread* refers to a thread that terminates, or has already terminated, by calling  
3572 *thrd\_exit()*, the behavior of *pthread\_join()* is undefined.

3573 Ref 7.26.5.5

3574 On page 1651 line 53819 section *pthread\_join()*, add a new RATIONALE paragraph:

3575 The *pthread\_join()* function cannot be used to obtain the exit status of a thread that was  
3576 created using *thrd\_create()* and which terminates by returning from its start routine, or of a  
3577 thread that terminates by calling *thrd\_exit()*, because such threads have an **int** exit status,  
3578 instead of the **void \*** that *pthread\_join()* returns via its *value\_ptr* argument.

3579 Ref 7.22.4.7

3580 On page 1765 line 57040 insert the following new *quick\_exit()* section:

3581 **NAME**

3582 *quick\_exit* — terminate a process

3583 **SYNOPSIS**

3584 `#include <stdlib.h>`

3585 `_Noreturn void quick_exit(int status);`

3586 **DESCRIPTION**

3587 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3588 Any conflict between the requirements described here and the ISO C standard is  
3589 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3590 The *quick\_exit()* function shall cause normal process termination to occur. It shall not call  
3591 functions registered with *atexit()* nor any registered signal handlers. If a process calls the  
3592 *quick\_exit()* function more than once, or calls the *exit()* function in addition to the  
3593 *quick\_exit()* function, the behavior is undefined. If a signal is raised while the *quick\_exit()*  
3594 function is executing, the behavior is undefined.

3595 The *quick\_exit()* function shall first call all functions registered by *at\_quick\_exit()*, in the  
3596 reverse order of their registration, except that a function is called after any previously

3597 registered functions that had already been called at the time it was registered. If, during the  
3598 call to any such function, a call to the *longjmp()* [CX] or *siglongjmp()*[/CX] function is made  
3599 that would terminate the call to the registered function, the behavior is undefined.

3600 If a function registered by a call to *at\_quick\_exit()* fails to return, the remaining registered  
3601 functions shall not be called and the rest of the *quick\_exit()* processing shall not be  
3602 completed.

3603 Finally, the *quick\_exit()* function shall terminate the process as if by a call to *\_Exit(status)*.

#### 3604 **RETURN VALUE**

3605 The *quick\_exit()* function does not return.

#### 3606 **ERRORS**

3607 No errors are defined.

#### 3608 **EXAMPLES**

3609 None.

#### 3610 **APPLICATION USAGE**

3611 None.

#### 3612 **RATIONALE**

3613 None.

#### 3614 **FUTURE DIRECTIONS**

3615 None.

#### 3616 **SEE ALSO**

3617 *\_Exit*, *at\_quick\_exit*, *atexit*, *exit*

3618 XBD <stdlib.h>

#### 3619 **CHANGE HISTORY**

3620 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3621 Ref 7.22.2.1 para 3, 7.1.4 para 5

3622 On page 1767 line 57095 section *rand()*, change:

3623 [CX]The *rand()* function need not be thread-safe.[/CX]

3624 to:

3625 The *rand()* function need not be thread-safe; however, *rand()* shall avoid data races with all  
3626 functions other than non-thread-safe pseudo-random sequence generation functions.

3627 Ref 7.22.2.2 para 3, 7.1.4 para 5

3628 On page 1767 line 57105 section *rand()*, add a new paragraph:

3629 The *srand()* function need not be thread-safe; however, *srand()* shall avoid data races with  
3630 all functions other than non-thread-safe pseudo-random sequence generation functions.

3631 Ref 7.22.3 para 1,2; 7.22.3.5 para 2,3,4; 7.31.12 para 2  
3632 On page 1788 line 57862-57892 section `realloc()`, [after applying bugs 374 and 1218](#) replace the  
3633 DESCRIPTION and RETURN VALUE sections with:

## 3634 DESCRIPTION

3635 [For `realloc\(\)`](#): [CX] The functionality described on this reference page is aligned with the  
3636 ISO C standard. Any conflict between the requirements described here and the ISO C  
3637 standard is unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3638 The `realloc()` function shall deallocate the old object pointed to by *ptr* and return a pointer to  
3639 a new object that has the size specified by *size*. The contents of the new object shall be the  
3640 same as that of the old object prior to deallocation, up to the lesser of the new and old sizes.  
3641 Any bytes in the new object beyond the size of the old object have indeterminate values.

3642 [\[CX\]The `reallocarray\(\)` function shall be equivalent to the call `realloc\(ptr, nelem \*  
3643 elsize\)` except that overflow in the multiplication shall be an error.\[/CX\]](#)

3644 If *ptr* is a null pointer, `realloc()` [\[CX\]or `reallocarray\(\)`](#)[/CX] shall be equivalent to `malloc()`  
3645 function for the specified size. Otherwise, if *ptr* does not match a pointer returned earlier by  
3646 `aligned_alloc()`, `calloc()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV] `realloc()`,  
3647 [\[CX\]`reallocarray\(\)`](#), or a function in POSIX.1-20xx that allocates memory as if by `malloc()`,  
3648 [\[/CX\]](#) or if the space has been deallocated by a call to `free()`, [\[CX\]`reallocarray\(\)`](#),[/CX] or  
3649 `realloc()`, the behavior is undefined.

3650 If *size* is non-zero and memory for the new object is not allocated, the old object shall not be  
3651 deallocated. ~~[OB]If *size* is zero and memory for the new object is not allocated, it is  
3652 implementation-defined whether the old object is deallocated; if the old object is not  
3653 deallocated, its value shall be unchanged.[/OB]~~

3654 The order and contiguity of storage allocated by successive calls to `realloc()` [\[CX\]or  
3655 `reallocarray\(\)`](#)[/CX] is unspecified. The pointer returned if the allocation succeeds shall be  
3656 suitably aligned so that it may be assigned to a pointer to any type of object with a  
3657 fundamental alignment requirement and then used to access such an object in the space  
3658 allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a  
3659 pointer to an object disjoint from any other object. The pointer returned shall point to the  
3660 start (lowest byte address) of the allocated space. If the space cannot be allocated, a null  
3661 pointer shall be returned. ~~[OB]If the size of the space requested is 0, the behavior is  
3662 implementation-defined: either a null pointer shall be returned to indicate an error, or the  
3663 behavior shall be as if the size were some non-zero value, except that the behavior is  
3664 undefined if the returned pointer is used to access an object.[/OB]~~

3665 For purposes of determining the existence of a data race, `realloc()` [\[CX\]or  
3666 `reallocarray\(\)`](#)[/CX] shall behave as though it accessed only memory locations accessible  
3667 through its arguments and not other static duration storage. The function may, however,  
3668 visibly modify the storage that it allocates or deallocates. Calls to `aligned_alloc()`, `calloc()`,  
3669 `free()`, `malloc()`, [ADV]`posix_memalign()`,[/ADV] [\[CX\]`reallocarray\(\)`](#),[/CX] and `realloc()`  
3670 that allocate or deallocate a particular region of memory shall occur in a single total order  
3671 (see [xref to XBD 4.12.1]), and each such deallocation call shall synchronize with the next  
3672 allocation (if any) in this order.

## 3673 RETURN VALUE

3674 | ~~The~~ Upon successful completion, *realloc()* [CX]and *reallocarray()*[/CX]function shall return  
3675 | a pointer to the new object (which can have the same value as a pointer to the old object), or  
3676 | a null pointer if the new object has not been allocated.

3677 | [OB]If size is zero,[/OB]  
3678 | [OB CX]or either *nelem* or *elsize* is 0,[/OB CX]  
3679 | [OB]either:

- 3680 | • A null pointer shall be returned [CX]and, if *ptr* is not a null pointer, *errno* shall be set  
3681 | to ~~an implementation-defined value~~[EINVAL].[/CX]
- 3682 | • A pointer to the allocated space shall be returned, and the memory object pointed to  
3683 | by *ptr* shall be freed. The application shall ensure that the pointer is not used to  
3684 | access an object.[/OB]

3685 | If there is not enough available memory, *realloc()* [CX]and *reallocarray()*[/CX] shall return  
3686 | a null pointer [CX]and set *errno* to [ENOMEM][/CX].

3687 | Ref 7.22.3.5 para 3,4  
3688 | On page 1789 line 57899 section *realloc()*, change:

3689 | The description of *realloc()* has been modified from previous versions of this standard to  
3690 | align with the ISO/IEC 9899: 1999 standard. Previous versions explicitly permitted a call to  
3691 | *realloc(p, 0)* to free the space pointed to by *p* and return a null pointer. While this behavior  
3692 | could be interpreted as permitted by this version of the standard, the C language committee  
3693 | have indicated that this interpretation is incorrect. Applications should assume that if  
3694 | *realloc()* returns a null pointer, the space pointed to by *p* has not been freed. Since this could  
3695 | lead to double-frees, implementations should also set *errno* if a null pointer actually  
3696 | indicates a failure, and applications should only free the space if *errno* was changed.

3697 | to:

3698 | The ISO C standard makes it implementation-defined whether a call to *realloc(p, 0)* frees the  
3699 | space pointed to by *p* if it returns a null pointer because memory for the new object was not  
3700 | allocated. POSIX.1 instead requires that implementations set *errno* if a null pointer is  
3701 | returned and the space has not been freed, and POSIX applications should only free the  
3702 | space if *errno* was changed.

3703 | Ref 7.31.12 para 2  
3704 | On page 1789 line 57909-57912 section *realloc()*, change FUTURE DIRECTIONS to:

3705 | The ISO C standard states that invoking *realloc()* with a *size* argument equal to zero is an  
3706 | obsolescent feature. This feature may be removed in a future version of this standard.

3707 | Ref 7.22.3.1  
3708 | On page 1789 line 57914 section *realloc()*, add *aligned\_alloc* to the SEE ALSO section.

3709 | Ref F.10.7.2 para 2  
3710 | On page 1809 line 58638 section *remainder()*, add a new paragraph:

3711 | [MX]When subnormal results are supported, the returned value shall be exact.[/MX]

3712 | Ref F.10.7.3 para 2

- 3713 On page 1814 line 58758 section `remquo()`, add a new paragraph:
- 3714 [MX]When subnormal results are supported, the returned value shall be exact.[/MX]
- 3715 Ref F.10.6.6 para 3
- 3716 On page 1828 line 59258 section `round()`, add a new paragraph:
- 3717 [MX]These functions may raise the inexact floating-point exception for finite non-integer  
3718 arguments.[/MX]
- 3719 Ref F.10.6.6 para 3
- 3720 On page 1828 line 59272 section `round()`, delete from APPLICATION USAGE:
- 3721 These functions may raise the inexact floating-point exception if the result differs in value  
3722 from the argument.
- 3723 Ref F.10.3.13 para 2
- 3724 On page 1829 line 59306 section `scalbln()`, add a new paragraph:
- 3725 [MX]If the calculation does not overflow or underflow, the returned value shall be exact and  
3726 shall be independent of the current rounding direction mode.[/MX]
- 3727 Ref 7.11.1.1 para 5
- 3728 On page 1903 line 61520 section `setlocale()`, change:
- 3729 [CX]The `setlocale()` function need not be thread-safe.[/CX]
- 3730 to:
- 3731 The `setlocale()` function need not be thread-safe; however, it shall avoid data races with all  
3732 function calls that do not affect and are not affected by the global locale.
- 3733 Ref 7.13.2.1 para 1
- 3734 On page 1970 line 63497 section `siglongjmp()`, change:
- 3735 `void siglongjmp(sigjmp_buf env, int val);`
- 3736 to:
- 3737 `_Noreturn void siglongjmp(sigjmp_buf env, int val);`
- 3738 Ref 7.13.2.1 para 4
- 3739 On page 1970 line 63504 section `siglongjmp()`, change:
- 3740 After `siglongjmp()` is completed, program execution shall continue ...
- 3741 to:
- 3742 After `siglongjmp()` is completed, thread execution shall continue ...
- 3743 Ref 7.14.1.1 para 5
- 3744 On page 1971 line 63564 section `signal()`, change:



3745 with static storage duration

3746 to:

3747 with static or thread storage duration that is not a lock-free atomic object

3748 Ref 7.14.1.1 para 7  
3749 On page 1972 line 63573 section `signal()`, add a new paragraph:

3750 [CX]The `signal()` function is required to be thread-safe. (See [xref to 2.9.1 Thread-Safety].)  
3751 [/CX]

3752 Ref 7.14.1.1 para 7  
3753 On page 1972 line 63591 section `signal()`, change RATIONALE from:

3754 None.

3755 to:

3756 The ISO C standard says that the use of `signal()` in a multi-threaded program results in  
3757 undefined behavior. However, POSIX.1 has required `signal()` to be thread-safe since before  
3758 threads were added to the ISO C standard.

3759 Ref F.10.4.5 para 1  
3760 On page 2009 line 64624 section `sqrt()`, add:

3761 [MX]The returned value shall be dependent on the current rounding direction mode.[/MX]

3762 Ref 7.24.6.2 para 3, 7.1.4 para 5  
3763 On page 2035 line 65231 section `strerror()`, change:

3764 [CX]The `strerror()` function need not be thread-safe.[/CX]

3765 to:

3766 The `strerror()` function need not be thread-safe; however, `strerror()` shall avoid data races  
3767 with all other functions.

3768 Ref 7.22.1.3 para 10  
3769 On page 2073 line 66514 section `strtod()`, change:

3770 If the correct value is outside the range of representable values

3771 to:  
3772 If the correct value would cause an overflow and default rounding is in effect

3773 Ref 7.24.5.8 para 6, 7.1.4 para 5  
3774 On page 2078 line 66674 section `strtok()`, change:

3775 [CX]The `strtok()` function need not be thread-safe.[/CX]

3776 to:

3777 The *strtok()* function need not be thread-safe; however, *strtok()* shall avoid data races with  
3778 all other functions.

3779 Ref 7.22.4.8, 7.1.4 para 5  
3780 On page 2107 line 67579 section *system()*, change:

3781 The *system()* function need not be thread-safe.

3782 to:

3783 [CX]If concurrent calls to *system()* are made from multiple threads, it is unspecified  
3784 whether:

- 3785 • each call saves and restores the dispositions of the SIGINT and SIGQUIT signals  
3786 independently, or
- 3787 • in a set of concurrent calls the dispositions in effect after the last call returns are  
3788 those that were in effect on entry to the first call.

3789 If a thread is cancelled while it is in a call to *system()*, it is unspecified whether the child  
3790 process is terminated and waited for, or is left running.[/CX]

3791 Ref 7.22.4.8, 7.1.4 para 5  
3792 On page 2108 line 67627 section *system()*, change:

3793 Using the *system()* function in more than one thread in a process or when the SIGCHLD  
3794 signal is being manipulated by more than one thread in a process may produce unexpected  
3795 results.

3796 to:

3797 Although *system()* is required to be thread-safe, it is recommended that concurrent calls  
3798 from multiple threads are avoided, since *system()* is not required to coordinate the saving  
3799 and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of  
3800 overlapping calls, and therefore the signals might end up being set to ignored after the last  
3801 call returns. Applications should also avoid cancelling a thread while it is in a call to  
3802 *system()* as the child process may be left running in that event. In addition, if another thread  
3803 alters the disposition of the SIGCHLD signal, a call to *signal()* may produce unexpected  
3804 results.

3805 Ref 7.22.4.8, 7.1.4 para 5  
3806 On page 2109 line 67675 section *system()*, delete:

3807 `#include <signal.h>`

3808 Ref 7.22.4.8, 7.1.4 para 5  
3809 On page 2109 line 67692,67696,67712 section *system()*, change *sigprocmask* to  
3810 *pthread\_sigmask*.

3811 Ref 7.22.4.8, 7.1.4 para 5  
3812 On page 2110 line 67718 section *system()*, change:

3813 Note also that the above example implementation is not thread-safe. Implementations can  
3814 provide a thread-safe *system()* function, but doing so involves complications such as how to  
3815 restore the signal dispositions for SIGINT and SIGQUIT correctly if there are overlapping  
3816 calls, and how to deal with cancellation. The example above would not restore the signal  
3817 dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-  
3818 safe implementation since canceling a non-thread-safe function results in undefined  
3819 behavior (see Section 2.9.5.2, on page 518). To avoid leaking a process ID, a thread-safe  
3820 implementation would need to terminate the child process when acting on a cancellation.

3821 to:

3822 Earlier versions of this standard did not require *system()* to be thread-safe because it alters  
3823 the process-wide disposition of the SIGINT and SIGQUIT signals. It is now required to be  
3824 thread-safe to align with the ISO C standard, which (since the introduction of threads in  
3825 2011) requires that it avoids data races. However, the function is not required to coordinate  
3826 the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set  
3827 of overlapping calls, and the above example does not do so. The example also does not  
3828 terminate and wait for the child process if the calling thread is cancelled, and so would leak  
3829 a process ID in that event.

3830 Ref 7.26.5

3831 On page 2148 line 68796 insert the following new *thrd\_\**() sections:

3832 **NAME**

3833 *thrd\_create* — thread creation

3834 **SYNOPSIS**

3835 `#include <threads.h>`

3836 `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`

3837 **DESCRIPTION**

3838 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3839 Any conflict between the requirements described here and the ISO C standard is  
3840 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3841 The *thrd\_create()* function shall create a new thread executing *func(arg)*. If the *thrd\_create()*  
3842 function succeeds, it shall set the object pointed to by *thr* to the identifier of the newly  
3843 created thread. (A thread's identifier might be reused for a different thread once the original  
3844 thread has exited and either been detached or joined to another thread.) The completion of  
3845 the *thrd\_create()* function shall synchronize with the beginning of the execution of the new  
3846 thread.

3847 [CX]The signal state of the new thread shall be initialized as follows:

- 3848 • The signal mask shall be inherited from the creating thread.
- 3849 • The set of signals pending for the new thread shall be empty.

3850 The thread-local current locale shall not be inherited from the creating thread.

3851 The floating-point environment shall be inherited from the creating thread.[/CX]

3852 [XSI] The alternate stack shall not be inherited from the creating thread.[/XSI]

3853 Returning from *func* shall have the same behavior as invoking *thrd\_exit()* with the value  
3854 returned from *func*.

3855 If *thrd\_create()* fails, no new thread shall be created and the contents of the location  
3856 referenced by *thr* are undefined.

3857 [CX]The *thrd\_create()* function shall not be affected if the calling thread executes a signal  
3858 handler during the call.[/CX]

3859 **RETURN VALUE**  
3860 The *thrd\_create()* function shall return `thrd_success` on success; or `thrd_nomem` if no  
3861 memory could be allocated for the thread requested; or `thrd_error` if the request could not  
3862 be honored, [CX]such as if the system-imposed limit on the total number of threads in a  
3863 process `{PTHREAD_THREADS_MAX}` would be exceeded.[/CX]

3864 **ERRORS**  
3865 See RETURN VALUE.

3866 **EXAMPLES**  
3867 None.

3868 **APPLICATION USAGE**  
3869 There is no requirement on the implementation that the ID of the created thread be available  
3870 before the newly created thread starts executing. The calling thread can obtain the ID of the  
3871 created thread through the *thr* argument of the *thrd\_create()* function, and the newly created  
3872 thread can obtain its ID by a call to *thrd\_current()*.

3873 **RATIONALE**  
3874 The *thrd\_create()* function is not affected by signal handlers for the reasons stated in [xref to  
3875 XRAT B.2.3].

3876 **FUTURE DIRECTIONS**  
3877 None.

3878 **SEE ALSO**  
3879 *pthread\_create*, *thrd\_current*, *thrd\_detach*, *thrd\_exit*, *thrd\_join*

3880 XBD Section 4.12.2, <**threads.h**>

3881 **CHANGE HISTORY**  
3882 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3883 **NAME**  
3884 `thrd_current` — get the calling thread ID

3885 **SYNOPSIS**  
3886 `#include <threads.h>`

3887 `thrd_t thrd_current(void);`

3888 **DESCRIPTION**

3889 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3890 Any conflict between the requirements described here and the ISO C standard is  
3891 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3892 The *thrd\_current()* function shall identify the thread that called it.

3893 **RETURN VALUE**

3894 The *thrd\_current()* function shall return the thread ID of the thread that called it.

3895 The *thrd\_current()* function shall always be successful. No return value is reserved to  
3896 indicate an error.

3897 **ERRORS**

3898 No errors are defined.

3899 **EXAMPLES**

3900 None.

3901 **APPLICATION USAGE**

3902 None.

3903 **RATIONALE**

3904 None.

3905 **FUTURE DIRECTIONS**

3906 None.

3907 **SEE ALSO**

3908 *pthread\_self*, *thrd\_create*, *thrd\_equal*

3909 XBD Section 4.12.2, <**threads.h**>

3910 **CHANGE HISTORY**

3911 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3912 **NAME**

3913 *thrd\_detach* — detach a thread

3914 **SYNOPSIS**

3915 `#include <threads.h>`

3916 `int thrd_detach(thrd_t thr);`

3917 **DESCRIPTION**

3918 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3919 Any conflict between the requirements described here and the ISO C standard is  
3920 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3921 The *thrd\_detach()* function shall change the thread *thr* from joinable to detached, indicating  
3922 to the implementation that any resources allocated to the thread can be reclaimed when that

3923 thread terminates. The application shall ensure that the thread identified by *thr* has not been  
3924 previously detached or joined with another thread.

3925 [CX]The *thr\_detach()* function shall not be affected if the calling thread executes a signal  
3926 handler during the call.[/CX]

#### 3927 **RETURN VALUE**

3928 The *thr\_detach()* function shall return *thr\_success* on success or *thr\_error* if the  
3929 request could not be honored.

#### 3930 **ERRORS**

3931 No errors are defined.

#### 3932 **EXAMPLES**

3933 None.

#### 3934 **APPLICATION USAGE**

3935 None.

#### 3936 **RATIONALE**

3937 The *thr\_detach()* function is not affected by signal handlers for the reasons stated in [xref  
3938 to XRAT B.2.3].

#### 3939 **FUTURE DIRECTIONS**

3940 None.

#### 3941 **SEE ALSO**

3942 *pthread\_detach*, *thr\_create*, *thr\_join*

3943 XBD <**threads.h**>

#### 3944 **CHANGE HISTORY**

3945 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

#### 3946 **NAME**

3947 *thr\_equal* — compare thread IDs

#### 3948 **SYNOPSIS**

3949 `#include <threads.h>`

3950 `int thr_equal(thrd_t thr0, thrd_t thr1);`

#### 3951 **DESCRIPTION**

3952 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3953 Any conflict between the requirements described here and the ISO C standard is  
3954 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3955 The *thr\_equal()* function shall determine whether the thread identified by *thr0* refers to the  
3956 thread identified by *thr1*.

3957 [CX]The *thr\_equal()* function shall not be affected if the calling thread executes a signal  
3958 handler during the call.[/CX]

3959 **RETURN VALUE**

3960 The *thrd\_equal()* function shall return a non-zero value if *thr0* and *thr1* are equal; otherwise,  
3961 zero shall be returned.

3962 If either *thr0* or *thr1* is not a valid thread ID [CX]and is not equal to PTHREAD\_NULL  
3963 (which is defined in <pthread.h>)[/CX], the behavior is undefined.

3964 **ERRORS**

3965 No errors are defined.

3966 **EXAMPLES**

3967 None.

3968 **APPLICATION USAGE**

3969 None.

3970 **RATIONALE**

3971 See the RATIONALE section for *pthread\_equal()*.

3972 The *thrd\_equal()* function is not affected by signal handlers for the reasons stated in [xref to  
3973 XRAT B.2.3].

3974 **FUTURE DIRECTIONS**

3975 None.

3976 **SEE ALSO**

3977 *pthread\_equal*, *thrd\_current*

3978 XBD <pthread.h>, <threads.h>

3979 **CHANGE HISTORY**

3980 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

3981 **NAME**

3982 *thrd\_exit* — thread termination

3983 **SYNOPSIS**

3984 #include <threads.h>

3985 \_Noreturn void *thrd\_exit*(int *res*);

3986 **DESCRIPTION**

3987 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
3988 Any conflict between the requirements described here and the ISO C standard is  
3989 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

3990 For every thread-specific storage key [CX](regardless of whether it has type **tss\_t** or  
3991 **pthread\_key\_t**)[/CX] which was created with a non-null destructor and for which the value  
3992 is non-null, *thrd\_exit()* shall set the value associated with the key to a null pointer value and  
3993 then invoke the destructor with its previous value. The order in which destructors are  
3994 invoked is unspecified.

3995 If after this process there remain keys with both non-null destructors and values, the  
3996 implementation shall repeat this process up to [CX]  
3997 {PTHREAD\_DESTRUCTOR\_ITERATIONS}[/CX] times.

3998 Following this, the *thrd\_exit()* function shall terminate execution of the calling thread and  
3999 shall set its exit status to *res*. [CX]Thread termination shall not release any application  
4000 visible process resources, including, but not limited to, mutexes and file descriptors, nor  
4001 shall it perform any process-level cleanup actions, including, but not limited to, calling any  
4002 *atexit()* routines that might exist.[/CX]

4003 An implicit call to *thrd\_exit()* is made when a thread that was created using *thrd\_create()*  
4004 returns from the start routine that was used to create it (see [xref to *thrd\_create()*]).

4005 [CX]The behavior of *thrd\_exit()* is undefined if called from a destructor function that was  
4006 invoked as a result of either an implicit or explicit call to *thrd\_exit()*.[/CX]

4007 The process shall exit with an exit status of zero after the last thread has been terminated.  
4008 The behavior shall be as if the implementation called *exit()* with a zero argument at thread  
4009 termination time.

#### 4010 RETURN VALUE

4011 This function shall not return a value.

#### 4012 ERRORS

4013 No errors are defined.

#### 4014 EXAMPLES

4015 None.

#### 4016 APPLICATION USAGE

4017 Calls to *thrd\_exit()* should not be made from threads created using *pthread\_create()* or via a  
4018 SIGEV\_THREAD notification, as their exit status has a different type (**void \*** instead of  
4019 **int**). If *thrd\_exit()* is called from the initial thread and it is not the last thread to terminate,  
4020 other threads should not try to obtain its exit status using *pthread\_join()*.

#### 4021 RATIONALE

4022 The normal mechanism by which a thread that was started using *thrd\_create()* terminates is  
4023 to return from the function that was specified in the *thrd\_create()* call that started it. The  
4024 *thrd\_exit()* function provides the capability for such a thread to terminate without requiring a  
4025 return from the start routine of that thread, thereby providing a function analogous to *exit()*.

4026 Regardless of the method of thread termination, the destructors for any existing thread-  
4027 specific data are executed.

#### 4028 FUTURE DIRECTIONS

4029 None.

#### 4030 SEE ALSO

4031 *exit*, *pthread\_create*, *thrd\_join*

4032 XBD <**threads.h**>



4033 **CHANGE HISTORY**

4034 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4035 **NAME**

4036 `thrd_join` — wait for thread termination

4037 **SYNOPSIS**

4038 `#include <threads.h>`

4039 `int thrd_join(thrd_t thr, int *res);`

4040 **DESCRIPTION**

4041 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
4042 Any conflict between the requirements described here and the ISO C standard is  
4043 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4044 The `thrd_join()` function shall join the thread identified by `thr` with the current thread by  
4045 blocking until the other thread has terminated. If the parameter `res` is not a null pointer,  
4046 `thrd_join()` shall store the thread's exit status in the integer pointed to by `res`. The  
4047 termination of the other thread shall synchronize with the completion of the `thrd_join()`  
4048 function. The application shall ensure that the thread identified by `thr` has not been  
4049 previously detached or joined with another thread.

4050 The results of multiple simultaneous calls to `thrd_join()` specifying the same target thread  
4051 are undefined.

4052 The behavior is undefined if the value specified by the `thr` argument to `thrd_join()` refers to  
4053 the calling thread.

4054 [CX]It is unspecified whether a thread that has exited but remains unjoined counts against  
4055 `{PTHREAD_THREADS_MAX}`.

4056 If `thr` refers to a thread that was created using `pthread_create()` or via a `SIGEV_THREAD`  
4057 notification and the thread terminates, or has already terminated, by returning from its start  
4058 routine, the behavior of `thrd_join()` is undefined. If `thr` refers to a thread that terminates, or  
4059 has already terminated, by calling `pthread_exit()` or by being cancelled, the behavior of  
4060 `thrd_join()` is undefined.

4061 The `thrd_join()` function shall not be affected if the calling thread executes a signal handler  
4062 during the call.[/CX]

4063 **RETURN VALUE**

4064 The `thrd_join()` function shall return `thrd_success` on success or `thrd_error` if the  
4065 request could not be honored.

4066 [CX]It is implementation-defined whether `thrd_join()` detects deadlock situations; if it does  
4067 detect them, it shall return `thrd_error` when one is detected.[/CX]

4068 **ERRORS**

4069 See RETURN VALUE.

4070 **EXAMPLES**

4071 None.

## 4072 APPLICATION USAGE

4073 None.

## 4074 RATIONALE

4075 The *thrd\_join()* function provides a simple mechanism allowing an application to wait for a  
4076 thread to terminate. After the thread terminates, the application may then choose to clean up  
4077 resources that were used by the thread. For instance, after *thrd\_join()* returns, any  
4078 application-provided stack storage could be reclaimed.

4079 The *thrd\_join()* or *thrd\_detach()* function should eventually be called for every thread that is  
4080 created using *thrd\_create()* so that storage associated with the thread may be reclaimed.

4081 The *thrd\_join()* function cannot be used to obtain the exit status of a thread that was created  
4082 using *pthread\_create()* or via a SIGEV\_THREAD notification and which terminates by  
4083 returning from its start routine, or of a thread that terminates by calling *pthread\_exit()*,  
4084 because such threads have a **void \*** exit status, instead of the **int** that *thrd\_join()* returns via  
4085 its *res* argument.

4086 The *thrd\_join()* function cannot be used to obtain the exit status of a thread that terminates  
4087 by being cancelled because it has no way to indicate that a thread was cancelled. (The  
4088 *pthread\_join()* function does this by returning a reserved **void \*** exit status; it is not possible  
4089 to reserve an **int** value for this purpose without introducing a conflict with the ISO C  
4090 standard.) The standard developers considered adding a *thrd\_cancelled* enumeration  
4091 constant that *thrd\_join()* would return in this case. However, this return would be  
4092 unexpected in code that is written to conform to the ISO C standard, and it would also not  
4093 solve the problem that threads which use only ISO C **<threads.h>** interfaces (such as ones  
4094 created by third party libraries written to conform to the ISO C standard) have no way to  
4095 handle being cancelled, as the ISO C standard does not provide cancellation cleanup  
4096 handlers.

4097 The *thrd\_join()* function is not affected by signal handlers for the reasons stated in [xref to  
4098 XRAT B.2.3].

## 4099 FUTURE DIRECTIONS

4100 None.

## 4101 SEE ALSO

4102 *pthread\_create*, *pthread\_exit*, *pthread\_join*, *thrd\_create*, *thrd\_exit*

4103 XBD Section 4.12.2, **<threads.h>**

## 4104 CHANGE HISTORY

4105 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

## 4106 NAME

4107 *thrd\_sleep* — suspend execution for an interval

## 4108 SYNOPSIS

4109 `#include <threads.h>`

4110 `int thrd_sleep(const struct timespec *duration,`  
4111 `struct timespec *remaining);`

4112 **DESCRIPTION**

4113 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
4114 Any conflict between the requirements described here and the ISO C standard is  
4115 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4116 The *thrd\_sleep()* function shall suspend execution of the calling thread until either the  
4117 interval specified by *duration* has elapsed or a signal is delivered to the calling thread whose  
4118 action is to invoke a signal-catching function or to terminate the process. If interrupted by a  
4119 signal and the *remaining* argument is not null, the amount of time remaining (the requested  
4120 interval minus the time actually slept) shall be stored in the interval it points to. The  
4121 *duration* and *remaining* arguments can point to the same object.

4122 The suspension time may be longer than requested because the interval is rounded up to an  
4123 integer multiple of the sleep resolution or because of the scheduling of other activity by the  
4124 system. But, except for the case of being interrupted by a signal, the suspension time shall  
4125 not be less than that specified, as measured by the system clock `TIME_UTC`.

4126 **RETURN VALUE**

4127 The *thrd\_sleep()* function shall return zero if the requested time has elapsed, -1 if it has  
4128 been interrupted by a signal, or a negative value (which may also be -1) if it fails for any  
4129 other reason. [CX]If it returns a negative value, it shall set *errno* to indicate the error.[/CX]

4130 **ERRORS**

4131 [CX]The *thrd\_sleep()* function shall fail if:

4132 [EINTR]

4133 The *thrd\_sleep()* function was interrupted by a signal.

4134 [EINVAL]

4135 The *duration* argument specified a nanosecond value less than zero or greater than or  
4136 equal to 1000 million.[/CX]

4137 **EXAMPLES**

4138 None.

4139 **APPLICATION USAGE**

4140 Since the return value may be -1 for errors other than [EINTR], applications should examine  
4141 *errno* to distinguish [EINTR] from other errors (and thus determine whether the unslept time  
4142 is available in the interval pointed to by *remaining*).

4143 **RATIONALE**

4144 The *thrd\_sleep()* function is identical to the *nanosleep()* function except that the return value  
4145 may be any negative value when it fails with an error other than [EINTR].

4146 **FUTURE DIRECTIONS**

4147 None.

4148 **SEE ALSO**

4149 *nanosleep*

4150 XBD <**threads.h**>, <**time.h**>

4151 **CHANGE HISTORY**  
4152 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4153 **NAME**  
4154 `thrd_yield` — yield the processor

4155 **SYNOPSIS**  
4156 `#include <threads.h>`  
4157 `void thrd_yield(void);`

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 [CX]The `thrd_yield()` function shall force the running thread to relinquish the processor until  
4163 it again becomes the head of its thread list.[/CX]

4164 **RETURN VALUE**  
4165 This function shall not return a value.

4166 **ERRORS**  
4167 No errors are defined.

4168 **EXAMPLES**  
4169 None.

4170 **APPLICATION USAGE**  
4171 See the APPLICATION USAGE section for `sched_yield()`.

4172 **RATIONALE**  
4173 The `thrd_yield()` function is identical to the `sched_yield()` function except that it does not  
4174 return a value.

4175 **FUTURE DIRECTIONS**  
4176 None.

4177 **SEE ALSO**  
4178 `sched_yield`  
4179 XBD <**threads.h**>

4180 **CHANGE HISTORY**  
4181 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4182 Ref 7.27.2.5  
4183 On page 2161 line 69278 insert a new `timespec_get()` section:

4184 **NAME**

4185 `timespec_get` — get time

4186 **SYNOPSIS**

4187 `#include <time.h>`

4188 `int timespec_get(struct timespec *ts, int base);`

4189 **DESCRIPTION**

4190 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
4191 Any conflict between the requirements described here and the ISO C standard is  
4192 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4193 The `timespec_get()` function shall set the interval pointed to by `ts` to hold the current  
4194 calendar time based on the specified time base.

4195 [CX]If `base` is `TIME_UTC`, the members of `ts` shall be set to the same values as would be  
4196 set by a call to `clock_gettime(CLOCK_REALTIME, ts)`. If the number of seconds will not  
4197 fit in an object of type `time_t`, the function shall return zero.[/CX]

4198 **RETURN VALUE**

4199 If the `timespec_get()` function is successful it shall return the non-zero value `base`; otherwise,  
4200 it shall return zero.

4201 **ERRORS**

4202 See DESCRIPTION.

4203 **EXAMPLES**

4204 None.

4205 **APPLICATION USAGE**

4206 None.

4207 **RATIONALE**

4208 None.

4209 **FUTURE DIRECTIONS**

4210 None.

4211 **SEE ALSO**

4212 `clock_getres`, `time`

4213 XBD `<time.h>`

4214 **CHANGE HISTORY**

4215 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4216 Ref 7.21.4.4 para 4, 7.1.4 para 5

4217 On page 2164 line 69377 section `tmpnam()`, change:

4218 [CX]The `tmpnam()` function need not be thread-safe if called with a NULL parameter.[/CX]

4219 to:

4220 If called with a null pointer argument, the *tmpnam()* function need not be thread-safe;  
4221 however, such calls shall avoid data races with calls to *tmpnam()* with a non-null argument  
4222 and with calls to all other functions.

4223 Ref 7.30.3.2.1 para 4

4224 On page 2171 line 69568 section *towctrans()*, change:

4225 If successful, the *towctrans()* [CX]and *towctrans\_l()*[/CX] functions shall return the mapped  
4226 value of *wc* using the mapping described by *desc*. Otherwise, they shall return *wc*  
4227 unchanged.

4228 to:

4229 If successful, the *towctrans()* [CX]and *towctrans\_l()*[/CX] functions shall return the mapped  
4230 value of *wc* using the mapping described by *desc*, or the value of *wc* unchanged if *desc* is  
4231 zero. [CX]Otherwise, they shall return *wc* unchanged.[/CX]

4232 Ref F.10.6.8 para 2

4233 On page 2177 line 69716 section *trunc()*, add a new paragraph:

4234 [MX]These functions may raise the inexact floating-point exception for finite non-integer  
4235 arguments.[/MX]

4236 Ref F.10.6.8 para 1,2

4237 On page 2177 line 69719 section *trunc()*, change:

4238 [MX]The result shall have the same sign as *x*.[/MX]

4239 to:

4240 [MX]The returned value shall be exact, shall be independent of the current rounding  
4241 direction mode, and shall have the same sign as *x*.[/MX]

4242 Ref F.10.6.8 para 2

4243 On page 2177 line 69730 section *trunc()*, delete from APPLICATION USAGE:

4244 These functions may raise the inexact floating-point exception if the result differs in value  
4245 from the argument.

4246 Ref 7.26.6

4247 On page 2182 line 69835 insert the following new *tss\_\**() sections:

4248 **NAME**

4249 *tss\_create* — thread-specific data key creation

4250 **SYNOPSIS**

4251 `#include <threads.h>`

4252 `int tss_create(tss_t *key, tss_dtor_t dtor);`

4253 **DESCRIPTION**

4254 [CX] The functionality described on this reference page is aligned with the ISO C standard.

4255 Any conflict between the requirements described here and the ISO C standard is  
4256 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4257 The *tss\_create()* function shall create a thread-specific storage pointer with destructor *dtor*,  
4258 which can be null.

4259 A null pointer value shall be associated with the newly created key in all existing threads.  
4260 Upon subsequent thread creation, the value associated with all keys shall be initialized to a  
4261 null pointer value in the new thread.

4262 Destructors associated with thread-specific storage shall not be invoked at process  
4263 termination.

4264 The behavior is undefined if the *tss\_create()* function is called from within a destructor.

4265 [CX]The *tss\_create()* function shall not be affected if the calling thread executes a signal  
4266 handler during the call.[/CX]

#### 4267 **RETURN VALUE**

4268 If the *tss\_create()* function is successful, it shall set the thread-specific storage pointed to by  
4269 *key* to a value that uniquely identifies the newly created pointer and shall return  
4270 *thrd\_success*; otherwise, *thrd\_error* shall be returned and the thread-specific storage  
4271 pointed to by *key* has an indeterminate value.

#### 4272 **ERRORS**

4273 No errors are defined.

#### 4274 **EXAMPLES**

4275 None.

#### 4276 **APPLICATION USAGE**

4277 The *tss\_create()* function performs no implicit synchronization. It is the responsibility of the  
4278 programmer to ensure that it is called exactly once per key before use of the key.

#### 4279 **RATIONALE**

4280 If the value associated with a key needs to be updated during the lifetime of the thread, it  
4281 may be necessary to release the storage associated with the old value before the new value is  
4282 bound. Although the *tss\_set()* function could do this automatically, this feature is not needed  
4283 often enough to justify the added complexity. Instead, the programmer is responsible for  
4284 freeing the stale storage:

```
4285 old = tss_get(key);  
4286 new = allocate();  
4287 destructor(old);  
4288 tss_set(key, new);
```

4289 There is no notion of a destructor-safe function. If an application does not call *thrd\_exit()* or  
4290 *pthread\_exit()* from a signal handler, or if it blocks any signal whose handler may call  
4291 *thrd\_exit()* or *pthread\_exit()* while calling async-unsafe functions, all functions can be safely  
4292 called from destructors.

4293 The *tss\_create()* function is not affected by signal handlers for the reasons stated in [xref to  
4294 XRAT B.2.3].

4295 **FUTURE DIRECTIONS**

4296 None.

4297 **SEE ALSO**

4298 *pthread\_exit*, *pthread\_key\_create*, *thr\_exit*, *tss\_delete*, *tss\_get*

4299 XBD <**threads.h**>

4300 **CHANGE HISTORY**

4301 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4302 **NAME**

4303 *tss\_delete* — thread-specific data key deletion

4304 **SYNOPSIS**

4305 `#include <threads.h>`

4306 `void tss_delete(tss_t key);`

4307 **DESCRIPTION**

4308 [CX] The functionality described on this reference page is aligned with the ISO C standard.

4309 Any conflict between the requirements described here and the ISO C standard is

4310 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4311 The *tss\_delete()* function shall release any resources used by the thread-specific storage  
4312 identified by *key*. The thread-specific data values associated with *key* need not be null at the  
4313 time *tss\_delete()* is called. It is the responsibility of the application to free any application  
4314 storage or perform any cleanup actions for data structures related to the deleted key or  
4315 associated thread-specific data in any threads; this cleanup can be done either before or after  
4316 *tss\_delete()* is called.

4317 The application shall ensure that the *tss\_delete()* function is only called with a value for *key*  
4318 that was returned by a call to *tss\_create()* before the thread commenced executing  
4319 destructors.

4320 If *tss\_delete()* is called while another thread is executing destructors, whether this will affect  
4321 the number of invocations of the destructor associated with *key* on that thread is unspecified.

4322 The *tss\_delete()* function shall be callable from within destructor functions. Calling  
4323 *tss\_delete()* shall not result in the invocation of any destructors. Any destructor function that  
4324 was associated with *key* shall no longer be called upon thread exit.

4325 Any attempt to use *key* following the call to *tss\_delete()* results in undefined behavior.

4326 [CX]The *tss\_delete()* function shall not be affected if the calling thread executes a signal  
4327 handler during the call.[/CX]

4328 **RETURN VALUE**

4329 This function shall not return a value.

4330 **ERRORS**



4331 No errors are defined.

4332 **EXAMPLES**

4333 None.

4334 **APPLICATION USAGE**

4335 None.

4336 **RATIONALE**

4337 A thread-specific data key deletion function has been included in order to allow the  
4338 resources associated with an unused thread-specific data key to be freed. Unused thread-  
4339 specific data keys can arise, among other scenarios, when a dynamically loaded module that  
4340 allocated a key is unloaded.

4341 Conforming applications are responsible for performing any cleanup actions needed for data  
4342 structures associated with the key to be deleted, including data referenced by thread-specific  
4343 data values. No such cleanup is done by *tss\_delete()*. In particular, destructor functions  
4344 are not called. See the RATIONALE for *pthread\_key\_delete()* for the reasons for this  
4345 division of responsibility.

4346 The *tss\_delete()* function is not affected by signal handlers for the reasons stated in [xref to  
4347 XRAT B.2.3].

4348 **FUTURE DIRECTIONS**

4349 None.

4350 **SEE ALSO**

4351 *pthread\_key\_create*, *tss\_create*

4352 XBD <**threads.h**>

4353 **CHANGE HISTORY**

4354 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4355 **NAME**

4356 *tss\_get*, *tss\_set* — thread-specific data management

4357 **SYNOPSIS**

4358 `#include <threads.h>`

4359 `void *tss_get(tss_t key);`  
4360 `int tss_set(tss_t key, void *val);`

4361 **DESCRIPTION**

4362 [CX] The functionality described on this reference page is aligned with the ISO C standard.  
4363 Any conflict between the requirements described here and the ISO C standard is  
4364 unintentional. This volume of POSIX.1-20xx defers to the ISO C standard.[/CX]

4365 The *tss\_get()* function shall return the value for the current thread held in the thread-specific  
4366 storage identified by *key*.

4367 The *tss\_set()* function shall set the value for the current thread held in the thread-specific

4368 storage identified by *key* to *val*. This action shall not invoke the destructor associated with  
4369 the key on the value being replaced.

4370 The application shall ensure that the *tss\_get()* and *tss\_set()* functions are only called with a  
4371 value for *key* that was returned by a call to *tss\_create()* before the thread commenced  
4372 executing destructors.

4373 The effect of calling *tss\_get()* or *tss\_set()* after *key* has been deleted with *tss\_delete()* is  
4374 undefined.

4375 [CX]Both *tss\_get()* and *tss\_set()* can be called from a thread-specific data destructor  
4376 function. A call to *tss\_get()* for the thread-specific data key being destroyed shall return a  
4377 null pointer, unless the value is changed (after the destructor starts) by a call to *tss\_set()*.  
4378 Calling *tss\_set()* from a thread-specific data destructor function may result either in lost  
4379 storage (after at least PTHREAD\_DESTRUCTOR\_ITERATIONS attempts at destruction)  
4380 or in an infinite loop.

4381 These functions shall not be affected if the calling thread executes a signal handler during  
4382 the call.[/CX]

#### 4383 **RETURN VALUE**

4384 The *tss\_get()* function shall return the value for the current thread. If no thread-specific data  
4385 value is associated with *key*, then a null pointer shall be returned.

4386 The *tss\_set()* function shall return *thrd\_success* on success or *thrd\_error* if the request  
4387 could not be honored.

#### 4388 **ERRORS**

4389 No errors are defined.

#### 4390 **EXAMPLES**

4391 None.

#### 4392 **APPLICATION USAGE**

4393 None.

#### 4394 **RATIONALE**

4395 These functions are not affected by signal handlers for the reasons stated in [xref to XRAT  
4396 B.2.3].

#### 4397 **FUTURE DIRECTIONS**

4398 None.

#### 4399 **SEE ALSO**

4400 *pthread\_getspecific*, *tss\_create*

4401 XBD <threads.h>

#### 4402 **CHANGE HISTORY**

4403 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

4404 Ref 7.31.11 para 2

4405 On page 2193 line 70145 section `ungetc()`, change FUTURE DIRECTIONS from:

4406           None.

4407 to:

4408           The ISO C standard states that the use of `ungetc()` on a binary stream where the file position  
4409           indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction  
4410           between binary and text streams, so this applies to all streams. This feature may be removed  
4411           in a future version of this standard.

4412 Ref 7.29.6.3 para 1, 7.1.4 para 5

4413 On page 2242 line 71441 section `wcrtomb()`, change:

4414           [CX]The `wcrtomb()` function need not be thread-safe if called with a NULL *ps*  
4415           argument.[/CX]

4416 to:

4417           If called with a null *ps* argument, the `wcrtomb()` function need not be thread-safe; however,  
4418           such calls shall avoid data races with calls to `wcrtomb()` with a non-null argument and with  
4419           calls to all other functions.

4420 Ref 7.29.6.4 para 1, 7.1.4 para 5

4421 On page 2266 line 72111 section `wcsrtombs()`, change:

4422           [CX]The `wcsnrtombs()` and `wcsrtombs()` functions need not be thread-safe if called with a  
4423           NULL *ps* argument.[/CX]

4424 to:

4425           [CX]If called with a null *ps* argument, the `wcsnrtombs()` function need not be thread-safe;  
4426           however, such calls shall avoid data races with calls to `wcsnrtombs()` with a non-null  
4427           argument and with calls to all other functions.[/CX]

4428           If called with a null *ps* argument, the `wcsrtombs()` function need not be thread-safe;  
4429           however, such calls shall avoid data races with calls to `wcsrtombs()` with a non-null  
4430           argument and with calls to all other functions.

4431 Ref 7.22.7 para 1, 7.1.4 para 5

4432 On page 2292 line 72879 section `wctomb()`, change:

4433           [CX]The `wctomb()` function need not be thread-safe.[/CX]

4434 to:

4435           The `wctomb()` function need not be thread-safe; however, it shall avoid data races with all  
4436           other functions.

## 4437 Changes to XCU

4438 Ref 7.22.2

4439 On page 2333 line 74167 section 1.1.2.2 Mathematical Functions, change:

4440 Section 7.20.2, Pseudo-Random Sequence Generation Functions

4441 to:

4442 Section 7.22.2, Pseudo-Random Sequence Generation Functions

4443 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4444 On page 2542 line 82220 section c99, rename the c99 page to c17.

4445 Ref 7.26

4446 On page 2545 line 82375 section c99 (now c17), change:

4447 ... , `<spawn.h>`, `<sys/socket.h>`, ...

4448 to:

4449 ... , `<spawn.h>`, `<sys/socket.h>`, `<threads.h>`, ...

4450 Ref 7.26

4451 On page 2545 line 82382 section c99 (now c17), change:

4452 This option shall make available all interfaces referenced in `<pthread.h>` and `pthread_kill()`  
4453 and `pthread_sigmask()` referenced in `<signal.h>`.

4454 to:

4455 This option shall make available all interfaces referenced in `<pthread.h>` and `<threads.h>`,  
4456 and also `pthread_kill()` and `pthread_sigmask()` referenced in `<signal.h>`.

4457 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4458 On page 2552-2553 line 82641-82677 section c99 (now c17), change CHANGE HISTORY to:

4459 First released in Issue 8. Included for alignment with the ISO/IEC 9899:20xx standard.

## 4460 Changes to XRAT

4461 Ref G.1 para 1

4462 On page 3483 line 117680 section A.1.7.1 Codes, add a new tagged paragraph:

4463 MXC This margin code is used to denote functionality related to the IEC 60559 Complex  
4464 Floating-Point option.

4465 Ref (none)

4466 On page 3489 line 117909 section A.3 Definitions (Byte), change:

4467 alignment with the ISO/IEC 9899: 1999 standard, where the **intN\_t** types are now defined.

4468 to:

4469 alignment with the ISO/IEC 9899: 1999 standard, where the **intN\_t** types were first defined.

4470 Ref 5.1.2.4, 7.17.3

4471 On page 3515 line 118946 section A.4.12 Memory Synchronization, change:

4472 **A.4.12 Memory Synchronization**

4473 to:

4474 **A.4.12 Memory Ordering and Synchronization**

4475 *A.4.12.1 Memory Ordering*

4476 There is no additional rationale provided for this section.

4477 *A.4.12.2 Memory Synchronization*

4478 Ref 6.10.8.1 para 1 (`__STDC_VERSION__`)

4479 On page 3556 line 120684 section A.12.2 Utility Syntax Guidelines, change:

4480 Thus, they had to devise a new name, *c89* (now superseded by *c99*), rather than ...

4481 to:

4482 Thus, they had to devise a new name, *c89* (subsequently superseded by *c99* and now by  
4483 *c17*), rather than ...

4484 Ref K.3.1.1

4485 On page 3567 line 121053 section B.2.2.1 POSIX.1 Symbols, add a new unnumbered subsection:

4486 **The `__STDC_WANT_LIB_EXT1__` Feature Test Macro**

4487 The ISO C standard specifies the feature test macro `__STDC_WANT_LIB_EXT1__` as the  
4488 announcement mechanism for the application that it requires functionality from Annex K. It  
4489 specifies that the symbols specified in Annex K (if supported) are made visible when  
4490 `__STDC_WANT_LIB_EXT1__` is 1 and are not made visible when it is 0, but leaves it  
4491 unspecified whether they are made visible when `__STDC_WANT_LIB_EXT1__` is  
4492 undefined. POSIX.1 requires that they are not made visible when the macro is undefined  
4493 (except for those symbols that are already explicitly allowed to be visible through the  
4494 definition of `_POSIX_C_SOURCE` or `_XOPEN_SOURCE`, or both).

4495 POSIX.1 does not include the interfaces specified in Annex K of the ISO C standard, but  
4496 allows the symbols to be made visible in headers when requested by the application in order  
4497 that applications can use symbols from Annex K and symbols from POSIX.1 in the same  
4498 translation unit.

4499 Ref 6.10.3.4

4500 On page 3570 line 121176 section B.2.2.2 The Name Space, change:

4501 as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C  
4502 standard

4503 to:

4504 as described for macros that expand to their own name as in Section 6.10.3.4 of the ISO C  
4505 standard

4506 Ref 7.5 para 2  
4507 On page 3571 line 121228-121243 section B.2.3 Error Numbers, change:

4508 The ISO C standard requires that *errno* be an assignable lvalue. Originally, ...  
4509 [...]  
4510 ... using the return value for a mixed purpose was judged to be of limited use and  
4511 error prone.

4512 to:

4513 The original ISO C standard just required that *errno* be a modifiable lvalue. Since the  
4514 introduction of threads in 2011, the ISO C standard has instead required that *errno* be a  
4515 macro which expands to a modifiable lvalue that has thread local storage duration.

4516 Ref 7.26  
4517 On page 3575 line 121390 section B.2.3 Error Numbers, change:

4518 In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a  
4519 special case since it will never occur.

4520 to:

4521 In particular, applications calling blocking interfaces need not handle any possible [EINTR]  
4522 return as a special case since it will never occur. In the case of threads functions in  
4523 <threads.h>, the requirement is stated in terms of the call not being affected if the calling  
4524 thread executes a signal handler during the call, since these functions return errors in a  
4525 different way and cannot distinguish an [EINTR] condition from other error conditions.

4526 Ref (none)  
4527 On page 3733 line 128128 section C.2.6.4 Arithmetic Expansion, change:

4528 Although the ISO/IEC 9899: 1999 standard now requires support for ...

4529 to:

4530 Although the ISO C standard requires support for ...

4531 Ref 7.17  
4532 On page 3789 line 129986 section E.1 Subprofiling Option Groups, change:

4533 by collecting sets of related functions

4534 to:

4535 by collecting sets of related functions and generic functions

4536 Ref 7.22.3.1, 7.27.2.5, 7.22.4  
4537 On page 3789, 3792 line 130022-130032, 130112-130114 section E.1 Subprofiling Option Groups,  
4538 add new functions (in sorted order) to the existing groups as indicated:

4539 POSIX\_C\_LANG\_SUPPORT  
4540 *aligned\_alloc()*, *timespec\_get()*

4541 POSIX\_MULTI\_PROCESS  
4542 *at\_quick\_exit()*, *quick\_exit()*

4543 Ref 7.17  
4544 On page 3789 line 129991 section E.1 Subprofiling Option Groups, add:

4545 POSIX\_C\_LANG\_ATOMICS: ISO C Atomic Operations  
4546 *atomic\_compare\_exchange\_strong()*, *atomic\_compare\_exchange\_strong\_explicit()*,  
4547 *atomic\_compare\_exchange\_weak()*, *atomic\_compare\_exchange\_weak\_explicit()*,  
4548 *atomic\_exchange()*, *atomic\_exchange\_explicit()*, *atomic\_fetch\_add()*,  
4549 *atomic\_fetch\_add\_explicit()*, *atomic\_fetch\_and()*, *atomic\_fetch\_and\_explicit()*,  
4550 *atomic\_fetch\_or()*, *atomic\_fetch\_or\_explicit()*, *atomic\_fetch\_sub()*,  
4551 *atomic\_fetch\_sub\_explicit()*, *atomic\_fetch\_xor()*, *atomic\_fetch\_xor\_explicit()*,  
4552 *atomic\_flag\_clear()*, *atomic\_flag\_clear\_explicit()*, *atomic\_flag\_test\_and\_set()*,  
4553 *atomic\_flag\_test\_and\_set\_explicit()*, *atomic\_init()*, *atomic\_is\_lock\_free()*,  
4554 *atomic\_load()*, *atomic\_load\_explicit()*, *atomic\_signal\_fence()*,  
4555 *atomic\_thread\_fence()*, *atomic\_store()*, *atomic\_store\_explicit()*, *kill\_dependency()*

4556 Ref 7.26  
4557 On page 3790 line 1300349 section E.1 Subprofiling Option Groups, add:

4558 POSIX\_C\_LANG\_THREADS: ISO C Threads  
4559 *call\_once()*, *cnd\_broadcast()*, *cnd\_signal()*, *cnd\_destroy()*, *cnd\_init()*,  
4560 *cnd\_timedwait()*, *cnd\_wait()*, *mtx\_destroy()*, *mtx\_init()*, *mtx\_lock()*, *mtx\_timedlock()*,  
4561 *mtx\_trylock()*, *mtx\_unlock()*, *thrd\_create()*, *thrd\_current()*, *thrd\_detach()*,  
4562 *thrd\_equal()*, *thrd\_exit()*, *thrd\_join()*, *thrd\_sleep()*, *thrd\_yield()*, *tss\_create()*,  
4563 *tss\_delete()*, *tss\_get()*, *tss\_set()*

4564 POSIX\_C\_LANG\_UCHAR: ISO C Unicode Utilities  
4565 *c16rtomb()*, *c32rtomb()*, *mbrtoc16()*, *mbrtoc32()*