TODO

Check for overlaps with Mantis bugs: 374 and 1218 (once resolved; NB 374 may also affect aligned_alloc()), and any that get tagged tc3 or issue8 after ~~2019-11-18~~2020-10-29

Introduction

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

Note that the removal of *gets*() is not included here, as it is has already ~~shaded OB and so will automatically~~ been removed by ~~default in Issue 8~~bug 1330.

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

Global Change

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

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

Changes to XBD

Ref G.1 para 1
On page 9 line 249 section 1.7.1 Codes, add a new code:

> [MXC]IEC 60559 Complex Floating-Point[/MXC]
> The functionality described is optional. The functionality described is mandated by the ISO C standard only for implementations that define __STDC_IEC_559_COMPLEX__.

Ref (none)
On page 29 line 1063, 1067 section 2.2.1 Strictly Conforming POSIX Application, change:

> the ISO/IEC 9899: 1999 standard

to:

> the ISO C standard

Ref 6.2.8
On page 34 line 1184 section 3.11 Alignment, change:

> See also the ISO C standard, Section B3.

to:

See also the ISO C standard, Section 6.2.8.

Ref 5.1.2.4
On page 38 line 1261 section 3 Definitions, add a new subsection:

3.31 Atomic Operation

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

Ref 7.26.3
On page 50 line 1581 section 3.107 Condition Variable, add a new paragraph:

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

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

Ref 5.1.2.4
On page 53 line 1635 section 3 Definitions, add a new subsection:

3.125 Data Race

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

Ref 5.1.2.4
On page 67 line 1973 section 3 Definitions, add a new subsection:

3.215 Lock-Free Operation

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

Ref 7.26.5.1
On page 70 line 2048 section 3.233 Multi-Threaded Program, change:

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

to:

the process can create additional threads using *pthread_create*(), *thrd_create*(), or SIGEV_THREAD notifications.

Ref 7.26.4
On page 70 line 2054 section 3.234 Mutex, add a new paragraph:

> There are two types of mutex: those of type **pthread_mutex_t** which are initialized using *pthread_mutex_init*() and those of type **mtx_t** which are initialized using *mtx_init*(). If an application attempts to use the two types interchangeably (that is, pass a mutex of type **pthread_mutex_t** to a function that takes a **mtx_t**, or vice versa), the behavior is undefined.
>
> **Note:** The *pthread_mutex_init*() and *mtx_init*() functions are defined in detail in the System Interfaces volume of POSIX.1-20xx.

Ref 7.26.5.5
On page 82 line 2345 section 3.303 Process Termination, change:

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

to:

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

Ref 7.26.5.1
On page 90 line 2530 section 3.354 Single-Threaded Program, change:

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

to:

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

Ref 5.1.2.4
On page 95 line 2639 section 3 Definition, add a new subsection:

> **3.382 Synchronization Operation**
>
> An operation that synchronizes memory. See [xref to XSH 4.12].

Ref 7.26.5.1
On page 99 line 2745 section 3.405 Thread ID, change:

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

to:

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

- The initial thread.
- A thread created using *pthread_create*().
- A thread created using *thrd_create*().
- A thread created via a SIGEV_THREAD notification.

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

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

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

to:

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

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

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

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

> A process global handle of type **pthread_key_t** which is used for naming thread-specific data.
>
> Although the same key value may be used by different threads, the values bound to the key by *pthread_setspecific*() and accessed by *pthread_getspecific*() are maintained on a per-thread basis and persist for the life of the calling thread.
>
> **Note:** The *pthread_getspecific*() and *pthread_setspecific*() functions are defined in detail in the System Interfaces volume of POSIX.1-2017.

to:

> A process global handle which is used for naming thread-specific data. There are two types of key: those of type **pthread_key_t** which are created using *pthread_key_create*() and those of type **tss_t** which are created using *tss_create*(). If an application attempts to use the two types of key interchangeably (that is, pass a key of type **pthread_key_t** to a function that takes a **tss_t**, or vice versa), the behavior is undefined.
>
> Although the same key value can be used by different threads, the values bound to the key by *pthread_setspecific*() for keys of type **pthread_key_t**, and by *tss_set*() for keys of type

tss_t, are maintained on a per-thread basis and persist for the life of the calling thread.

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

Ref 5.1.2.4, 7.17.3
On page 111 line 3060 section 4.12 Memory Synchronization, change:

4.12 Memory Synchronization

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

to:

4.12 Memory Ordering and Synchronization

4.12.1 Memory Ordering

4.12.1.1 Data Races

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- A happens before B, and
- there is no other side effect X to M such that A happens before X and X happens before B.

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

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

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

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

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

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

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

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

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

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

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

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

4.12.1.2 Memory Order and Consistency

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

For `memory_order_relaxed`, no operation orders memory.

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

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

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

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

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

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

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

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

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

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

- there is a `memory_order_seq_cst` fence X such that A is sequenced before X, and X precedes B in S, or
- there is a `memory_order_seq_cst` fence Y such that Y is sequenced before B, and A precedes Y in S, or
- there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S.

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

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

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

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

4.12.2 Memory Synchronization

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

Ref 7.26.3, 7.26.4
On page 111 line 3066-3075 section 4.12 Memory Synchronization, add the following to the list of

functions that synchronize memory:

cnd_broadcast()	*mtx_lock*()	*thrd_create*()
cnd_signal()	*mtx_timedlock*()	*thrd_join*()
cnd_timedwait()	*mtx_trylock*()	
cnd_wait()	*mtx_unlock*()	

Ref 7.26.2.1, 7.26.4
On page 111 line 3076 section 4.12 Memory Synchronization, change:

> The *pthread_once*() function shall synchronize memory for the first call in each thread for a given **pthread_once_t** object. If the *init_routine* called by *pthread_once*() is a cancellation point and is canceled, a call to *pthread_once*() for the same **pthread_once_t** object made from a cancellation cleanup handler shall also synchronize memory.
>
> The *pthread_mutex_lock*() function need not synchronize memory if the mutex type if PTHREAD_MUTEX_RECURSIVE and the calling thread already owns the mutex. The *pthread_mutex_unlock*() function need not synchronize memory if the mutex type is PTHREAD_MUTEX_RECURSIVE and the mutex has a lock count greater than one.

to:

> The *pthread_once*() and *call_once*() functions shall synchronize memory for the first call in each thread for a given **pthread_once_t** or **once_flag** object, respectively. If the *init_routine* called by *pthread_once*() or *call_once*() is a cancellation point and is canceled, a call to *pthread_once*() for the same **pthread_once_t** object, or to *call_once*() for the same **once_flag** object, made from a cancellation cleanup handler shall also synchronize memory.
>
> The *pthread_mutex_lock*() and *thrd_lock*() functions, and their related “timed” and “try” variants, need not synchronize memory if the mutex is a recursive mutex and the calling thread already owns the mutex. The *pthread_mutex_unlock*() and *thrd_unlock*() functions need not synchronize memory if the mutex is a recursive mutex and has a lock count greater than one.

Ref 7.12.1 para 7
On page 117 line 3319 section 4.20 Treatment of Error Conditions for Mathematical Functions, change:

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

to:

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

Ref 7.12.1 para 3
On page 117 line 3330 section 4.20.2 Pole Error, change:

> A ``pole error’’ occurs if the mathematical result of the function is an exact infinity (for

example, log(0.0)).

to:

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

Ref 7.12.1 para 4
On page 118 line 3339 section 4.20.3 Range Error, after:

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

add:

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

Ref 7.29.1 para 5
On page 129 line 3749 section 6.3 C Language Wide-Character Codes, add a new paragraph:

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

Ref 7.3.1 para 2
On page 224 line 7541 section <complex.h>, add a new paragraph:

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

Ref G.6 para 1
On page 224 line 7551 section <complex.h>, after:

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

add:

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

_Imaginary_I.[/MXC]

Ref 7.3.9.3
On page 224 line 7553 section <complex.h>, add:

The following shall be defined as macros.

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

Ref 7.3.1 para 2
On page 226 line 7623 section <complex.h>, add a new first paragraph to APPLICATION USAGE:

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

Ref 7.3.9.3
On page 226 line 7649 section <complex.h>, add CMPLX() to the SEE ALSO list before cabs().

Ref 7.5 para 2
On page 234 line 7876 section <errno.h>, change:

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

to:

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

Ref (none)
On page 245 line 8290 section <fenv.h>, change:

the ISO/IEC 9899: 1999 standard

to:

the ISO C standard

Ref 5.2.4.2.2 para 11
On page 248 line 8369 section <float.h>, add the following new paragraphs:

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

−1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

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

Ref 5.2.4.2.2 para 12
On page 248 line 8378 section <float.h>, add a new bullet item:

> Number of decimal digits, *n*, such that any floating-point number with *p* radix *b* digits can be rounded to a floating-point number with *n* decimal digits and back again without change to the value.
>
> [math stuff]
>
> FLT_DECIMAL_DIG 6
>
> DBL_DECIMAL_DIG 10
>
> LDBL_DECIMAL_DIG 10

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

Ref 5.2.4.2.2 para 14
On page 250 line 8429 section <float.h>, add a new bullet item:

> Minimum positive floating-point number.
>
> FLT_TRUE_MIN 1E-37
>
> DBL_TRUE_MIN 1E-37
>
> LDBL_TRUE_MIN 1E-37
>
> **Note:** If the presence or absence of subnormal numbers is indeterminable, then the value is intended to be a positive number no greater than the minimum normalized positive number for the type.

Ref (none)
On page 270 line 8981 section <limits.h>, change:

> the ISO/IEC 9899: 1999 standard

to:

> the ISO C standard

Ref 7.22.4.3
On page 271 line 9030 section <limits.h>, change:

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

to:

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

Ref 5.2.4.2.1 para 2
On page 280 line 9419 section <limits.h>, change:

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

to:

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

Ref (none)
On page 294 line 10016 section <math.h>, change:

the ISO/IEC 9899: 1999 standard provides for …

to:

the ISO/IEC 9899: 1999 standard provided for …

Ref 7.26.5.5
On page 317 line 10742 section <pthread.h>, change:

```
void  pthread_exit(void *);
```

to:

```
_Noreturn void  pthread_exit(void *);
```

Ref 7.13.2.1 para 1
On page 331 line 11204 section <setjmp.h>, change:

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

to:

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

Ref 7.15

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

NAME

stdalign.h — alignment macros

SYNOPSIS

```
#include <stdalign.h>
```

DESCRIPTION

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

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

alignas Expands to **_Alignas**

alignof Expands to **_Alignof**

__alignas_is_defined
Expands to the integer constant 1

__alignof_is_defined
Expands to the integer constant 1

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

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

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

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

NAME

stdatomic.h — atomics

SYNOPSIS

```
#include <stdatomic.h>
```

DESCRIPTION

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

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

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

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

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

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

atomic_uintptr_t	**_Atomic uintptr_t**
atomic_size_t	**_Atomic size_t**
atomic_ptrdiff_t	**_Atomic ptrdiff_t**
atomic_intmax_t	**_Atomic intmax_t**
atomic_uintmax_t	**_Atomic uintmax_t**

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

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
```

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

ATOMIC_BOOL_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE

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

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

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

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

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

```
_Bool      atomic_compare_exchange_strong(volatile A *, C *, C);
_Bool      atomic_compare_exchange_strong_explicit(volatile A *,
                C *, C, memory_order, memory_order);
_Bool      atomic_compare_exchange_weak(volatile A *, C *, C);
_Bool      atomic_compare_exchange_weak_explicit(volatile A *, C *,
                C, memory_order, memory_order);
C          atomic_exchange(volatile A *, C);
C          atomic_exchange_explicit(volatile A *, C, memory_order);
C          atomic_fetch_add(volatile A *, M);
C          atomic_fetch_add_explicit(volatile A *, M,
                memory_order);
C          atomic_fetch_and(volatile A *, M);
C          atomic_fetch_and_explicit(volatile A *, M,
                memory_order);
C          atomic_fetch_or(volatile A *, M);
C          atomic_fetch_or_explicit(volatile A *, M, memory_order);
C          atomic_fetch_sub(volatile A *, M);
C          atomic_fetch_sub_explicit(volatile A *, M,
                memory_order);
C          atomic_fetch_xor(volatile A *, M);
C          atomic_fetch_xor_explicit(volatile A *, M,
                memory_order);
void       atomic_init(volatile A *, C);
_Bool      atomic_is_lock_free(const volatile A *);
C          atomic_load(const volatile A *);
C          atomic_load_explicit(const volatile A *, memory_order);
void       atomic_store(volatile A *, C);
void       atomic_store_explicit(volatile A *, C, memory_order);
```

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

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

```
void       atomic_flag_clear(volatile atomic_flag *);
void       atomic_flag_clear_explicit(volatile atomic_flag *,
                memory_order);
_Bool      atomic_flag_test_and_set(volatile atomic_flag *);
_Bool      atomic_flag_test_and_set_explicit(
                volatile atomic_flag *, memory_order);
void       atomic_signal_fence(memory_order);
void       atomic_thread_fence(memory_order);
```

APPLICATION USAGE

None.

RATIONALE

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

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

FUTURE DIRECTIONS

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

SEE ALSO

Section 4.12.1

XSH *atomic_compare_exchange_strong*(), *atomic_compare_exchange_weak*(), *atomic_exchange*(), *atomic_fetch_**key***(), *atomic_flag_clear*(), *atomic_flag_test_and_set*(), *atomic_init*(), *atomic_is_lock_free*(), *atomic_load*(), *atomic_signal_fence*(), *atomic_store*(), *atomic_thread_fence*(), *kill_dependency*().

CHANGE HISTORY

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

Ref 7.31.9
On page 345 line 11747 section <stdbool.h>, add OB shading to:

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

Ref 7.19 para 2
On page 346 line 11774 section <stddef.h>, add:

max_align_t Object type whose alignment is the greatest fundamental alignment.

Ref (none)
On page 348 line 11834 section <stdint.h>, change:

the ISO/IEC 9899: 1999 standard

to:

the ISO C standard

Ref 7.20.1.1 para 1
On page 348 line 11841 section <stdint.h>, change:

denotes a signed integer type

to:

denotes such a signed integer type

Ref 7.20.1.1 para 2
On page 348 line 11843 section <stdint.h>, change:

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

to:

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

Ref 7.21.1 para 2
On page 355 line 12064 section <stdio.h>, change:

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

to:

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

Ref 7.21.1 para 3
On page 357 line 12186 section <stdio.h>, change RATIONALE from:

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

to:

None.

Ref 7.22.4.5 para 1
On page 359 line 12267 section <stdlib.h>, change:

```
void            _Exit(int);
```

to:

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

Ref 7.22.4.1 para 1
On page 359 line 12269 section <stdlib.h>, change:

```
void            abort(void);
```

to:

```
_Noreturn void  abort(void);
```

Ref 7.22.3.1, 7.22.4.3
On page 359 line 12270 section <stdlib.h>, add:

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

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

```
void            exit(int);
```

to:

```
_Noreturn void  exit(int);
```

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

```
_Noreturn void  quick_exit(int);
```

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

NAME

stdnoreturn.h — noreturn macro

SYNOPSIS

```
#include <stdnoreturn.h>
```

DESCRIPTION

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

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

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

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

Ref G.7
On page 422 line 14340 section <tgmath.h>, add two new paragraphs:

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

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

cos(*iy*)	=	*cosh*(*y*)
sin(*iy*)	=	*i sinh*(*y*)
tan(*iy*)	=	*i tanh*(*y*)
cosh(*iy*)	=	*cos*(*y*)
sinh(*iy*)	=	*i sin*(*y*)
tanh(*iy*)	=	*i tan*(*y*)
asin(*iy*)	=	*i asinh*(*y*)
atan(*iy*)	=	*i atanh*(*y*)
asinh(*iy*)	=	*i asin*(*y*)
atanh(*iy*)	=	*i atan*(*y*)

[/MXC]

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

the ISO/IEC 9899: 1999 standard

to:

the ISO C standard

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

NAME

threads.h — ISO C threads

SYNOPSIS

```
#include <threads.h>
```

DESCRIPTION

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

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

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

thread_local ~~which shall~~ ~~e~~Expands to **_Thread_local**~~,~~.

ONCE_FLAG_INIT~~ which shall~~ ~~e~~Expands to a value that can be used to initialize an object of type **once_flag**~~, and~~.

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

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

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

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

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

```
void            call_once(once_flag *, void (*)(void));
int             cnd_broadcast(cnd_t *);
void            cnd_destroy(cnd_t *);
int             cnd_init(cnd_t *);
int             cnd_signal(cnd_t *);
int             cnd_timedwait(cnd_t * restrict, mtx_t * restrict,
                    const struct timespec * restrict);
int             cnd_wait(cnd_t *, mtx_t *);
void            mtx_destroy(mtx_t *);
int             mtx_init(mtx_t *, int);
int             mtx_lock(mtx_t *);
int             mtx_timedlock(mtx_t * restrict,
                    const struct timespec * restrict);
int             mtx_trylock(mtx_t *);
int             mtx_unlock(mtx_t *);
int             thrd_create(thrd_t *, thrd_start_t, void *);
thrd_t          thrd_current(void);
int             thrd_detach(thrd_t);
int             thrd_equal(thrd_t, thrd_t);
_Noreturn void  thrd_exit(int);
int             thrd_join(thrd_t, int *);
int             thrd_sleep(const struct timespec *,
                    struct timespec *);
void            thrd_yield(void);
int             tss_create(tss_t *, tss_dtor_t);
void            tss_delete(tss_t);
void           *tss_get(tss_t);
```

```
int             tss_set(tss_t, void *);
```

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

APPLICATION USAGE

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

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

RATIONALE

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

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

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

FUTURE DIRECTIONS

None.

SEE ALSO

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

XSH Section 2.9, *call_once*(), *cnd_broadcast*(), *cnd_destroy*(), *cnd_timedwait*(), *mtx_destroy*(), *mtx_lock*(), *sysconf*(), *thrd_create*(), *thrd_current*(), *thrd_detach*(), *thrd_equal*(), *thrd_exit*(), *thrd_join*(), *thrd_sleep*(), *thrd_yield*(), *tss_create*(), *tss_delete*(), *tss_get*().

CHANGE HISTORY

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

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

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

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

and change the members to:

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

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

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

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

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

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

NAME

uchar.h — Unicode character handling

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

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

```
size_t     c16rtomb(char *restrict, char16_t,
                mbstate_t *restrict);
size_t     c32rtomb(char *restrict, char32_t,
                mbstate_t *restrict);
size_t     mbrtoc16(char16_t *restrict, const char *restrict,
                size_t, mbstate_t *restrict);
size_t     mbrtoc32(char32_t *restrict, const char *restrict,
                size_t, mbstate_t *restrict);
```

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

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

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

CHANGE HISTORY

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

Ref 7.22.4.5 para 1
On page 447 line 15388 section <unistd.h>, change:

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

to:

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

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

mbstate_t An object type other than an array type …

to:

mbstate_t A complete object type other than an array type …

Changes to XSH

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

> 6. Functions shall prevent data races as follows: A function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s arguments. A function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s non-const arguments. Implementations may share their own internal objects between threads if the objects are not visible to applications and are protected against data races.
>
> 7. Functions shall perform all operations solely within the current thread if those operations have effects that are visible to applications.

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

> 2.2.1.3 *The __STDC_WANT_LIB_EXT1__ Feature Test Macro*
>
> A POSIX-conforming [XSI]or XSI-conforming[/XSI] application can define the feature test macro __STDC_WANT_LIB_EXT1__ before inclusion of any header.
>
> When an application includes a header described by POSIX.1-20xx, and when this feature test macro is defined to have the value 1, the header may make visible those symbols specified for the header in Annex K of the ISO C standard that are not already explicitly permitted by POSIX.1-20xx to be made visible in the header. These symbols are listed in [xref to 2.2.2].
>
> When an application includes a header described by POSIX.1-20xx, and when this feature test macro is either undefined or defined to have the value 0, the header shall not make any additional symbols visible that are not already made visible by the feature test macro _POSIX_C_SOURCE [XSI]or _XOPEN_SOURCE[/XSI] as described above, except when enabled by another feature test macro.

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

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

Ref 7.31.15 para 1
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]		

Ref 7.31.8 para 1

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

<stdatomic.h>	ATOMIC_[A-Z]

Ref 7.31.14 para 1

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

<time.h>	TIME_[A-Z]

Ref K.3.4 - K.3.9

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

When the feature test macro__STDC_WANT_LIB_EXT1__ is defined with the value 1 (see [xref to 2.2.1]), implementations may add symbols to the headers shown in the following table provided the identifiers for those symbols have one of the corresponding 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, wcrtomb_s, wmemcpy_s, wmemmove_s, wprintf_s, wscanf_s

When the feature test macro__STDC_WANT_LIB_EXT1__ is defined with the value 1 (see [xref to 2.2.1]), if any header in the following table is included, macros with the complete names shown may be defined.

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

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

Ref 7.1.3 para 1

On page 478 line 16438 section 2.2.2, change:

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

to:

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

Ref 7.1.3 para 1
On page 478 line 16448 section 2.2.2, change:

> that have external linkage are always reserved

to:

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

Ref 7.1.3 para 1
On page 479 line 16453 section 2.2.2, add the following in the appropriate place in the list:

aligned_alloc
at_quick_exit
atomic_compare_exchange_strong
atomic_compare_exchange_strong_explicit
atomic_compare_exchange_weak
atomic_compare_exchange_weak_explicit
atomic_exchange
atomic_exchange_explicit
atomic_fetch_add
atomic_fetch_add_explicit
atomic_fetch_and
atomic_fetch_and_explicit
atomic_fetch_or
atomic_fetch_or_explicit
atomic_fetch_sub
atomic_fetch_sub_explicit
atomic_fetch_xor
atomic_fetch_xor_explicit
atomic_flag_clear
atomic_flag_clear_explicit
atomic_flag_test_and_set
atomic_flag_test_and_set_explicit
atomic_init
atomic_is_lock_free
atomic_load
atomic_load_explicit
atomic_signal_fence
atomic_store
atomic_store_explicit
atomic_thread_fence
c16rtomb
c32rtomb
call_once
cnd_broadcast
cnd_destroy
cnd_init
cnd_signal
cnd_timedwait
cnd_wait
kill_dependency
mbrtoc16
mbrtoc32
mtx_destroy
mtx_init
mtx_lock
mtx_timedlock
mtx_trylock
mtx_unlock
quick_exit
thrd_create
thrd_current
thrd_detach
thrd_equal
thrd_exit
thrd_join
thrd_sleep
thrd_yield
timespec_get
tss_create
tss_delete
tss_get
tss_set

Ref 7.1.2 para 4
On page 480 line 16551 section 2.2.2, change:

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

to:

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

Ref 7.26.5.1
On page 490 line 16980 section 2.4.2 Realtime Signal Generation and Delivery, change:

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

to:

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

Ref 7.14.1.1 para 5
On page 493 line 17088 section 2.4.3 Signal Actions, change:

> with static storage duration

to:

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

Ref 7.14.1.1 para 5
On page 493 line 17090 section 2.4.3 Signal Actions, after applying bug 711 change:

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

to:

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

Ref 7.14.1.1 para 5
On page 494 line 17133 section 2.4.3 Signal Actions, add *quick_exit*() to the table of async-signal-safe functions.

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

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

to:

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

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

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

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

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

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

to:

The definition of a stream includes an “orientation”.

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

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

to:

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

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

Ref 7.12.8.3, 7.22.4.8
On page 513 line 17907-17927 section 2.9.1 Thread-Safety, delete the following from the list of functions that need not be thread-safe:

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

Note to reviewers: deletion of mblen(), mbtowc(), and wctomb() from this list is the subject of Mantis bug 708.

Ref 7.28.1 para 1
On page 513 line 17928 section 2.9.1 Thread-Safety, change:

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

to:

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

Ref 7.1.4 para 5
On page 513 line 17934 section 2.9.1 Thread-Safety, change:

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

to:

> Some functions that are not required to be thread-safe are nevertheless required to avoid data races with either all or some other functions, as specified on their individual reference pages.
>
> Implementations shall provide internal synchronization as necessary in order to satisfy thread-safety requirements.

Ref 7.26.5
On page 513 line 17944 section 2.9.2 Thread IDs, change:

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

to:

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

Ref 7.26.5
On page 514 line 17950 section 2.9.2 Thread IDs, change:

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

to:

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

Ref 7.26.4
On page 514 line 17956 section 2.9.3 Thread Mutexes, change:

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

to:

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

Ref 7.26.3, 7.26.4
On page 514 line 17972 section 2.9.3 Thread Mutexes, add two new paragraphs and lists:

> A thread shall become the owner of a mutex, *m*, of type **mtx_t** when one of the following occurs:
>
> - It calls *mtx_lock*() with *m* as the *mtx* argument and the call returns `thrd_success`.
> - It calls *mtx_trylock*() with *m* as the *mtx* argument and the call returns `thrd_success`.
> - It calls *mtx_timedlock*() with *m* as the *mtx* argument and the call returns `thrd_success`.
> - It calls *cnd_wait*() with *m* as the *mtx* argument and the call returns `thrd_success`.
> - It calls *cnd_timedwait*() with *m* as the *mtx* argument and the call returns `thrd_success` or `thrd_timedout`.
>
> The thread shall remain the owner of *m* until one of the following occurs:
>
> - It executes *mtx_unlock*() with *m* as the *mtx* argument.
> - It blocks in a call to *cnd_wait*() with *m* as the *mtx* argument.
> - It blocks in a call to *cnd_timedwait*() with *m* as the *mtx* argument.

Ref 7.26.4
On page 514 line 17980 section 2.9.3 Thread Mutexes, change:

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

to:

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

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

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

to:

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

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

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

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

> Each thread maintains a list of cancellation cleanup handlers.

to:

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

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

> as described for *pthread_key_create*()

to:

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

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

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

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

> **pthread_cond_t**

to

> **pthread_cond_t**, **cnd_t**

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

> **pthread_key_t**
> **pthread_mutex_t**

to

pthread_key_t, **tss_t**
pthread_mutex_t, mtx_t

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

pthread_once_t

to

pthread_once_t, **once_flag**

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

pthread_t

to

pthread_t, thrd_t

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

NAME

CMPLX — make a complex value

SYNOPSIS

```
#include <complex.h>

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

DESCRIPTION

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

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

RETURN VALUE

The CMPLX macros return the complex value *x* + *i* *y* (where *i* is the imaginary unit).

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

```
#define CMPLX(x, y) ((double complex)((double)(x) + \
                     _Imaginary_I * (double)(y)))
#define CMPLXF(x, y) ((float complex)((float)(x) + \
                     _Imaginary_I * (float)(y)))
#define CMPLXL(x, y) ((long double complex)((long double)(x) + \
                     _Imaginary_I * (long double)(y)))
```

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

XBD **<complex.h>**

CHANGE HISTORY

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

Ref 7.22.4.5 para 1
On page 553 line 19384 section _Exit(), change:

```
void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

to:

```
_Noreturn void _Exit(int status);

#include <unistd.h>

_Noreturn void _exit(int status);
```

Ref 7.22.4.5 para 2
On page 553 line 19396 section _Exit(), change:

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

to:

shall not call functions registered with *atexit*() nor *at_quick_exit*(), nor any registered signal handlers

Ref (none)
On page 557 line 19562 section _Exit(), change:

> The ISO/IEC 9899: 1999 standard adds the *_Exit*() function

to:

> The ISO/IEC 9899: 1999 standard added the *_Exit*() function

Ref 7.22.4.3, 7.22.4.7
On page 557 line 19568 section _Exit(), add *at_quick_exit* and *quick_exit* to the SEE ALSO section.

Ref 7.22.4.1 para 1
On page 565 line 19761 section abort(), change:

```
void abort(void);
```

to:

```
_Noreturn void abort(void);
```

Ref (none)
On page 565 line 19785 section abort(), change:

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

to:

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

Ref 7.22.3.1
On page 597 line 20771 insert the following new aligned_alloc() section:

NAME

aligned_alloc — allocate memory with a specified alignment

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

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

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

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

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

RETURN VALUE

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

- A null pointer shall be returned [CX]and *errno* may be set to an implementation-defined value,[/CX] or

- A pointer to the allocated space shall be returned. The application shall ensure that the pointer is not used to access an object.

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

ERRORS

The *aligned_alloc*() function shall fail if:

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

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

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

XBD **<stdlib.h>**

CHANGE HISTORY

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

Ref 7.27.3, 7.1.4 para 5
On page 600 line 20911 section asctime(), change:

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

to:

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

Ref 7.22.4.3
On page 618 line 21380 insert the following new at_quick_exit() section:

NAME

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

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

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

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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

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

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

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

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

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

XBD **<stdlib.h>**

CHANGE HISTORY

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

Ref 7.22.4.3
On page 618 line 21381 section atexit(), change:

atexit — register a function to run at process termination

to:

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

Ref 7.22.4.2 para 2, 7.22.4.3
On page 618 line 21389 section atexit(), change:

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

to:

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

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

Ref 7.22.4.2 para 2
On page 618 line 21405 section atexit(), insert a new first APPLICATION USAGE paragraph:

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

Ref 7.22.4.3
On page 618 line 21410 section atexit(), change:

> Since the behavior is undefined if the *exit*() function is called more than once, portable applications calling *atexit*() must ensure that the *exit*() function is not called at normal process termination when all functions registered by the *atexit*() function are called.
>
> All functions registered by the *atexit*() function are called at normal process termination, which occurs by a call to the *exit*() function or a return from *main*() or on the last thread termination, when the behavior is as if the implementation called *exit*() with a zero argument at thread termination time.
>
> If, at normal process termination, a function registered by the *atexit*() function is called and a portable application needs to stop further *exit*() processing, it must call the *_exit*() function or the *_Exit*() function or one of the functions which cause abnormal process termination.

to:

> Since the behavior is undefined if the *exit*() function is called more than once, portable applications calling *atexit*() must ensure that the *exit*() function is not called when the functions registered by the *atexit*() function are called.
>
> If a function registered by the *atexit*( ) function is called and a portable application needs to stop further *exit*() processing, it must call the *_exit*() function or the *_Exit*() function or one of the functions which cause abnormal process termination.

Ref 7.22.4.3
On page 619 line 21425 section atexit(), add *at_quick_exit* to the SEE ALSO section.

Ref 7.16
On page 624 line 21548 insert the following new atomic_*() sections:

NAME

> atomic_compare_exchange_strong, atomic_compare_exchange_strong_explicit, atomic_compare_exchange_weak, atomic_compare_exchange_weak_explicit — atomically

compare and exchange the values of two objects

SYNOPSIS

```
#include <stdatomic.h>
_Bool atomic_compare_exchange_strong(volatile A *object,
     C *expected, C desired);
_Bool atomic_compare_exchange_strong_explicit(volatile A *object,
     C *expected, C desired, memory_order success,
     memory_order failure);
_Bool atomic_compare_exchange_weak(volatile A *object,
     C *expected, C desired);
_Bool atomic_compare_exchange_weak_explicit(volatile A *object,
     C *expected, C desired, memory_order success,
     memory_order failure);
```

DESCRIPTION

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

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

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

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

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

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

RETURN VALUE

These generic functions shall return the result of the comparison.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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

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

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

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

NAME

atomic_exchange, atomic_exchange_explicit — atomically exchange the value of an object

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

NAME

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

SYNOPSIS

```
#include <stdatomic.h>
C     atomic_fetch_add(volatile A *object, M operand);
C     atomic_fetch_add_explicit(volatile A *object, M operand,
           memory_order order);
C     atomic_fetch_and(volatile A *object, M operand);
C     atomic_fetch_and_explicit(volatile A *object, M operand,
           memory_order order);
C     atomic_fetch_or(volatile A *object, M operand);
C     atomic_fetch_or_explicit(volatile A *object, M operand,
           memory_order order);
C     atomic_fetch_sub(volatile A *object, M operand);
C     atomic_fetch_sub_explicit(volatile A *object, M operand,
           memory_order order);
C     atomic_fetch_xor(volatile A *object, M operand);
C     atomic_fetch_xor_explicit(volatile A *object, M operand,
           memory_order order);
```

DESCRIPTION

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

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

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

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

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

sub	subtraction
or	bitwise inclusive OR
xor	bitwise exclusive OR
and	bitwise AND

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

NAME

atomic_flag_clear, atomic_flag_clear_explicit — clear an atomic flag

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURN VALUE

These functions shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

NAME

atomic_flag_test_and_set, atomic_flag_test_and_set_explicit — test and set an atomic flag

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

NAME

atomic_init — initialize an atomic object

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

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

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

XBD **<stdatomic.h>**

CHANGE HISTORY

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

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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

FUTURE DIRECTIONS

None.

SEE ALSO

XBD **<stdatomic.h>**

CHANGE HISTORY

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

NAME

atomic_load, atomic_load_explicit — atomically obtain the value of an object

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

NAME

atomic_signal_fence, atomic_thread_fence — fence operations

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

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

- have no effects, if *order* is equal to `memory_order_relaxed`;
- be an acquire fence, if *order* is equal to `memory_order_acquire` or `memory_order_consume`;
- be a release fence, if *order* is equal to `memory_order_release`;
- be both an acquire fence and a release fence, if *order* is equal to `memory_order_acq_rel`;
- be a sequentially consistent acquire and release fence, if *order* is equal to `memory_order_seq_cst`.

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

RETURN VALUE

These functions shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

NAME

atomic_store, atomic_store_explicit — atomically store a value in an object

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURN VALUE

These generic functions shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

Ref 7.28.1, 7.1.4 para 5
On page 633 line 21891 insert a new c16rtomb() section:

NAME

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

SYNOPSIS

```
#include <uchar.h>

size_t c16rtomb(char *restrict s, char16_t c16,
           mbstate_t *restrict ps);
size_t c32rtomb(char *restrict s, char32_t c32,
           mbstate_t *restrict ps);
```

DESCRIPTION

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

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

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

where *buf* is an internal buffer.

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

the resulting state described shall be the initial conversion state.

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

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

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

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

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

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

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

RETURN VALUE

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

ERRORS

These function shall fail if:

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

These functions may fail if:

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

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbrtoc16

XBD **<uchar.h>**

CHANGE HISTORY

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

Ref G.6 para 6, F.10.4.3, F.10.4.2, F.10 para 11

On page 633 line 21905 section cabs(), add:

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

If z is ±0 ± i0, +0 shall be returned.

If the real or imaginary part of z is ±Inf, +Inf shall be returned, even if the other part is NaN.

If the real or imaginary part of z is NaN and the other part is not ±Inf, NaN shall be returned.
[/MXC]

Ref G.6.1.1

On page 634 line 21935 section cacos(), add:

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

If z is ±0 + i0, π/2 − i0 shall be returned.

If z is ±0 + iNaN, π/2 + iNaN shall be returned.

If z is x + iInf where x is finite, π/2 − iInf shall be returned.

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

If z is −Inf + iy where y is positive-signed and finite, π − iInf shall be returned.

If z is +Inf + iy where y is positive-signed and finite, +0 − iInf shall be returned.

If z is −Inf + iInf, 3π/4 − iInf shall be returned.

If z is +Inf + iInf, π/4 − iInf shall be returned.

If z is ±Inf + iNaN, NaN ± iInf shall be returned; the sign of the imaginary part of the result is unspecified.

If z is NaN + iy where y is finite, NaN + iNaN shall be returned and the invalid floating-point exception may be raised.

If z is NaN + iInf, NaN − iInf shall be returned.

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

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

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

If z is ±0 + i0, +0 +iπ/2 shall be returned.

If z is x + iInf where x is finite, +Inf +iπ/2 shall be returned.

If z is 0 + iNaN, NaN ± iπ/2 shall be returned; the sign of the imaginary part of the result is unspecified.

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

If z is −Inf + iy where y is positive-signed and finite, +Inf +iπ shall be returned.

If z is +Inf + iy where y is positive-signed and finite, +Inf + i0 shall be returned.

If z is −Inf + iInf, +Inf + i3π/4 shall be returned.

If z is +Inf + iInf, +Inf + iπ/4 shall be returned.

If z is ±Inf + iNaN, +Inf + iNaN shall be returned.

If z is NaN + iy where y is finite, NaN + iNaN shall be returned and the invalid floating-point exception may be raised.

If z is NaN + iInf, +Inf + iNaN shall be returned.

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

Ref 7.26.2.1
On page 637 line 21989 insert the following new call_once() section:

NAME

call_once — dynamic package initialization

SYNOPSIS

```
#include <threads.h>

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

DESCRIPTION

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

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

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

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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

RATIONALE

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

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


int random_function()
{
    call_once(&random_is_initialized, initialize_random);
    ...
    /* Operations performed after initialization. */
}
```

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_once

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

CHANGE HISTORY

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

Ref 7.22.3 para 1
On page 637 line 22002 section calloc(), change:

> a pointer to any type of object

to:

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

Ref 7.22.3 para 1
On page 637 line 22007 section calloc(), change:

> either a null pointer shall be returned, or …

to:

> either a null pointer shall be returned to indicate an error, or …

Ref 7.22.3 para 2
On page 637 line 22008 section calloc(), add a new paragraph:

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

Ref 7.22.3.1
On page 637 line 22029 section calloc(), add *aligned_alloc* to the SEE ALSO section.

Ref G.6 para 6, F.10.1.4, F.10 para 11
On page 639 line 22055 section carg(), add:

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

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

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

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

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

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

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

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

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

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

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

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

Ref G.6 para 7, G.6.2.2
On page 640 line 22086 section casin(), add:

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

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

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

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

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

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

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

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

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

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 result is unspecified.

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

Ref G.6 para 7

On page 640 line 22094 section casin(), change RATIONALE from:

> None.

to:

> The MXC special cases for *casin*() are derived from those for *casinh*() by applying the formula *casin*(z) = $-i$ *casinh*(iz).

Ref G.6.2.2

On page 641 line 22118 section casinh(), add:

> [MXC]*casinh*(*conj*(z)), *casinhf*(*conjf*(z)) and *casinhl*(*conjl*(z)) shall return exactly the same value as *conj*(*casinh*(z)), *conjf*(*casinhf*(z)) and *conjl*(*casinhl*(z)), respectively, and *casinh*($-z$), *casinhf*($-z$) and *casinhl*($-z$) shall return exactly the same value as −*casinh*(z), −*casinhf*(z) and −*casinhl*(z), respectively, including for the special values of z below.
>
> If z is +0 + i0, 0 + i0 shall be returned.
>
> If z is x + iInf where x is positive-signed and finite, +Inf + $i\pi/2$ shall be returned.
>
> If z is x + iNaN where x is finite, NaN + iNaN shall be returned and the invalid floating-point exception may be raised.
>
> If z is +Inf + iy where y is positive-signed and finite, +Inf + i0 shall be returned.
>
> If z is +Inf + iInf, +Inf + $i\pi/4$ shall be returned.
>
> If z is +Inf + iNaN, +Inf + iNaN shall be returned.
>
> If z is NaN + i0, NaN + i0 shall be returned.
>
> If z is NaN + iy where y is non-zero and finite, NaN + iNaN shall be returned and the invalid floating-point exception may be raised.
>
> If z is NaN + iInf, ±Inf + iNaN shall be returned; the sign of the real part of the result is unspecified.
>
> If z is NaN + iNaN, NaN + iNaN shall be returned.[/MXC]

Ref G.6 para 7, G.6.2.3

On page 643 line 22157 section catan, add:

> [MXC]*catan*(*conj*(iz)), *catanf*(*conjf*(iz)) and *catanl*(*conjl*(iz)) shall return exactly the same value as *conj*(*catan*(iz)), *conjf*(*catanf*(iz)) and *conjl*(*catanl*(iz)), respectively, and *catan*($-iz$), *catanf*($-iz$) and *catanl*($-iz$) shall return exactly the same value as −*catan*(iz), −*catanf*(iz) and −*catanl*(iz), respectively, including for the special values of iz below.
>
> If iz is +0 + i0, $-i$ (+0 + i0) shall be returned.

If *iz* is +0 + *i*NaN, −*i* (+0 + *i*NaN) shall be returned.

If *iz* is +1 + *i*0, −*i* (+Inf + *i*0) shall be returned and the divide-by-zero floating-point exception shall be raised.

If *iz* is *x* + *i*Inf where *x* is positive-signed and finite, −*i* (+0 + *i*π/2) shall be returned.

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

If *iz* is +Inf + *i*y where *y* is positive-signed and finite, −*i* (+0 + *i*π/2) shall be returned.

If *iz* is +Inf + *i*Inf, −*i* (+0 + *i*π/2) shall be returned.

If *iz* is +Inf + *i*NaN, −*i* (+0 + *i*NaN) shall be returned.

If *iz* is NaN + *i*y where *y* is finite, −*i* (NaN + *i*NaN) shall be returned and the invalid floating-point exception may be raised.

If *iz* is NaN + *i*Inf, −*i* (±0 + *i*π/2) shall be returned; the sign of the imaginary part of the result is unspecified.

If *iz* is NaN + *i*NaN, −*i* (NaN + *i*NaN) shall be returned.[/MXC]

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

None.

to:

The MXC special cases for *catan*() are derived from those for *catanh*() by applying the formula *catan*(*z*) = −*i* *catanh*(*iz*).

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

[MXC]*catanh*(*conj*(*z*)), *catanhf*(*conjf*(*z*)) and *catanhl*(*conjl*(*z*)) shall return exactly the same value as *conj*(*catanh*(*z*)), *conjf*(*catanhf*(*z*)) and *conjl*(*catanhl*(*z*)), respectively, and *catanh*(−*z*), *catanhf*(−*z*) and *catanhl*(−*z*) shall return exactly the same value as −*catanh*(*z*), −*catanhf*(*z*) and −*catanhl*(*z*), respectively, including for the special values of *z* below.

If *z* is +0 + *i*0, +0 + *i*0 shall be returned.

If *z* is +0 + *i*NaN, +0 + *i*NaN shall be returned.

If *z* is +1 + *i*0, +Inf + *i*0 shall be returned and the divide-by-zero floating-point exception shall be raised.

If *z* is *x* + *i*Inf where *x* is positive-signed and finite, +0 + *i*π/2 shall be returned.

If *z* is *x* + *i*NaN where *x* is non-zero and finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

If *z* is +Inf + *i*y where *y* is positive-signed and finite, +0 + *i*π/2 shall be returned.

If *z* is +Inf + *i*Inf, +0 + *i*π/2 shall be returned.

If *z* is +Inf + *i*NaN, +0 + *i*NaN shall be returned.

If *z* is NaN + *i*y where *y* is finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

If *z* is NaN + *i*Inf, ±0 + *i*π/2 shall be returned; the sign of the real part of the result is unspecified.

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

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

[MXC]*ccos*(*conj*(*iz*)), *ccosf*(*conjf*(*iz*)) and *ccosl*(*conjl*(*iz*)) shall return exactly the same value as *conj*(*ccos*(*iz*)), *conjf*(*ccosf*(*iz*)) and *conjl*(*ccosl*(*iz*)), respectively, and *ccos*(−*iz*), *ccosf*(−*iz*) and *ccosl*(−*iz*) shall return exactly the same value as *ccos*(*iz*), *ccosf*(*iz*) and *ccosl*(*iz*), respectively, including for the special values of *iz* below.

If *iz* is +0 + *i*0, 1 + *i*0 shall be returned.

If *iz* is +0 + *i*Inf, NaN ± *i*0 shall be returned and the invalid floating-point exception shall be raised; the sign of the imaginary part of the result is unspecified.

If *iz* is +0 + *i*NaN, NaN ± *i*0 shall be returned; the sign of the imaginary part of the result is unspecified.

If *iz* is *x* + *i*Inf where *x* is non-zero and finite, NaN + *i*NaN shall be returned and the invalid floating-point exception shall be raised.

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

If *iz* is +Inf + *i*0, +Inf + *i*0 shall be returned.

If *iz* is +Inf + *iy* where *y* is non-zero and finite, +Inf (cos(*y*) + *i*sin(*y*)) shall be returned.

If *iz* is +Inf + *i*Inf, ±Inf + *i*NaN shall be returned and the invalid floating-point exception shall be raised; the sign of the real part of the result is unspecified.

If *iz* is +Inf + *i*NaN, +Inf + *i*NaN shall be returned.

If *iz* is NaN + *i*0, NaN ± *i*0 shall be returned; the sign of the imaginary part of the result is unspecified.

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

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

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

None.

to:

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

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

[MXC]*ccosh*(*conj*(*z*)), *ccoshf*(*conjf*(*z*)) and *ccoshl*(*conjl*(*z*)) shall return exactly the same value as *conj*(*ccosh*(*z*)), *conjf*(*ccoshf*(*z*)) and *conjl*(*ccoshl*(*z*)), respectively, and *ccosh*(−*z*), *ccoshf*(−*z*) and *ccoshl*(−*z*) shall return exactly the same value as *ccosh*(*z*), *ccoshf*(*z*) and *ccoshl*(*z*), respectively, including for the special values of *z* below.

If *z* is +0 + *i*0, 1 + *i*0 shall be returned.

If *z* is +0 + *i*Inf, NaN ± *i*0 shall be returned and the invalid floating-point exception shall be raised; the sign of the imaginary part of the result is unspecified.

If *z* is +0 + *i*NaN, NaN ± *i*0 shall be returned; the sign of the imaginary part of the result is unspecified.

If *z* is *x* + *i*Inf where *x* is non-zero and finite, NaN + *i*NaN shall be returned and the invalid floating-point exception shall be raised.

If *z* is *x* + *i*NaN where *x* is non-zero and finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

If *z* is +Inf + *i*0, +Inf + *i*0 shall be returned.

If *z* is +Inf + *iy* where *y* is non-zero and finite, +Inf (cos(*y*) + *i*sin(*y*)) shall be returned.

If *z* is +Inf + *i*Inf, ±Inf + *i*NaN shall be returned and the invalid floating-point exception shall be raised; the sign of the real part of the result is unspecified.

If *z* is +Inf + *i*NaN, +Inf + *i*NaN shall be returned.

If *z* is NaN + *i*0, NaN ± *i*0 shall be returned; the sign of the imaginary part of the result is unspecified.

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

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

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

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

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

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

to:

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

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

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

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

> [MXC]*cexp*(*conj*(z)), *cexpf*(*conjf*(z)) and *cexpl*(*conjl*(z)) shall return exactly the same value as *conj*(*cexp*(z)), *conjf*(*cexpf*(z)) and *conjl*(*cexpl*(z)), respectively, including for the special values of *z* below.
>
> If *z* is ±0 + *i*0, 1 + *i*0 shall be returned.
>
> If *z* is *x* + *i*Inf where *x* is finite, NaN + *i*NaN shall be returned and the invalid floating-point exception shall be raised.
>
> If *z* is *x* + *i*NaN where *x* is finite, NaN + iNaN shall be returned and the invalid floating-point exception may be raised.
>
> If *z* is +Inf + *i*0, +Inf + *i*0 shall be returned.
>
> If *z* is −Inf + *iy* where *y* is finite, +0 (cos(*y*) + *i*sin(*y*)) shall be returned.
>
> If *z* is +Inf + *iy* where *y* is non-zero and finite, +Inf (cos(*y*) + *i*sin(*y*)) shall be returned.
>
> If *z* is −Inf + *i*Inf, ±0 ± *i*0 shall be returned; the signs of the real and imaginary parts of the result are unspecified.
>
> If *z* is +Inf + *i*Inf, ±Inf + *i*NaN shall be returned and the invalid floating-point exception shall be raised; the sign of the real part of the result is unspecified.
>
> If *z* is −Inf + *i*NaN, ±0 ± *i*0 shall be returned; the signs of the real and imaginary parts of the result are unspecified.

If *z* is +Inf + *i*NaN, ±Inf + *i*NaN shall be returned; the sign of the real part of the result is unspecified.

If *z* is NaN + *i*0, NaN + *i*0 shall be returned.

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

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

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

including the *nanosleep*() function

to:

including the *nanosleep*() and *thrd_sleep*() functions

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

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

If *z* is −0 + *i*0, −Inf + *i*π shall be returned and the divide-by-zero floating-point exception shall be raised.

If *z* is +0 + *i*0, −Inf + *i*0 shall be returned and the divide-by-zero floating-point exception shall be raised.

If *z* is *x* + *i*Inf where *x* is finite, +Inf + *i*π/2 shall be returned.

If *z* is *x* + iNaN where *x* is finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

If *z* is −Inf + *iy* where *y* is positive-signed and finite, +Inf + *i*π shall be returned.

If *z* is +Inf + *iy* where *y* is positive-signed and finite, +Inf + *i*0 shall be returned.

If *z* is −Inf + *i*Inf, +Inf + *i*3π/4 shall be returned.

If *z* is +Inf + *i*Inf, +Inf + *i*π/4 shall be returned.

If *z* is ±Inf + *i*NaN, +Inf + *i*NaN shall be returned.

If *z* is NaN + *iy* where *y* is finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

If *z* is NaN + *i*Inf, +Inf + *i*NaN shall be returned.

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

Ref 7.26.3

On page 698 line 23854 insert the following new cnd_*() sections:

Note to reviewers: changes to cnd_broadcast and cnd_signal may be needed depending on the outcome of Mantis bug 609.

NAME

cnd_broadcast, cnd_signal — broadcast or signal a condition

SYNOPSIS

```
#include <threads.h>

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

DESCRIPTION

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

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

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

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

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

The *cnd_broadcast*() and *cnd_signal*() functions can be called by a thread whether or not it currently owns the mutex that threads calling *cnd_wait*() or *cnd_timedwait*() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling *cnd_broadcast*() or *cnd_signal*().

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

See the APPLICATION USAGE section for *pthread_cond_broadcast*(), substituting *cnd_broadcast*() for *pthread_cond_broadcast*() and *cnd_signal*() for *pthread_cond_signal*().

RATIONALE

As for *pthread_cond_broadcast*() and *pthread_cond_signal*(), spurious wakeups may occur with *cnd_broadcast*() and *cnd_signal*(), necessitating that applications code a predicate-testing-loop around the condition wait. (See the RATIONALE section for *pthread_cond_broadcast*().)

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

FUTURE DIRECTIONS

None.

SEE ALSO

cnd_destroy, *cnd_timedwait*, *pthread_cond_broadcast*

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

CHANGE HISTORY

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

NAME

cnd_destroy, cnd_init — destroy and initialize condition variables

SYNOPSIS

```
#include <threads.h>

void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t *cond);
```

DESCRIPTION

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

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

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

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

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

RETURN VALUE

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

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

ERRORS

See RETURN VALUE.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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

FUTURE DIRECTIONS

None.

SEE ALSO

cnd_broadcast, *cnd_timedwait*

XBD **<threads.h>**

CHANGE HISTORY

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

NAME

cnd_timedwait, cnd_wait — wait on a condition

SYNOPSIS

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

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

ERRORS

See RETURN VALUE.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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

FUTURE DIRECTIONS

None.

SEE ALSO

cnd_broadcast, *cnd_destroy*, *timespec_get*

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

CHANGE HISTORY

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

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

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

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

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

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

> None.

to:

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

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

> [MXC]*csin*(*conj*(*iz*)), *csinf*(*conjf*(*iz*)) and *csinl*(*conjl*(*iz*)) shall return exactly the same value as *conj*(*csin*(*iz*)), *conjf*(*csinf*(*iz*)) and *conjl*(*csinl*(*iz*)), respectively, and *csin*(−*iz*), *csinf*(−*iz*) and *csinl*(−*iz*) shall return exactly the same value as −*csin*(*iz*), −*csinf*(*iz*) and −*csinl*(*iz*), respectively, including for the special values of *iz* below.
>
> If *iz* is $+0 + i0$, $-i\,(+0 + i0)$ shall be returned.
>
> If *iz* is $+0 + i\mathrm{Inf}$, $-i\,(\pm 0 + i\mathrm{NaN})$ shall be returned and the invalid floating-point exception shall be raised; the sign of the imaginary part of the result is unspecified.
>
> If *iz* is $+0 + i\mathrm{NaN}$, $-i\,(\pm 0 + i\mathrm{NaN})$ shall be returned; the sign of the imaginary part of the result is unspecified.
>
> If *iz* is $x + i\mathrm{Inf}$ where x is positive and finite, $-i\,(\mathrm{NaN} + i\mathrm{NaN})$ shall be returned and the invalid floating-point exception shall be raised.
>
> If *iz* is $\mathrm{x} + i\mathrm{NaN}$ where x is non-zero and finite, $-i\,(\mathrm{NaN} + i\mathrm{NaN})$ shall be returned and the invalid floating-point exception may be raised.
>
> If *iz* is $+\mathrm{Inf} + i0$, $-i\,(+\mathrm{Inf} + i0)$ shall be returned.
>
> If *iz* is $+\mathrm{Inf} + iy$ where y is positive and finite, $-i\mathrm{Inf}\,(\cos(y) + i\sin(y))$ shall be returned.
>
> If *iz* is $+\mathrm{Inf} + i\mathrm{Inf}$, $-i\,(\pm\mathrm{Inf} + i\mathrm{NaN})$ shall be returned and the invalid floating-point exception shall be raised; the sign of the imaginary part of the result is unspecified.

If *iz* is +Inf + *i*NaN, −*i* (±Inf + *i*NaN) shall be returned; the sign of the imaginary part of the result is unspecified.

If *iz* is NaN + *i*0, −*i* (NaN + *i*0) shall be returned.

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

If *iz* is NaN + *i*NaN, −*i* (NaN + *i*NaN) shall be returned.[/MXC]

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

None.

to:

The MXC special cases for *csin*() are derived from those for *csinh*() by applying the formula *csin*(*z*) = −*i* *csinh*(*iz*).

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

[MXC]*csinh*(*conj*(*z*)), *csinhf*(*conjf*(*z*)) and *csinhl*(*conjl*(*z*)) shall return exactly the same value as *conj*(*csinh*(*z*)), *conjf*(*csinhf*(*z*)) and *conjl*(*csinhl*(*z*)), respectively, and *csinh*(−*z*), *csinhf*(−*z*) and *csinhl*(−*z*) shall return exactly the same value as −*csinh*(*z*), −*csinhf*(*z*) and −*csinhl*(*z*), respectively, including for the special values of *z* below.

If *z* is +0 + *i*0, +0 + *i*0 shall be returned.

If *z* is +0 + *i*Inf, ±0 + *i*NaN shall be returned and the invalid floating-point exception shall be raised; the sign of the real part of the result is unspecified.

If *z* is +0 + *i*NaN, ±0 + *i*NaN shall be returned; the sign of the real part of the result is unspecified.

If *z* is *x* + *i*Inf where *x* is positive and finite, NaN + *i*NaN shall be returned and the invalid floating-point exception shall be raised.

If *z* is x + *i*NaN where *x* is non-zero and finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

If *z* is +Inf + *i*0, +Inf + *i*0 shall be returned.

If *z* is +Inf + *iy* where *y* is positive and finite, +Inf (cos(*y*) + *i*sin(*y*)) shall be returned.

If *z* is +Inf + *i*Inf, ±Inf + *i*NaN shall be returned and the invalid floating-point exception shall be raised; the sign of the real part of the result is unspecified.

If *z* is +Inf + *i*NaN, ±Inf + *i*NaN shall be returned; the sign of the real part of the result is unspecified.

If *z* is NaN + *i*0, NaN + *i*0 shall be returned.

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

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

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

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

If *z* is ±0 + *i*0, +0 + *i*0 shall be returned.

If the imaginary part of *z* is Inf, +Inf + *i*Inf, shall be returned.

If *z* is *x* + iNaN where *x* is finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

If *z* is −Inf + *iy* where *y* is positive-signed and finite, +0 + *i*Inf shall be returned.

If *z* is +Inf + *iy* where *y* is positive-signed and finite, +Inf + *i*0 shall be returned.

If *z* is −Inf + *i*NaN, NaN ± *i*Inf shall be returned; the sign of the imaginary part of the result is unspecified.

If *z* is +Inf + *i*NaN, +Inf + *i*NaN shall be returned.

If *z* is NaN + *iy* where *y* is finite, NaN + *i*NaN shall be returned and the invalid floating-point exception may be raised.

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

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

[MXC]*ctan*(*conj*(*iz*)), *ctanf*(*conjf*(*iz*)) and *ctanl*(*conjl*(*iz*)) shall return exactly the same value as *conj*(*ctan*(*iz*)), *conjf*(*ctanf*(*iz*)) and *conjl*(*ctanl*(*iz*)), respectively, and *ctan*(−*iz*), *ctanf*(−*iz*) and *ctanl*(−*iz*) shall return exactly the same value as −*ctan*(*iz*), −*ctanf*(*iz*) and −*ctanl*(*iz*), respectively, including for the special values of *iz* below.

If *iz* is +0 + *i*0, −*i* (+0 + *i*0) shall be returned.

If *iz* is 0 + *i*Inf, −*i* (0 + *i*NaN) shall be returned and the invalid floating-point exception shall be raised.

If *iz* is *x* + *i*Inf where *x* is non-zero and finite, −*i* (NaN + *i*NaN) shall be returned and the invalid floating-point exception shall be raised.

If *iz* is 0 + *i*NaN, −*i* (0 + *i*NaN) shall be returned.

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

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

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

If iz is $+\mathrm{Inf} + i\mathrm{NaN}$, $-i\,(1 \pm i0)$ shall be returned; the sign of the real part of the result is unspecified.

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

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

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

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

None.

to:

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

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

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

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

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

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

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

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

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

If z is +Inf + iInf, 1 ± i0 shall be returned; the sign of the imaginary part of the result is unspecified.

If z is +Inf + iNaN, 1 ± i0 shall be returned; the sign of the imaginary part of the result is unspecified.

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

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

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

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

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

to:

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

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

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

to:

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

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

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

to:

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

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

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

Ref 7.22.4.4 para 2

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

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

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

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

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

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

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

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

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

to:

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

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

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

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

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

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

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

to:

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

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

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

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

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

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

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

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

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

to:

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

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

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

to:

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

~~Ref 7.21.5.3 para 3~~
~~On page 892 line 30125 section fopen(), add:~~

~~*wx* or *wbx*   Create file for writing.~~

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

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

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

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

to:

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

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

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

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

If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and …

to:

If the first character in *mode* is *w* or *a*, and …

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

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

to:

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

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

> The file descriptor …

to:

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

Ref (none; see bug 411)
On page 893 line 30160 section fopen(), change the first column heading from:

> ***fopen*() Mode**

to:

> ***fopen*() Mode Without Suffix**

and add the following text after the table:

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

Ref 7.21.5.3 para 5
On page 893 line 30166 section fopen(), add the following new paragraphs:

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

***fopen*() Mode Without Suffix**	***open*() Flags**
[CX]*a* or *ab*	O_WRONLY|O_CREAT|O_EXCL|O_APPEND
a+ or a+*b* or *ab*+	O_RDWR|O_CREAT|O_EXCL|O_APPEND[/CX]
w~~*x*~~ or *wb*~~*x*~~	O_WRONLY|O_CREAT|O_EXCL|O_TRUNC
w+~~*x*~~ or *w*+*b*~~*x*~~ or *wb*+~~*x*~~	O_RDWR|O_CREAT|O_EXCL|O_TRUNC

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

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

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

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

~~Ref 7.21.5.3 para 5~~
~~On page 893 line 30175 section fopen(), add (within the CX shading):~~

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

Ref 7.21.5.3 para 5
On page 895 line 30236 section fopen(), change APPLICATION USAGE from:

None.

to:

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

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

Ref 7.21.5.3 para 5
On page 895 line 30238 section fopen(), after applying bug 411, change ~~RATIONALE from~~:

~~None.~~The *x* mode suffix character was added by C1x only for files opened with a mode string beginning with *w*.

to:

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

and then add two new paragraphs after the one that starts with the above text:

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

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

Note to reviewers: The above change may need to be updated depending on ~~the outcome of~~whether WG14 ~~discussions about~~clarify the "exclusive access" requirement.

Ref 7.22.3.3 para 2
On page 933 line 31673 section free(), change:

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

to:

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

Ref 7.22.3 para 2
On page 933 line 31677 section free(), add a new paragraph:

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

Ref 7.22.3.1
On page 933 line 31691 section free(), add *aligned_alloc* to the SEE ALSO section.

Ref 7.21.5.3 para 5

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

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

to:

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

Ref 7.21.5.3 para 3,5; 7.21.5.4 para 2
On page 942 line 32010 section freopen(), replace the following text:

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

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

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

Ref (none)
On page 944 line 32094 section freopen(), change:

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

to:

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

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

Ref (none)
On page 944 line 32102 section freopen(), after applying bug 411 change:

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

to:

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

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

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

to:

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

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

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

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

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

to:

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

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

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

to:

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

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

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

to:

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

standard, it has not been done here.

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

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

to:

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

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

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

to:

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

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

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

to:

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

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

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

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

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

to:

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

Ref 7.12.6.5 para 2

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

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

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

to:

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

Ref F.10.3.5 para 3
On page 1133 line 38300 section ilogb(), change:

> [XSI]The *x* argument is zero, NaN, or ±Inf.[/XSI]

to:

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

Ref F.10.11 para 1
On page 1174 line 39604 section isgreater(),
and page 1175 line 39642 section isgreaterequal(),
and page 1177 line 39708 section isless(),
and page 1178 line 39746 section islessequal(),
and page 1179 line 39784 section islessgreater(), add a new paragraph:

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

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

Ref 7.30.2.2.1 para 4
On page 1202 line 40411 section iswctype(), remove the CX shading from:

> If *charclass* is (**wctype_t**)0, these functions shall return 0.

Ref 7.17.3.1
On page 1229 line 41126 insert a new kill_dependency() section:

NAME

kill_dependency — terminate a dependency chain

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

Ref 7.12.8.3, 7.1.4 para 5
On page 1241 line 41433 section lgamma(), change:

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

to:

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

XSI]

Ref 7.12.8.3, 7.1.4 para 5
On page 1242 line 41464 section lgamma(), add a new paragraph to APPLICATION USAGE:

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

Ref 7.12.8.3, 7.1.4 para 5
On page 1242 line 41466 section lgamma(), change RATIONALE from:

> None.

to:

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

Ref 7.11.2.1, 7.1.4 para 5
On page 1262 line 42124 section localeconv(), change:

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

to:

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

Ref 7.27.3, 7.1.4 para 5
On page 1265 line 42217 section localtime(), change:

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

to:

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

Ref F.10.3.11 para 2
On page 1280 line 42723 section logb(), add a new paragraph:

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

Ref 7.13.2.1 para 1
On page 1283 line 42780 section longjmp(), change:

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

to:

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

Ref 7.13.2.1 para 2
On page 1283 line 42804 section longjmp(), remove the CX shading from:

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

Ref 7.13.2.1 para 4
On page 1283 line 42807 section longjmp(), change:

> After *longjmp*() is completed, program execution continues …

to:

> After *longjmp*() is completed, thread execution shall continue …

Ref 7.22.3 para 1
On page 1295 line 43144 section malloc(), change:

> a pointer to any type of object

to:

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

Ref 7.22.3 para 1
On page 1295 line 43148 section malloc(), change:

> either a null pointer shall be returned, or …

to:

> either a null pointer shall be returned to indicate an error, or …

Ref 7.22.3 para 2
On page 1295 line 43150 section malloc(), add a new paragraph:

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

Ref 7.22.3.1

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

Ref 7.22.7.1 para 2
On page 1297 line 43194 section mblen(), change:

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

to:

```
mbtowc((wchar_t *)0, (const char *)0, 0);
mbtowc((wchar_t *)0, s, n);
```

Ref 7.22.7 para 1
On page 1297 line 43198 section mblen(), change:

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

to:

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

Ref 7.22.7 para 1, 7.1.4 para 5
On page 1297 line 43206 section mblen(), change:

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

to:

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

Ref 7.29.6.3 para 1, 7.1.4 para 5
On page 1299 line 43254 section mbrlen(), change:

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

to:

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

Ref 7.28.1, 7.1.4 para 5
On page 1301 line 43296 insert a new mbrtoc16() section:

NAME

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

SYNOPSIS

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

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

wide character (which is the value stored).

between 1 and *n* inclusive
: If the next *n* or fewer bytes complete a valid character (which is the value stored); the value returned shall be the number of bytes that complete the character.

(**size_t**)−3
: If the next character resulting from a previous call has been stored, in which case no bytes from the input shall be consumed by the call.

(**size_t**)−2
: If the next *n* bytes contribute to an incomplete but potentially valid character, and all *n* bytes have been processed (no value is stored). When *n* has at least the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

(**size_t**)−1
: If an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

ERRORS

These function shall fail if:

[EILSEQ]
: An invalid character sequence is detected. [CX]In the POSIX locale an [EILSEQ] error cannot occur since all byte values are valid characters.[/CX]

These functions may fail if:

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

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

c16rtomb

XBD **<uchar.h>**

CHANGE HISTORY

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

Ref 7.29.6.3 para 1, 7.1.4 para 5
On page 1301 line 43322 section mbrtowc(), change:

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

to:

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

Ref 7.29.6.4 para 1, 7.1.4 para 5
On page 1304 line 43451 section mbsrtowcs(), change:

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

to:

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

Ref 7.22.7 para 1
On page 1308 line 43557 section mbtowc(), change:

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

to:

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

Ref 7.22.7 para 1, 7.1.4 para 5
On page 1308 line 43567 section mbtowc(), change:

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

to:

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

Ref 7.24.5.1 para 2
On page 1311 line 43642 section memchr(), change:

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

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

to:

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

Ref F.10.3.12 para 2
On page 1346 line 44854 section modf(), add a new paragraph:

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

Ref 7.26.4
On page 1384 line 46032 insert the following new mtx_*() sections:

NAME

mtx_destroy, mtx_init — destroy and initialize a mutex

SYNOPSIS

```
#include <threads.h>

void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
```

DESCRIPTION

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

The *mtx_destroy*() function shall release any resources used by the mutex pointed to by *mtx*. A destroyed mutex object can be reinitialized using *mtx_init*(); the results of otherwise referencing the object after it has been destroyed are undefined. It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a *cnd_timedwait*() or *cnd_wait*() call by another thread, results in undefined behavior. The behavior is undefined if the value specified by the *mtx* argument to *mtx_destroy*() does not refer to an initialized mutex.

The *mtx_init*() function shall initialize a mutex object with properties indicated by *type*, whose valid values include:

`mtx_plain`	for a simple non-recursive mutex,
`mtx_timed`	for a non-recursive mutex that supports timeout,
`mtx_plain | mtx_recursive`	for a simple recursive mutex, or
`mtx_timed | mtx_recursive`	for a recursive mutex that supports timeout.

If the *mtx_init*() function succeeds, it shall set the mutex pointed to by *mtx* to a value that uniquely identifies the newly initialized mutex. Upon successful initialization, the state of the mutex becomes initialized and unlocked. Attempting to initialize an already initialized mutex results in undefined behavior.

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

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

RETURN VALUE

The *mtx_destroy*() function shall not return a value.

The *mtx_init*() function shall return `thrd_success` on success or `thrd_error` if the request could not be honored.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

A mutex can be destroyed immediately after it is unlocked. However, since attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a *cnd_timedwait*() or *cnd_wait*() call by another thread results in undefined behavior, care must be taken to ensure that no other thread may be referencing the mutex.

RATIONALE

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

FUTURE DIRECTIONS

None.

SEE ALSO

mtx_lock

XBD **<threads.h>**

CHANGE HISTORY

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

NAME

mtx_lock, mtx_timedlock, mtx_trylock, mtx_unlock — lock and unlock a mutex

SYNOPSIS

```
#include <threads.h>

int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t * restrict mtx,
                const struct timespec * restrict ts);
```

```
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
```

DESCRIPTION

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

The *mtx_lock*() function shall block until it locks the mutex pointed to by *mtx*. If the mutex is non-recursive, the application shall ensure that it is not already locked by the calling thread.

The *mtx_timedlock*() function shall block until it locks the mutex pointed to by mtx or until after the TIME_UTC -based calendar time pointed to by *ts*. The application shall ensure that the specified mutex supports timeout. [CX]Under no circumstance shall the function fail with a timeout if the mutex can be locked immediately. The validity of the *ts* parameter need not be checked if the mutex can be locked immediately.[/CX]

The *mtx_trylock*() function shall endeavor to lock the mutex pointed to by *mtx*. If the mutex is already locked (by any thread, including the current thread), the function shall return without blocking. If the mutex is recursive and the mutex is currently owned by the calling thread, the mutex lock count (see below) shall be incremented by one and the *mtx_trylock*() function shall immediately return success.

[CX]These functions shall not be affected if the calling thread executes a signal handler during the call; if a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.[/CX]

If a call to *mtx_lock*(), *mtx_timedlock*() or *mtx_trylock*() locks the mutex, prior calls to *mtx_unlock*() on the same mutex shall synchronize with this lock operation.

The *mtx_unlock*() function shall unlock the mutex pointed to by *mtx* . The application shall ensure that the mutex pointed to by *mtx* is locked by the calling thread. [CX]If there are threads blocked on the mutex object referenced by *mtx* when *mtx_unlock*() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.[/CX]

A recursive mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire.

For purposes of determining the existence of a data race, mutex lock and unlock operations on mutexes of type **mtx_t** behave as atomic operations. All lock and unlock operations on a particular mutex occur in some particular total order.

If *mtx* does not refer to an initialized mutex object, the behavior of these functions is undefined.

RETURN VALUE

The *mtx_lock*() and *mtx_unlock*() functions shall return `thrd_success` on success, or `thrd_error` if the request could not be honored.

The *mtx_timedlock*() function shall return `thrd_success` on success, or `thrd_timedout` if the time specified was reached without acquiring the requested resource, or `thrd_error` if the request could not be honored.

The *mtx_trylock*() function shall return `thrd_success` on success, or `thrd_busy` if the resource requested is already in use, or `thrd_error` if the request could not be honored. The *mtx_trylock*() function can spuriously fail to lock an unused resource, in which case it shall return `thrd_busy`.

ERRORS

See RETURN VALUE.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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

Since <**pthread.h**> has no equivalent of the `mtx_timed` mutex property, if the <**threads.h**> interfaces are implemented as a thin wrapper around <**pthread.h**> interfaces (meaning **mtx_t** and **pthread_mutex_t** are the same type), all mutexes support timeout and *mtx_timedlock*() will not fail for a mutex that was not initialized with `mtx_timed`. Alternatively, implementations can use a less thin wrapper where **mtx_t** contains additional properties that are not held in **pthread_mutex_t** in order to be able to return a failure indication from *mtx_timedlock*() calls where the mutex was not initialized with `mtx_timed`.

FUTURE DIRECTIONS

None.

SEE ALSO

mtx_destroy, *timespec_get*

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

CHANGE HISTORY

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

Ref F.10.8.2 para 2
On page 1388 line 46143 section nan(), add a new paragraph:

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

Ref F.10.8.3 para 2, F.10.8.4 para 2
On page 1395 line 46388 section nextafter(), add a new paragraph:

> [MX]Even though underflow or overflow can occur, the returned value shall be independent of the current rounding direction mode.[/MX]

Ref 7.22.3 para 2
On page 1448 line 48069 section posix_memalign(), add a new (unshaded) paragraph:

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

Ref 7.22.3.1
On page 1449 line 48107 section posix_memalign(), add *aligned_alloc* to the SEE ALSO section.

Ref F.10.4.4 para 1
On page 1548 line 50724 section pow(), change:

> On systems that support the IEC 60559 Floating-Point option, if *x* is ±0, a pole error shall occur and *pow*(), *powf*(), and *powl*() shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively if *y* is an odd integer, or HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively if *y* is not an odd integer.

to:

> On systems that support the IEC 60559 Floating-Point option, if *x* is ±0:
>
> - if *y* is an odd integer, a pole error shall occur and *pow*(), *powf*(), and *powl*() shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively;
> - if *y* is finite and is not an odd integer, a pole error shall occur and *pow*(), *powf*(), and *powl*() shall return HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively;
> - if y is -Inf, a pole error may occur and *pow*(), *powf*(), and *powl*() shall return HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

Ref 7.26
On page 1603 line 52244 section pthread_cancel(), add a new paragraph:

> If *thread* refers to a thread that was created using *thrd_create*(), the behavior is undefined.

Ref 7.26.5.6
On page 1603 line 52277 section pthread_cancel(), add a new RATIONALE paragraph:

> Use of *pthread_cancel*() to cancel a thread that was created using *thrd_create*() is undefined because *thrd_join*() has no way to indicate a thread was cancelled. The standard developers considered adding a `thrd_canceled` enumeration constant that *thrd_join*() would return in

this case. However, this return would be unexpected in code that is written to conform to the ISO C standard, and it would also not solve the problem that threads which use only ISO C **<threads.h>** interfaces (such as ones created by third party libraries written to conform to the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not provide cancellation cleanup handlers.

Ref 7.26.5.5
On page 1639 line 53422 section pthread_exit(), change:

```
void pthread_exit(void *value_ptr);
```

to:

```
_Noreturn void pthread_exit(void *value_ptr);
```

Ref 7.26.6
On page 1639 line 53427 section pthread_exit(), change:

> After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions shall be called in an unspecified order.

to:

> After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data (whether associated with key type **tss_t** or **pthread_key_t**), appropriate destructor functions shall be called in an unspecified order.

Ref 7.26.5.5
On page 1639 line 53432 section pthread_exit(), change:

> An implicit call to *pthread_exit*() is made when a thread other than the thread in which *main*() was first invoked returns from the start routine that was used to create it.

to:

> An implicit call to *pthread_exit*() is made when a thread that was not created using *thrd_create*(), and is not the thread in which *main*() was first invoked, returns from the start routine that was used to create it.

Ref 7.26.5.5
On page 1639 line 53451 section pthread_exit(), change APPLICATION USAGE from:

> None.

to:

> Calls to *pthread_exit*() should not be made from threads created using *thrd_create*(), as their exit status has a different type (**int** instead of **void ***). If *pthread_exit*() is called from the initial thread and it is not the last thread to terminate, other threads should not try to obtain its exit status using *thrd_join*().

Ref 7.26.5.5

On page 1639 line 53453 section pthread_exit(), change:

> The normal mechanism by which a thread terminates is to return from the routine that was specified in the *pthread_create*() call that started it.

to:

> The normal mechanism by which a thread that was started using *pthread_create*() terminates is to return from the routine that was specified in the *pthread_create*() call that started it.

Ref 7.26.5.5, 7.26.6
On page 1640 line 53470 section pthread_exit(), add pthread_key_create, thrd_create, thrd_exit and tss_create to the SEE ALSO section.

Ref 7.26.5.5
On page 1649 line 53748 section pthread_join(), add a new paragraph:

> If *thread* refers to a thread that was created using *thrd_create*() and the thread terminates, or has already terminated, by returning from its start routine, the behavior of *pthread_join*() is undefined. If *thread* refers to a thread that terminates, or has already terminated, by calling *thrd_exit*(), the behavior of *pthread_join*() is undefined.

Ref 7.26.5.5
On page 1651 line 53819 section pthread_join(), add a new RATIONALE paragraph:

> The *pthread_join*() function cannot be used to obtain the exit status of a thread that was created using *thrd_create*() and which terminates by returning from its start routine, or of a thread that terminates by calling *thrd_exit*(), because such threads have an **int** exit status, instead of the **void *** that *pthread_join*() returns via its *value_ptr* argument.

Ref 7.22.4.7
On page 1765 line 57040 insert the following new quick_exit() section:

NAME

> quick_exit — terminate a process

SYNOPSIS

```
#include <stdlib.h>

_Noreturn void quick_exit(int status);
```

DESCRIPTION

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

> The *quick_exit*() function shall cause normal process termination to occur. It shall not call functions registered with *atexit*() nor any registered signal handlers. If a process calls the *quick_exit*() function more than once, or calls the *exit*() function in addition to the *quick_exit*() function, the behavior is undefined. If a signal is raised while the *quick_exit*() function is executing, the behavior is undefined.

The *quick_exit*() function shall first call all functions registered by *at_quick_exit*(), in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the *longjmp*() [CX] or *siglongjmp*()[/CX] function is made that would terminate the call to the registered function, the behavior is undefined.

If a function registered by a call to *at_quick_exit*() fails to return, the remaining registered functions shall not be called and the rest of the *quick_exit*() processing shall not be completed.

Finally, the *quick_exit*() function shall terminate the process as if by a call to *_Exit*(*status*).

RETURN VALUE

The *quick_exit*() function does not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

_Exit, *at_quick_exit*, *atexit*, *exit*

XBD **<stdlib.h>**

CHANGE HISTORY

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

Ref 7.22.2.1 para 3, 7.1.4 para 5
On page 1767 line 57095 section rand(), change:

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

to:

The *rand*() function need not be thread-safe; however, *rand*() shall avoid data races with all functions other than non-thread-safe pseudo-random sequence generation functions.

Ref 7.22.2.2 para 3, 7.1.4 para 5
On page 1767 line 57105 section rand(), add a new paragraph:

The *srand*() function need not be thread-safe; however, *srand*() shall avoid data races with all functions other than non-thread-safe pseudo-random sequence generation functions.

Ref 7.22.3 para 1,2; 7.22.3.5 para 2,3,4; 7.31.12 para 2
On page 1788 line 57862-57892 section realloc(), replace the DESCRIPTION and RETURN VALUE sections with:

DESCRIPTION

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

The *realloc*() function shall deallocate the old object pointed to by *ptr* and return a pointer to a new object that has the size specified by *size*. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

If *ptr* is a null pointer, *realloc*() shall be equivalent to *malloc*() function for the specified size. Otherwise, if *ptr* does not match a pointer returned earlier by *aligned_alloc*(), *calloc*(), *malloc*(), [ADV]*posix_memalign*(),[/ADV] *realloc*(), or a function in POSIX.1-20xx that allocates memory as if by *malloc*(), or if the space has been deallocated by a call to *free*() or *realloc*(), the behavior is undefined.

If *size* is non-zero and memory for the new object is not allocated, the old object shall not be deallocated. [OB]If *size* is zero and memory for the new object is not allocated, it is implementation-defined whether the old object is deallocated; if the old object is not deallocated, its value shall be unchanged.[/OB]

The order and contiguity of storage allocated by successive calls to *realloc*() is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. [OB]If the size of the space requested is 0, the behavior is implementation-defined: either a null pointer shall be returned to indicate an error, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object.[/OB]

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

RETURN VALUE

The *realloc*() function shall return a pointer to the new object (which can have the same value as a pointer to the old object), or a null pointer if the new object has not been allocated.

[OB]If size is zero, either:

- A null pointer shall be returned [CX]and, if *ptr* is not a null pointer, *errno* shall be set to an implementation-defined value.[/CX]
- A pointer to the allocated space shall be returned, and the memory object pointed to by *ptr* shall be freed. The application shall ensure that the pointer is not used to access an object.[/OB]

If there is not enough available memory, *realloc*() shall return a null pointer [CX]and set *errno* to [ENOMEM][/CX].

Ref 7.22.3.5 para 3,4
On page 1789 line 57899 section realloc(), change:

The description of *realloc*() has been modified from previous versions of this standard to align with the ISO/IEC 9899: 1999 standard. Previous versions explicitly permitted a call to *realloc(p, 0) to free the space pointed to by p and return a null pointer. While this behavior could be interpreted as permitted by this version of the standard, the C language committee have indicated that this interpretation is incorrect. Applications should assume that if realloc() returns a null pointer, the space pointed to by p has not been freed. Since this could lead to double-frees, implementations should also set errno if a null pointer actually indicates a failure, and applications should only free the space if errno was changed.*

to:

The ISO C standard makes it implementation-defined whether a call to *realloc*(p, 0) frees the space pointed to by *p* if it returns a null pointer because memory for the new object was not allocated. POSIX.1 instead requires that implementations set *errno* if a null pointer is returned and the space has not been freed, and POSIX applications should only free the space if *errno* was changed.

Ref 7.31.12 para 2
On page 1789 line 57909-57912 section realloc(), change FUTURE DIRECTIONS to:

The ISO C standard states that invoking *realloc*() with a *size* argument equal to zero is an obsolescent feature. This feature may be removed in a future version of this standard.

Ref 7.22.3.1
On page 1789 line 57914 section realloc(), add *aligned_alloc* to the SEE ALSO section.

Ref F.10.7.2 para 2
On page 1809 line 58638 section remainder(), add a new paragraph:

[MX]When subnormal results are supported, the returned value shall be exact.[/MX]

Ref F.10.7.3 para 2
On page 1814 line 58758 section remquo(), add a new paragraph:

[MX]When subnormal results are supported, the returned value shall be exact.[/MX]

Ref F.10.6.6 para 3

On page 1828 line 59258 section round(), add a new paragraph:

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

Ref F.10.6.6 para 3
On page 1828 line 59272 section round(), delete from APPLICATION USAGE:

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

Ref F.10.3.13 para 2
On page 1829 line 59306 section scalbln(), add a new paragraph:

> [MX]If the calculation does not overflow or underflow, the returned value shall be exact and shall be independent of the current rounding direction mode.[/MX]

Ref 7.11.1.1 para 5
On page 1903 line 61520 section setlocale(), ~~remove the CX shading from~~change:

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

to:

> The *setlocale*() function need not be thread-safe; however, it shall avoid data races with all function calls that do not affect and are not affected by the global locale.

Ref 7.13.2.1 para 1
On page 1970 line 63497 section siglongjmp(), change:

```
void siglongjmp(sigjmp_buf env, int val);
```

to:

```
_Noreturn void siglongjmp(sigjmp_buf env, int val);
```

Ref 7.13.2.1 para 4
On page 1970 line 63504 section siglongjmp(), change:

> After *siglongjmp*() is completed, program execution shall continue …

to:

> After *siglongjmp*() is completed, thread execution shall continue …

Ref 7.14.1.1 para 5
On page 1971 line 63564 section signal(), change:

> with static storage duration

to:

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

Ref 7.14.1.1 para 7
On page 1972 line 63573 section signal(), add a new paragraph:

[CX]The *signal*() function is required to be thread-safe. (See [xref to 2.9.1 Thread-Safety].)[/CX]

Ref 7.14.1.1 para 7
On page 1972 line 63591 section signal(), change RATIONALE from:

None.

to:

The ISO C standard says that the use of *signal*() in a multi-threaded program results in undefined behavior. However, POSIX.1 has required *signal*() to be thread-safe since before threads were added to the ISO C standard.

Ref F.10.4.5 para 1
On page 2009 line 64624 section sqrt(), add:

[MX]The returned value shall be dependent on the current rounding direction mode.[/MX]

Ref 7.24.6.2 para 3, 7.1.4 para 5
On page 2035 line 65231 section strerror(), change:

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

to:

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

Ref 7.22.1.3 para 10
On page 2073 line 66514 section strtod(), change:

If the correct value is outside the range of representable values

to:

If the correct value would cause an overflow and default rounding is in effect

Ref 7.24.5.8 para 6, 7.1.4 para 5
On page 2078 line 66674 section strtok(), change:

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

to:

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

Ref 7.22.4.8, 7.1.4 para 5
On page 2107 line 67579 section system(), change:

> The *system*() function need not be thread-safe.

to:

> [CX]If concurrent calls to *system*() are made from multiple threads, it is unspecified whether:
>
> - each call saves and restores the dispositions of the SIGINT and SIGQUIT signals independently, or
> - in a set of concurrent calls the dispositions in effect after the last call returns are those that were in effect on entry to the first call.
>
> If a thread is cancelled while it is in a call to *system*(), it is unspecified whether the child process is terminated and waited for, or is left running.[/CX]

Ref 7.22.4.8, 7.1.4 para 5
On page 2108 line 67627 section system(), change:

> Using the *system*() function in more than one thread in a process or when the SIGCHLD signal is being manipulated by more than one thread in a process may produce unexpected results.

to:

> Although *system*() is required to be thread-safe, it is recommended that concurrent calls from multiple threads are avoided, since *system*() is not required to coordinate the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of overlapping calls, and therefore the signals might end up being set to ignored after the last call returns. Applications should also avoid cancelling a thread while it is in a call to *system*() as the child process may be left running in that event. In addition, if another thread alters the disposition of the SIGCHLD signal, a call to *signal*() may produce unexpected results.

Ref 7.22.4.8, 7.1.4 para 5
On page 2109 line 67675 section system(), delete:

```
#include <signal.h>
```

Ref 7.22.4.8, 7.1.4 para 5
On page 2109 line 67692,67696,67712 section system(), change `sigprocmask` to `pthread_sigmask`.

Ref 7.22.4.8, 7.1.4 para 5
On page 2110 line 67718 section system(), change:

> Note also that the above example implementation is not thread-safe. Implementations can provide a thread-safe *system*() function, but doing so involves complications such as how to restore the signal dispositions for SIGINT and SIGQUIT correctly if there are overlapping calls, and how to deal with cancellation. The example above would not restore the signal dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-

safe implementation since canceling a non-thread-safe function results in undefined behavior (see Section 2.9.5.2, on page 518). To avoid leaking a process ID, a thread-safe implementation would need to terminate the child process when acting on a cancellation.

to:

Earlier versions of this standard did not require *system*() to be thread-safe because it alters the process-wide disposition of the SIGINT and SIGQUIT signals. It is now required to be thread-safe to align with the ISO C standard, which (since the introduction of threads in 2011) requires that it avoids data races. However, the function is not required to coordinate the saving and restoring of the dispositions of the SIGINT and SIGQUIT signals across a set of overlapping calls, and the above example does not do so. The example also does not terminate and wait for the child process if the calling thread is cancelled, and so would leak a process ID in that event.

Ref 7.26.5
On page 2148 line 68796 insert the following new thrd_*() sections:

NAME

thrd_create — thread creation

SYNOPSIS

```
#include <threads.h>

int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

DESCRIPTION

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

The *thrd_create*() function shall create a new thread executing *func*(*arg*). If the *thrd_create*() function succeeds, it shall set the object pointed to by *thr* to the identifier of the newly created thread. (A thread's identifier might be reused for a different thread once the original thread has exited and either been detached or joined to another thread.) The completion of the *thrd_create*() function shall synchronize with the beginning of the execution of the new thread.

[CX]The signal state of the new thread shall be initialized as follows:

- The signal mask shall be inherited from the creating thread.

- The set of signals pending for the new thread shall be empty.

The thread-local current locale shall not be inherited from the creating thread.

The floating-point environment shall be inherited from the creating thread.[/CX]

[XSI] The alternate stack shall not be inherited from the creating thread.[/XSI]

Returning from *func* shall have the same behavior as invoking *thrd_exit*() with the value returned from *func*.

If *thrd_create*() fails, no new thread shall be created and the contents of the location referenced by *thr* are undefined.

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

RETURN VALUE

The *thrd_create*() function shall return `thrd_success` on success; or `thrd_nomem` if no memory could be allocated for the thread requested; or `thrd_error` if the request could not be honored, [CX]such as if the system-imposed limit on the total number of threads in a process {PTHREAD_THREADS_MAX} would be exceeded.[/CX]

ERRORS

See RETURN VALUE.

EXAMPLES

None.

APPLICATION USAGE

There is no requirement on the implementation that the ID of the created thread be available before the newly created thread starts executing. The calling thread can obtain the ID of the created thread through the *thr* argument of the *thrd_create*() function, and the newly created thread can obtain its ID by a call to *thrd_current*().

RATIONALE

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create, *thrd_current*, *thrd_detach*, *thrd_exit*, *thrd_join*

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

CHANGE HISTORY

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

NAME

thrd_current — get the calling thread ID

SYNOPSIS

```
#include <threads.h>

thrd_t thrd_current(void);
```

DESCRIPTION

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

The *thrd_current*() function shall identify the thread that called it.

RETURN VALUE

The *thrd_current*() function shall return the thread ID of the thread that called it.

The *thrd_current*() function shall always be successful. No return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_self, *thrd_create*, *thrd_equal*

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

CHANGE HISTORY

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

NAME

thrd_detach — detach a thread

SYNOPSIS

```
#include <threads.h>

int thrd_detach(thrd_t thr);
```

DESCRIPTION

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

The *thrd_detach*() function shall change the thread *thr* from joinable to detached, indicating to the implementation that any resources allocated to the thread can be reclaimed when that thread terminates. The application shall ensure that the thread identified by *thr* has not been previously detached or joined with another thread.

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

RETURN VALUE

The *thrd_detach*() function shall return `thrd_success` on success or `thrd_error` if the request could not be honored.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_detach, thrd_create, thrd_join

XBD **<threads.h>**

CHANGE HISTORY

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

NAME

thrd_equal — compare thread IDs

SYNOPSIS

```
#include <threads.h>

int thrd_equal(thrd_t thr0, thrd_t thr1);
```

DESCRIPTION

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

The *thrd_equal*() function shall determine whether the thread identified by *thr0* refers to the thread identified by *thr1*.

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

RETURN VALUE

The *thrd_equal*() function shall return a non-zero value if *thr0* and *thr1* are equal; otherwise, zero shall be returned.

If either *thr0* or *thr1* is not a valid thread ID [CX]and is not equal to PTHREAD_NULL (which is defined in **<pthread.h>**)[/CX], the behavior is undefined.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

See the RATIONALE section for *pthread_equal*().

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_equal, thrd_current

XBD **<pthread.h>**, **<threads.h>**

CHANGE HISTORY

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

NAME

thrd_exit — thread termination

SYNOPSIS

```
#include <threads.h>

_Noreturn void thrd_exit(int res);
```

DESCRIPTION

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

For every thread-specific storage key [CX](regardless of whether it has type **tss_t** or **pthread_key_t**)[/CX] which was created with a non-null destructor and for which the value is non-null, *thrd_exit*() shall set the value associated with the key to a null pointer value and then invoke the destructor with its previous value. The order in which destructors are invoked is unspecified.

If after this process there remain keys with both non-null destructors and values, the implementation shall repeat this process up to [CX] {PTHREAD_DESTRUCTOR_ITERATIONS}[/CX] times.

Following this, the *thrd_exit*() function shall terminate execution of the calling thread and shall set its exit status to *res*. [CX]Thread termination shall not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor shall it perform any process-level cleanup actions, including, but not limited to, calling any *atexit*() routines that might exist.[/CX]

An implicit call to *thrd_exit*() is made when a thread that was created using *thrd_create*() returns from the start routine that was used to create it (see [xref to thrd_create()]).

[CX]The behavior of *thrd_exit*() is undefined if called from a destructor function that was invoked as a result of either an implicit or explicit call to *thrd_exit*().[/CX]

The process shall exit with an exit status of zero after the last thread has been terminated. The behavior shall be as if the implementation called *exit*() with a zero argument at thread termination time.

RETURN VALUE

This function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Calls to *thrd_exit*() should not be made from threads created using *pthread_create*() or via a SIGEV_THREAD notification, as their exit status has a different type (**void *** instead of **int**). If *thrd_exit*() is called from the initial thread and it is not the last thread to terminate, other threads should not try to obtain its exit status using *pthread_join*().

RATIONALE

The normal mechanism by which a thread that was started using *thrd_create*() terminates is to return from the function that was specified in the *thrd_create*() call that started it. The *thrd_exit*() function provides the capability for such a thread to terminate without requiring a return from the start routine of that thread, thereby providing a function analogous to *exit*().

Regardless of the method of thread termination, the destructors for any existing thread-specific data are executed.

FUTURE DIRECTIONS

None.

SEE ALSO

exit, pthread_create, thrd_join

XBD **<threads.h>**

CHANGE HISTORY

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

NAME

thrd_join — wait for thread termination

SYNOPSIS

```
#include <threads.h>

int thrd_join(thrd_t thr, int *res);
```

DESCRIPTION

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

The *thrd_join*() function shall join the thread identified by *thr* with the current thread by blocking until the other thread has terminated. If the parameter *res* is not a null pointer, *thrd_join*() shall store the thread's exit status in the integer pointed to by *res*. The termination of the other thread shall synchronize with the completion of the *thrd_join*() function. The application shall ensure that the thread identified by *thr* has not been previously detached or joined with another thread.

The results of multiple simultaneous calls to *thrd_join*() specifying the same target thread are undefined.

The behavior is undefined if the value specified by the *thr* argument to *thrd_join*() refers to the calling thread.

[CX]It is unspecified whether a thread that has exited but remains unjoined counts against {PTHREAD_THREADS_MAX}.

If *thr* refers to a thread that was created using *pthread_create*() or via a SIGEV_THREAD notification and the thread terminates, or has already terminated, by returning from its start routine, the behavior of *thrd_join*() is undefined. If *thr* refers to a thread that terminates, or has already terminated, by calling *pthread_exit*() or by being cancelled, the behavior of *thrd_join*() is undefined.

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

RETURN VALUE

The *thrd_join*() function shall return `thrd_success` on success or `thrd_error` if the request could not be honored.

[CX]It is implementation-defined whether *thrd_join*() detects deadlock situations; if it does detect them, it shall return `thrd_error` when one is detected.[/CX]

ERRORS

See RETURN VALUE.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *thrd_join*() function provides a simple mechanism allowing an application to wait for a thread to terminate. After the thread terminates, the application may then choose to clean up resources that were used by the thread. For instance, after *thrd_join*() returns, any application-provided stack storage could be reclaimed.

The *thrd_join*() or *thrd_detach*() function should eventually be called for every thread that is created using *thrd_create*() so that storage associated with the thread may be reclaimed.

The *thrd_join*() function cannot be used to obtain the exit status of a thread that was created using *pthread_create*() or via a SIGEV_THREAD notification and which terminates by returning from its start routine, or of a thread that terminates by calling *pthread_exit*(), because such threads have a **void *** exit status, instead of the **int** that *thrd_join*() returns via its *res* argument.

The *thrd_join*() function cannot be used to obtain the exit status of a thread that terminates by being cancelled because it has no way to indicate that a thread was cancelled. (The *pthread_join*() function does this by returning a reserved **void *** exit status; it is not possible to reserve an **int** value for this purpose without introducing a conflict with the ISO C standard.) The standard developers considered adding a `thrd_canceled` enumeration constant that *thrd_join*() would return in this case. However, this return would be unexpected in code that is written to conform to the ISO C standard, and it would also not solve the problem that threads which use only ISO C **<threads.h>** interfaces (such as ones created by third party libraries written to conform to the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not provide cancellation cleanup handlers.

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create, pthread_exit, pthread_join, thrd_create, thrd_exit

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

CHANGE HISTORY

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

NAME

thrd_sleep — suspend execution for an interval

SYNOPSIS

```
#include <threads.h>

int thrd_sleep(const struct timespec *duration,
     struct timespec *remaining);
```

DESCRIPTION

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

The *thrd_sleep*() function shall suspend execution of the calling thread until either the interval specified by *duration* has elapsed or a signal is delivered to the calling thread whose action is to invoke a signal-catching function or to terminate the process. If interrupted by a signal and the *remaining* argument is not null, the amount of time remaining (the requested interval minus the time actually slept) shall be stored in the interval it points to. The *duration* and *remaining* arguments can point to the same object.

The suspension time may be longer than requested because the interval is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than that specified, as measured by the system clock TIME_UTC.

RETURN VALUE

The *thrd_sleep*() function shall return zero if the requested time has elapsed, −1 if it has been interrupted by a signal, or a negative value (which may also be −1) if it fails for any other reason. [CX]If it returns a negative value, it shall set *errno* to indicate the error.[/CX]

ERRORS

[CX]The *thrd_sleep*() function shall fail if:

[EINTR]
: The *thrd_sleep*() function was interrupted by a signal.

[EINVAL]
: The *duration* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.[/CX]

EXAMPLES

None.

APPLICATION USAGE

Since the return value may be -1 for errors other than [EINTR], applications should examine *errno* to distinguish [EINTR] from other errors (and thus determine whether the unslept time is available in the interval pointed to by *remaining*).

RATIONALE

The *thrd_sleep*() function is identical to the *nanosleep*() function except that the return value may be any negative value when it fails with an error other than [EINTR].

FUTURE DIRECTIONS

None.

SEE ALSO

nanosleep

XBD **<threads.h>**, **<time.h>**

CHANGE HISTORY

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

NAME

thrd_yield — yield the processor

SYNOPSIS

```
#include <threads.h>

void thrd_yield(void);
```

DESCRIPTION

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

[CX]The *thrd_yield*() function shall force the running thread to relinquish the processor until it again becomes the head of its thread list.[/CX]

RETURN VALUE

This function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

See the APPLICATION USAGE section for *sched_yield*().

RATIONALE

The *thrd_yield*() function is identical to the *sched_yield*() function except that it does not return a value.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_yield

XBD **<threads.h>**

CHANGE HISTORY

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

Ref 7.27.2.5
On page 2161 line 69278 insert a new timespec_get() section:

NAME

timespec_get — get time

SYNOPSIS

```
#include <time.h>
```

```
int timespec_get(struct timespec *ts, int base);
```

DESCRIPTION

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

The *timespec_get*() function shall set the interval pointed to by *ts* to hold the current calendar time based on the specified time base.

[CX]If *base* is TIME_UTC, the members of *ts* shall be set to the same values as would be set by a call to *clock_gettime*(CLOCK_REALTIME, *ts*). If the number of seconds will not fit in an object of type **time_t**, the function shall return zero.[/CX]

RETURN VALUE

If the *timespec_get*() function is successful it shall return the non-zero value *base*; otherwise, it shall return zero.

ERRORS

See DESCRIPTION.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres, time

XBD **<time.h>**

CHANGE HISTORY

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

Ref 7.21.4.4 para 4, 7.1.4 para 5
On page 2164 line 69377 section tmpnam(), change:

[CX]The *tmpnam*() function need not be thread-safe if called with a NULL parameter.[/CX]

to:

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

Ref 7.30.3.2.1 para 4
On page 2171 line 69568 section towctrans(), change:

> If successful, the *towctrans*() [CX]and *towctrans_l*()[/CX] functions shall return the mapped value of *wc* using the mapping described by *desc*. Otherwise, they shall return *wc* unchanged.

to:

> If successful, the *towctrans*() [CX]and *towctrans_l*()[/CX] functions shall return the mapped value of *wc* using the mapping described by *desc*, or the value of *wc* unchanged if *desc* is zero. [CX]Otherwise, they shall return *wc* unchanged.[/CX]

Ref F.10.6.8 para 2
On page 2177 line 69716 section trunc(), add a new paragraph:

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

Ref F.10.6.8 para 1,2
On page 2177 line 69719 section trunc(), change:

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

to:

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

Ref F.10.6.8 para 2
On page 2177 line 69730 section trunc(), delete from APPLICATION USAGE:

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

Ref 7.26.6
On page 2182 line 69835 insert the following new tss_*() sections:

NAME

> tss_create — thread-specific data key creation

SYNOPSIS

```
#include <threads.h>

int tss_create(tss_t *key, tss_dtor_t dtor);
```

DESCRIPTION

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

> The *tss_create*() function shall create a thread-specific storage pointer with destructor *dtor*, which can be null.

A null pointer value shall be associated with the newly created key in all existing threads. Upon subsequent thread creation, the value associated with all keys shall be initialized to a null pointer value in the new thread.

Destructors associated with thread-specific storage shall not be invoked at process termination.

The behavior is undefined if the *tss_create*() function is called from within a destructor.

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

RETURN VALUE

If the *tss_create*() function is successful, it shall set the thread-specific storage pointed to by *key* to a value that uniquely identifies the newly created pointer and shall return `thrd_success`; otherwise, `thrd_error` shall be returned and the thread-specific storage pointed to by *key* has an indeterminate value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *tss_create*() function performs no implicit synchronization. It is the responsibility of the programmer to ensure that it is called exactly once per key before use of the key.

RATIONALE

If the value associated with a key needs to be updated during the lifetime of the thread, it may be necessary to release the storage associated with the old value before the new value is bound. Although the *tss_set*() function could do this automatically, this feature is not needed often enough to justify the added complexity. Instead, the programmer is responsible for freeing the stale storage:

```
old = tss_get(key);
new = allocate();
destructor(old);
tss_set(key, new);
```

There is no notion of a destructor-safe function. If an application does not call *thrd_exit*() or *pthread_exit*() from a signal handler, or if it blocks any signal whose handler may call *thrd_exit*() or *pthread_exit*() while calling async-unsafe functions, all functions can be safely called from destructors.

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_exit, *pthread_key_create*, *thrd_exit*, *tss_delete*, *tss_get*

XBD **<threads.h>**

CHANGE HISTORY

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

NAME

tss_delete — thread-specific data key deletion

SYNOPSIS

```
#include <threads.h>

void tss_delete(tss_t key);
```

DESCRIPTION

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

The *tss_delete*() function shall release any resources used by the thread-specific storage identified by *key*. The thread-specific data values associated with *key* need not be null at the time *tss_delete*() is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after *tss_delete*() is called.

The application shall ensure that the *tss_delete*() function is only called with a value for *key* that was returned by a call to *tss_create*() before the thread commenced executing destructors.

If *tss_delete*() is called while another thread is executing destructors, whether this will affect the number of invocations of the destructor associated with *key* on that thread is unspecified.

The *tss_delete*() function shall be callable from within destructor functions. Calling *tss_delete*() shall not result in the invocation of any destructors. Any destructor function that was associated with *key* shall no longer be called upon thread exit.

Any attempt to use *key* following the call to *tss_delete*() results in undefined behavior.

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

RETURN VALUE

This function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A thread-specific data key deletion function has been included in order to allow the resources associated with an unused thread-specific data key to be freed. Unused thread-specific data keys can arise, among other scenarios, when a dynamically loaded module that allocated a key is unloaded.

Conforming applications are responsible for performing any cleanup actions needed for data structures associated with the key to be deleted, including data referenced by thread-specific data values. No such cleanup is done by *tss_delete*(). In particular, destructor functions are not called. See the RATIONALE for *pthread_key_delete*() for the reasons for this division of responsibility.

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_key_create, *tss_create*

XBD **<threads.h>**

CHANGE HISTORY

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

NAME

tss_get, tss_set — thread-specific data management

SYNOPSIS

```
#include <threads.h>

void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

DESCRIPTION

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

The *tss_get*() function shall return the value for the current thread held in the thread-specific storage identified by *key*.

The *tss_set*() function shall set the value for the current thread held in the thread-specific storage identified by *key* to *val*. This action shall not invoke the destructor associated with the key on the value being replaced.

The application shall ensure that the *tss_get*() and *tss_set*() functions are only called with a value for *key* that was returned by a call to *tss_create*() before the thread commenced executing destructors.

The effect of calling *tss_get*() or *tss_set*() after *key* has been deleted with *tss_delete*() is undefined.

[CX]Both *tss_get*() and *tss_set*() can be called from a thread-specific data destructor function. A call to *tss_get*() for the thread-specific data key being destroyed shall return a null pointer, unless the value is changed (after the destructor starts) by a call to *tss_set*(). Calling *tss_set*() from a thread-specific data destructor function may result either in lost storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

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

RETURN VALUE

The *tss_get*() function shall return the value for the current thread. If no thread-specific data value is associated with *key*, then a null pointer shall be returned.

The *tss_set*() function shall return `thrd_success` on success or `thrd_error` if the request could not be honored.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_getspecific, *tss_create*

XBD **<threads.h>**

CHANGE HISTORY

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

Ref 7.31.11 para 2

On page 2193 line 70145 section ungetc(), change FUTURE DIRECTIONS from:

None.

to:

> The ISO C standard states that the use of *ungetc*() on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction between binary and text streams, so this applies to all streams. This feature may be removed in a future version of this standard.

Ref 7.29.6.3 para 1, 7.1.4 para 5
On page 2242 line 71441 section wcrtomb(), change:

> [CX]The *wcrtomb*() function need not be thread-safe if called with a NULL *ps* argument.[/CX]

to:

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

Ref 7.29.6.4 para 1, 7.1.4 para 5
On page 2266 line 72111 section wcsrtombs(), change:

> [CX]The *wcsnrtombs*() and *wcsrtombs*() functions need not be thread-safe if called with a NULL *ps* argument.[/CX]

to:

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

Ref 7.22.7 para 1, 7.1.4 para 5
On page 2292 line 72879 section wctomb(), change:

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

to:

> The *wctomb*() function need not be thread-safe; however, it shall avoid data races with all other functions.

Changes to XCU

Ref 7.22.2
On page 2333 line 74167 section 1.1.2.2 Mathematical Functions, change:

Section 7.20.2, Pseudo-Random Sequence Generation Functions

to:

Section 7.22.2, Pseudo-Random Sequence Generation Functions

Ref 6.10.8.1 para 1 (__STDC_VERSION__)
On page 2542 line 82220 section c99, rename the c99 page to c17.

Ref 7.26
On page 2545 line 82375 section c99 (now c17), change:

... , **<spawn.h>**, **<sys/socket.h>**, ...

to:

... , **<spawn.h>**, **<sys/socket.h>**, **<threads.h>**, ...

Ref 7.26
On page 2545 line 82382 section c99 (now c17), change:

This option shall make available all interfaces referenced in **<pthread.h>** and *pthread_kill*() and *pthread_sigmask*() referenced in **<signal.h>**.

to:

This option shall make available all interfaces referenced in **<pthread.h>** and **<threads.h>**, and also *pthread_kill*() and *pthread_sigmask*() referenced in **<signal.h>**.

Ref 6.10.8.1 para 1 (__STDC_VERSION__)
On page 2552-2553 line 82641-82677 section c99 (now c17), change CHANGE HISTORY to:

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

Changes to XRAT

Ref G.1 para 1
On page 3483 line 117680 section A.1.7.1 Codes, add a new tagged paragraph:

MXC This margin code is used to denote functionality related to the IEC 60559 Complex Floating-Point option. ~~This functionality is mandated by the ISO C standard for IEC 60559 implementations that support **<complex.h>**.~~

Ref (none)
On page 3489 line 117909 section A.3 Definitions (Byte), change:

alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types are now defined.

to:

alignment with the ISO/IEC 9899: 1999 standard, where the **intN_t** types were first defined.

Ref 5.1.2.4, 7.17.3
On page 3515 line 118946 section A.4.12 Memory Synchronization, change:

A.4.12 Memory Synchronization

to:

A.4.12 Memory Ordering and Synchronization

A.4.12.1 Memory Ordering

There is no additional rationale provided for this section.

A.4.12.2 Memory Synchronization

Ref 6.10.8.1 para 1 (__STDC_VERSION__)
On page 3556 line 120684 section A.12.2 Utility Syntax Guidelines, change:

Thus, they had to devise a new name, *c89* (now superseded by *c99*), rather than …

to:

Thus, they had to devise a new name, *c89* (subsequently superseded by *c99* and now by *c17*), rather than …

Ref K.3.1.1
On page 3567 line 121053 section B.2.2.1 POSIX.1 Symbols, add a new unnumbered subsection:

The __STDC_WANT_LIB_EXT1__ Feature Test Macro

The ISO C standard specifies the feature test macro __STDC_WANT_LIB_EXT1__ as the announcement mechanism for the application that it requires functionality from Annex K. It specifies that the symbols specified in Annex K (if supported) are made visible when __STDC_WANT_LIB_EXT1__ is 1 and are not made visible when it is 0, but leaves it unspecified whether they are made visible when __STDC_WANT_LIB_EXT1__ is undefined. POSIX.1 requires that they are not made visible when the macro is undefined (except for those symbols that are already explicitly allowed to be visible through the definition of _POSIX_C_SOURCE or _XOPEN_SOURCE, or both).

POSIX.1 does not include the interfaces specified in Annex K of the ISO C standard, but allows the symbols to be made visible in headers when requested by the application in order that applications can use symbols from Annex K and symbols from POSIX.1 in the same translation unit.

Ref 6.10.3.4
On page 3570 line 121176 section B.2.2.2 The Name Space, change:

as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C standard

to:

as described for macros that expand to their own name as in Section 6.10.3.4 of the ISO C standard

Ref 7.5 para 2
On page 3571 line 121228-121243 section B.2.3 Error Numbers, change:

> The ISO C standard requires that *errno* be an assignable lvalue. Originally, …
> […]
> … using the return value for a mixed purpose was judged to be of limited use and error prone.

to:

> The original ISO C standard just required that *errno* be an modifiable lvalue. Since the introduction of threads in 2011, the ISO C standard has instead required that *errno* be a macro which expands to a modifiable lvalue that has thread local storage duration.

Ref 7.26
On page 3575 line 121390 section B.2.3 Error Numbers, change:

> In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a special case since it will never occur.

to:

> In particular, applications calling blocking interfaces need not handle any possible [EINTR] return as a special case since it will never occur. In the case of threads functions in **<threads.h>**, the requirement is stated in terms of the call not being affected if the calling thread executes a signal handler during the call, since these functions return errors in a different way and cannot distinguish an [EINTR] condition from other error conditions.

Ref (none)
On page 3733 line 128128 section C.2.6.4 Arithmetic Expansion, change:

> Although the ISO/IEC 9899: 1999 standard now requires support for …

to:

> Although the ISO C standard requires support for …

Ref 7.17
On page 3789 line 129986 section E.1 Subprofiling Option Groups, change:

> by collecting sets of related functions

to:

> by collecting sets of related functions and generic functions

Ref 7.22.3.1, 7.27.2.5, 7.22.4
On page 3789, 3792 line 130022-130032, 130112-130114 section E.1 Subprofiling Option Groups, add new functions (in sorted order) to the existing groups as indicated:

POSIX_C_LANG_SUPPORT
aligned_alloc(), *timespec_get*()

POSIX_MULTI_PROCESS
at_quick_exit(), *quick_exit*()

Ref 7.17
On page 3789 line 129991 section E.1 Subprofiling Option Groups, add:

POSIX_C_LANG_ATOMICS: ISO C Atomic Operations
atomic_compare_exchange_strong(), *atomic_compare_exchange_strong_explicit*(), *atomic_compare_exchange_weak*(), *atomic_compare_exchange_weak_explicit*(), *atomic_exchange*(), *atomic_exchange_explicit*(), *atomic_fetch_add*(), *atomic_fetch_add_explicit*(), *atomic_fetch_and*(), *atomic_fetch_and_explicit*(), *atomic_fetch_or*(), *atomic_fetch_or_explicit*(), *atomic_fetch_sub*(), *atomic_fetch_sub_explicit*(), *atomic_fetch_xor*(), *atomic_fetch_xor_explicit*(), *atomic_flag_clear*(), *atomic_flag_clear_explicit*(), *atomic_flag_test_and_set*(), *atomic_flag_test_and_set_explicit*(), *atomic_init*(), *atomic_is_lock_free*(), *atomic_load*(), *atomic_load_explicit*(), *atomic_signal_fence*(), *atomic_thread_fence*(), *atomic_store*(), *atomic_store_explicit*(), *kill_dependency*()

Ref 7.26
On page 3790 line 1300349 section E.1 Subprofiling Option Groups, add:

POSIX_C_LANG_THREADS: ISO C Threads
call_once(), *cnd_broadcast*(), *cnd_signal*(), *cnd_destroy*(), *cnd_init*(), *cnd_timedwait*(), *cnd_wait*(), *mtx_destroy*(), *mtx_init*(), *mtx_lock*(), *mtx_timedlock*(), *mtx_trylock*(), *mtx_unlock*(), *thrd_create*(), *thrd_current*(), *thrd_detach*(), *thrd_equal*(), *thrd_exit*(), *thrd_join*(), *thrd_sleep*(), *thrd_yield*(), *tss_create*(), *tss_delete*(), *tss_get*(), *tss_set*()

POSIX_C_LANG_UCHAR: ISO C Unicode Utilities
c16rtomb(), *c32rtomb*(), *mbrtoc16*(), *mbrtoc32*()