

# Xenomai POSIX skin API Reference Manual

## 2.3.50

Generated by Doxygen 1.4.6

Sun May 27 23:16:12 2007



# Contents

<b>1</b>	<b>Xenomai POSIX skin API Module Index</b>	<b>1</b>
1.1	Xenomai POSIX skin API Modules . . . . .	1
<b>2</b>	<b>Xenomai POSIX skin API File Index</b>	<b>3</b>
2.1	Xenomai POSIX skin API File List . . . . .	3
<b>3</b>	<b>Xenomai POSIX skin API Module Documentation</b>	<b>5</b>
3.1	Thread cancellation. . . . .	5
3.2	Clocks and timers services. . . . .	10
3.3	Condition variables services. . . . .	17
3.4	Interruptions management services. . . . .	24
3.5	POSIX skin. . . . .	27
3.6	Message queues services. . . . .	29
3.7	Mutex services. . . . .	37
3.8	Threads scheduling services. . . . .	46
3.9	Semaphores services. . . . .	50
3.10	Shared memory services. . . . .	56
3.11	Signals services. . . . .	62
3.12	Threads management services. . . . .	70
3.13	Thread creation attributes. . . . .	77
3.14	Thread-specific data. . . . .	88
<b>4</b>	<b>Xenomai POSIX skin API File Documentation</b>	<b>91</b>
4.1	ksrc/skins/posix/syscall.c File Reference . . . . .	91



# Chapter 1

## Xenomai POSIX skin API Module Index

### 1.1 Xenomai POSIX skin API Modules

Here is a list of all modules:

POSIX skin. . . . .	27
Clocks and timers services. . . . .	10
Condition variables services. . . . .	17
Interruptions management services. . . . .	24
Message queues services. . . . .	29
Mutex services. . . . .	37
Semaphores services. . . . .	50
Shared memory services. . . . .	56
Signals services. . . . .	62
Threads management services. . . . .	70
Thread cancellation. . . . .	5
Threads scheduling services. . . . .	46
Thread creation attributes. . . . .	77
Thread-specific data. . . . .	88



## Chapter 2

# Xenomai POSIX skin API File Index

### 2.1 Xenomai POSIX skin API File List

Here is a list of all documented files with brief descriptions:

ksrc/skins/posix/ <b>cancel.h</b>	??
ksrc/skins/posix/ <b>cond.h</b>	??
ksrc/skins/posix/ <b>internal.h</b>	??
ksrc/skins/posix/ <b>intr.h</b>	??
ksrc/skins/posix/ <b>mq.h</b>	??
ksrc/skins/posix/ <b>mutex.h</b>	??
ksrc/skins/posix/ <b>registry.h</b>	??
ksrc/skins/posix/ <b>sem.h</b>	??
ksrc/skins/posix/ <b>shm.h</b>	??
ksrc/skins/posix/ <b>sig.h</b>	??
ksrc/skins/posix/ <b>syscall.c</b> (This file is part of the Xenomai project )	91
ksrc/skins/posix/ <b>thread.h</b>	??
ksrc/skins/posix/ <b>timer.h</b>	??
ksrc/skins/posix/ <b>tsd.h</b>	??





## Chapter 3

# Xenomai POSIX skin API Module Documentation

### 3.1 Thread cancellation.

#### 3.1.1 Detailed Description

Thread cancellation.

Cancellation is the mechanism by which a thread can terminate the execution of a Xenomai POSIX skin thread (created with [pthread\\_create\(\)](#)). More precisely, a thread can send a cancellation request to a Xenomai POSIX skin thread and depending on its cancelability type (see [pthread\\_setcanceltype\(\)](#)) and state (see [pthread\\_setcancelstate\(\)](#)), the target thread can then either ignore the request, honor it immediately, or defer it till it reaches a cancellation point. When threads are first created by [pthread\\_create\(\)](#), they always defer cancellation requests.

When a thread eventually honors a cancellation request, it behaves as if [pthread\\_exit\(PTHREAD\\_CANCELED\)](#) was called. All cleanup handlers are executed in reverse order, finalization functions for thread-specific data are called, and finally the thread stops executing. If the canceled thread was joinable, the return value PTHREAD\_CANCELED is provided to whichever thread calls [pthread\\_join\(\)](#) on it. See [pthread\\_exit\(\)](#) for more information.

Cancellation points are the points where the thread checks for pending cancellation requests and performs them. The POSIX threads functions [pthread\\_join\(\)](#), [pthread\\_cond\\_wait\(\)](#), [pthread\\_cond\\_timedwait\(\)](#), [pthread\\_testcancel\(\)](#), [sem\\_wait\(\)](#), [sem\\_timedwait\(\)](#), [sigwait\(\)](#), [sigwaitinfo\(\)](#) and [sigtimedwait\(\)](#) are cancellation points.

See also:

[Specification.](#)

#### Functions

- [int pthread\\_cancel](#) (pthread\_t thread)  
*Cancel a thread.*
- [void pthread\\_cleanup\\_push](#) (cleanup\_routine\_t \*routine, void \*arg)  
*Register a cleanup handler to be executed at the time of cancellation.*

- void [pthread\\_cleanup\\_pop](#) (int *execute*)  
*Unregister the last registered cleanup handler.*
- int [pthread\\_setcanceltype](#) (int *type*, int \**oldtype\_ptr*)  
*Set cancelability type of the current thread.*
- int [pthread\\_setcancelstate](#) (int *state*, int \**oldstate\_ptr*)  
*Set cancelability state of the current thread.*
- void [pthread\\_testcancel](#) (void)  
*Test if a cancellation request is pending.*

### 3.1.2 Function Documentation

#### 3.1.2.1 int [pthread\\_cancel](#) ([pthread\\_t](#) *thread*)

Cancel a thread.

This service sends a cancellation request to the thread *thread* and returns immediately. Depending on the target thread cancelability state (see [pthread\\_setcancelstate\(\)](#)) and type (see [pthread\\_setcanceltype\(\)](#)), its termination is either immediate, deferred or ignored.

When the cancellation request is handled and before the thread is terminated, the cancellation cleanup handlers (registered with the [pthread\\_cleanup\\_push\(\)](#) service) are called, then the thread-specific data destructor functions (registered with [pthread\\_key\\_create\(\)](#)).

##### Returns:

- 0 on success;
- an error number if:
  - ESRCH, the thread *thread* was not found.

##### See also:

[Specification.](#)

#### 3.1.2.2 void [pthread\\_cleanup\\_pop](#) (int *execute*)

Unregister the last registered cleanup handler.

If the calling thread is a Xenomai POSIX skin thread (i.e. created with [pthread\\_create\(\)](#)), this service unregisters the last routine which was registered with [pthread\\_cleanup\\_push\(\)](#) and call it if *execute* is not null.

If the caller context is invalid (not a Xenomai POSIX skin thread), this service has no effect.

This service may be called at any place, but for maximal portability, should only called in the same lexical scope as the matching call to [pthread\\_cleanup\\_push\(\)](#).

##### Parameters:

- execute* if non zero, the last registered cleanup handler should be executed before it is un-registered.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread,
- Xenomai POSIX skin user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.1.2.3 void pthread\_cleanup\_push (cleanup\_routine\_t \* routine, void \* arg)**

Register a cleanup handler to be executed at the time of cancellation.

This service registers the given *routine* to be executed at the time of cancellation of the calling thread, if this thread is a Xenomai POSIX skin thread (i.e. created with the [pthread\\_create\(\)](#) service). If the caller context is invalid (not a Xenomai POSIX skin thread), this service has no effect.

If allocation from the system heap fails (because the system heap size is too small), this service fails silently.

The routines registered with this service get called in LIFO order when the calling thread calls [pthread\\_exit\(\)](#) or is canceled, or when it calls the [pthread\\_cleanup\\_pop\(\)](#) service with a non null argument.

**Parameters:**

- routine* the cleanup routine to be registered;
- arg* the argument associated with this routine.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread,
- Xenomai POSIX skin user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.1.2.4 int pthread\_setcancelstate (int state, int \* oldstate\_ptr)**

Set cancelability state of the current thread.

This service atomically sets the cancelability state of the calling thread and returns its previous value at the address *oldstate\_ptr*, if the calling thread is a Xenomai POSIX skin thread (i.e. created with the [pthread\\_create](#) service).

The cancelability state of a POSIX thread may be:

- PTHREAD\_CANCEL\_ENABLE, meaning that cancellation requests will be handled if received;
- PTHREAD\_CANCEL\_DISABLE, meaning that cancellation requests will not be handled if received.

**Parameters:**

- state* new cancelability state of the calling thread;

*oldstate\_ptr* address where the old cancelability state will be stored on success.

**Returns:**

0 on success;

an error number if:

- EINVAL, *state* is not a valid cancelability state;
- EPERM, the caller context is invalid.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread,
- Xenomai POSIX skin user-space thread (switches to primary mode).

**See also:**

[Specification](#).

### 3.1.2.5 int pthread\_setcanceltype (int *type*, int \* *oldtype\_ptr*)

Set cancelability type of the current thread.

This service atomically sets the cancelability type of the calling thread, and return its previous value at the address *oldtype\_ptr*, if this thread is a Xenomai POSIX skin thread (i.e. was created with the [pthread\\_create\(\)](#) service).

The cancelability type of a POSIX thread may be:

- PTHREAD\_CANCEL\_DEFERRED, meaning that cancellation requests are only handled in services which are cancellation points;
- PTHREAD\_CANCEL\_ASYNCHRONOUS, meaning that cancellation requests are handled as soon as they are sent.

**Parameters:**

*type* new cancelability type of the calling thread;

*oldtype\_ptr* address where the old cancelability type will be stored on success.

**Returns:**

0 on success;

an error number if:

- EINVAL, *type* is not a valid cancelability type;
- EPERM, the caller context is invalid.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread,
- Xenomai POSIX skin user-space thread (switches to primary mode).

**See also:**

[Specification](#).

### 3.1.2.6 void pthread\_testcancel (void)

Test if a cancellation request is pending.

This function creates a cancellation point if the calling thread is a Xenomai POSIX skin thread (i.e. created with the [pthread\\_create\(\)](#) service).

This function is a cancellation point. It has no effect if cancellation is disabled.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread,
- Xenomai POSIX skin user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

## 3.2 Clocks and timers services.

### 3.2.1 Detailed Description

Clocks and timers services.

Xenomai POSIX skin supports two clocks:

CLOCK\_REALTIME maps to the nucleus system clock, keeping time as the amount of time since the Epoch, with a resolution of one system clock tick.

CLOCK\_MONOTONIC maps to an architecture-dependent high resolution counter, so is suitable for measuring short time intervals. However, when used for sleeping (with [clock\\_nanosleep\(\)](#)), the CLOCK\_MONOTONIC clock has a resolution of one system clock tick, like the CLOCK\_REALTIME clock.

Setting any of the two clocks with [clock\\_settime\(\)](#) is currently not supported.

Timer objects may be created with the [timer\\_create\(\)](#) service using either of the two clocks, but the resolution of these timers is one system clock tick, as is the case for [clock\\_nanosleep\(\)](#).

**Note:**

The duration of the POSIX clock tick depends on the active time base (configurable at compile-time with the constant `CONFIG_XENO_OPT_POSIX_PERIOD`, and at run-time with the `xeno_posix` module parameter `tick_arg`). When the time base is aperiodic (which is the default) the system clock tick is one nanosecond.

**See also:**

[Specification](#).

## Functions

- int [clock\\_getres](#) (clockid\_t clock\_id, struct timespec \*res)  
*Get the resolution of the specified clock.*
- int [clock\\_gettime](#) (clockid\_t clock\_id, struct timespec \*tp)  
*Read the specified clock.*
- int [clock\\_settime](#) (clockid\_t clock\_id, const struct timespec \*tp)  
*Set the specified clock.*
- int [clock\\_nanosleep](#) (clockid\_t clock\_id, int flags, const struct timespec \*rqtp, struct timespec \*rmtp)  
*Sleep some amount of time.*
- int [nanosleep](#) (const struct timespec \*rqtp, struct timespec \*rmtp)  
*Sleep some amount of time.*
- int [timer\\_create](#) (clockid\_t clockid, const struct sigevent \*\_\_restrict\_\_ evp, timer\_t \*\_\_restrict\_\_ timerid)  
*Create a timer object.*
- int [timer\\_delete](#) (timer\_t timerid)

*Delete a timer object.*

- `int timer_settime(timer_t timerid, int flags, const struct itimerspec *__restrict__ value, struct itimerspec *__restrict__ ovalue)`

*Start or stop a timer.*

- `int timer_gettime(timer_t timerid, struct itimerspec *value)`

*Get timer next expiration date and reload value.*

- `int timer_getoverrun(timer_t timerid)`

*Get expiration overruns count since the most recent timer expiration signal delivery.*

### 3.2.2 Function Documentation

#### 3.2.2.1 `int clock_getres(clockid_t clock_id, struct timespec * res)`

Get the resolution of the specified clock.

This service returns, at the address *res*, if it is not *NULL*, the resolution of the clock *clock\_id*.

For both `CLOCK_REALTIME` and `CLOCK_MONOTONIC`, this resolution is the duration of one system clock tick. No other clock is supported.

##### Parameters:

*clock\_id* clock identifier, either `CLOCK_REALTIME` or `CLOCK_MONOTONIC`;

*res* the address where the resolution of the specified clock will be stored on success.

##### Return values:

*0* on success;

*-1* with *errno* set if:

- `EINVAL`, *clock\_id* is invalid;

##### See also:

[Specification.](#)

#### 3.2.2.2 `int clock_gettime(clockid_t clock_id, struct timespec * tp)`

Read the specified clock.

This service returns, at the address *tp* the current value of the clock *clock\_id*. If *clock\_id* is:

- `CLOCK_REALTIME`, the clock value represents the amount of time since the Epoch, with a precision of one system clock tick;
- `CLOCK_MONOTONIC`, the clock value is given by an architecture-dependent high resolution counter, with a precision independent from the system clock tick duration.

##### Parameters:

*clock\_id* clock identifier, either `CLOCK_REALTIME` or `CLOCK_MONOTONIC`;

*tp* the address where the value of the specified clock will be stored.

**Return values:**

- 0 on success;
- 1 with *errno* set if:
  - EINVAL, *clock\_id* is invalid.

**See also:**

[Specification.](#)

### 3.2.2.3 int clock\_nanosleep (clockid\_t *clock\_id*, int *flags*, const struct timespec \* *rqtp*, struct timespec \* *rmtp*)

Sleep some amount of time.

This service suspends the calling thread until the wakeup time specified by *rqtp*, or a signal is delivered to the caller. If the flag TIMER\_ABSTIME is set in the *flags* argument, the wakeup time is specified as an absolute value of the clock *clock\_id*. If the flag TIMER\_ABSTIME is not set, the wakeup time is specified as a time interval.

If this service is interrupted by a signal, the flag TIMER\_ABSTIME is not set, and *rmtp* is not NULL, the time remaining until the specified wakeup time is returned at the address *rmtp*.

The resolution of this service is one system clock tick.

**Parameters:**

*clock\_id* clock identifier, either CLOCK\_REALTIME or CLOCK\_MONOTONIC.

*flags* one of:

- 0 meaning that the wakeup time *rqtp* is a time interval;
- TIMER\_ABSTIME, meaning that the wakeup time is an absolute value of the clock *clock\_id*.

*rqtp* address of the wakeup time.

*rmtp* address where the remaining time before wakeup will be stored if the service is interrupted by a signal.

**Returns:**

- 0 on success;
- an error number if:
  - EPERM, the caller context is invalid;
  - ENOTSUP, the specified clock is unsupported;
  - EINVAL, the specified wakeup time is invalid;
  - EINTR, this service was interrupted by a signal.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)



**3.2.2.4 int clock\_settime (clockid\_t *clock\_id*, const struct timespec \* *tp*)**

Set the specified clock.

This service is not supported.

See also:

[Specification.](#)

**3.2.2.5 int nanosleep (const struct timespec \* *rqtp*, struct timespec \* *rmtp*)**

Sleep some amount of time.

This service suspends the calling thread until the wakeup time specified by *rqtp*, or a signal is delivered. The wakeup time is specified as a time interval.

If this service is interrupted by a signal and *rmtp* is not *NULL*, the time remaining until the specified wakeup time is returned at the address *rmtp*.

The resolution of this service is one system clock tick.

**Parameters:**

*rqtp* address of the wakeup time.

*rmtp* address where the remaining time before wakeup will be stored if the service is interrupted by a signal.

**Return values:**

0 on success;

-1 with *errno* set if:

- EPERM, the caller context is invalid;
- EINVAL, the specified wakeup time is invalid;
- EINTR, this service was interrupted by a signal.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

See also:

[Specification.](#)

**3.2.2.6 int timer\_create (clockid\_t *clockid*, const struct sigevent \* \_\_restrict *evp*, timer\_t \* \_\_restrict *timerid*)**

Create a timer object.

This service creates a time object using the clock *clockid*.

If *evp* is not *NULL*, it describes the notification mechanism used on timer expiration. Only notification via signal delivery is supported (member *sigev\_notify* of *evp* set to *SIGEV\_SIGNAL*). The signal will be sent to the thread starting the timer with the [timer\\_settime\(\)](#) service. If *evp* is *NULL*, the *SIGALRM* signal will be used.

Note that signals sent to user-space threads will cause them to switch to secondary mode.

If this service succeeds, an identifier for the created timer is returned at the address *timerid*. The timer is unarmed until started with the [timer\\_settime\(\)](#) service.

**Parameters:**

- clockid* clock used as a timing base;
- evp* description of the asynchronous notification to occur when the timer expires;
- timerid* address where the identifier of the created timer will be stored on success.

**Return values:**

- 0 on success;
- 1 with *errno* set if:
  - EINVAL, the clock *clockid* is invalid;
  - EINVAL, the member *sige\_v\_notify* of the **sigevent** structure at the address *evp* is not SIGEV\_SIGNAL;
  - EINVAL, the member *sige\_v\_signo* of the **sigevent** structure is an invalid signal number;
  - EAGAIN, the maximum number of timers was exceeded, recompile with a larger value.

**See also:**

[Specification.](#)

### 3.2.2.7 int timer\_delete (timer\_t timerid)

Delete a timer object.

This service deletes the timer *timerid*.

**Parameters:**

- timerid* identifier of the timer to be removed;

**Return values:**

- 0 on success;
- 1 with *errno* set if:
  - EINVAL, *timerid* is invalid.

**See also:**

[Specification.](#)

### 3.2.2.8 int timer\_getoverrun (timer\_t timerid)

Get expiration overruns count since the most recent timer expiration signal delivery.

This service returns *timerid* expiration overruns count since the most recent timer expiration signal delivery. If this count is more than *DELAYTIMER\_MAX* expirations, *DELAYTIMER\_MAX* is returned.

**Parameters:**

- timerid* Timer identifier.

**Returns:**

- the overruns count on success;
- 1 with *errno* set if:
  - EINVAL, *timerid* is invalid.

**See also:**

[Specification.](#)

**3.2.2.9 int timer\_gettime (timer\_t timerid, struct itimerspec \* value)**

Get timer next expiration date and reload value.

This service stores, at the address *value*, the expiration date (member *it\_value*) and reload value (member *it\_interval*) of the timer *timerid*. The values are returned as time intervals, and as multiples of the system clock tick duration (see note in section [Clocks and timers services](#) for details on the duration of the system clock tick). If the timer was not started, the returned members *it\_value* and *it\_interval* of *value* are zero.

**Parameters:**

- timerid* timer identifier;
- value* address where the timer expiration date and reload value are stored on success.

**Return values:**

- 0 on success;
- 1 with *errno* set if:
  - EINVAL, *timerid* is invalid.

**See also:**

[Specification.](#)

**3.2.2.10 int timer\_settime (timer\_t timerid, int flags, const struct itimerspec \*\_\_restrict\_\_ value, struct itimerspec \*\_\_restrict\_\_ ovalue)**

Start or stop a timer.

This service sets a timer expiration date and reload value of the timer *timerid*. If *ovalue* is not *NULL*, the current expiration date and reload value are stored at the address *ovalue* as with [timer\\_gettime\(\)](#).

If the member *it\_value* of the **itimerspec** structure at *value* is zero, the timer is stopped, otherwise the timer is started. If the member *it\_interval* is not zero, the timer is periodic. The current thread must be a POSIX skin thread (created with [pthread\\_create\(\)](#)) and will be notified via signal of timer expirations. Note that these notifications will cause user-space threads to switch to secondary mode.

When starting the timer, if *flags* is *TIMER\_ABSTIME*, the expiration value is interpreted as an absolute date of the clock passed to the [timer\\_create\(\)](#) service. Otherwise, the expiration value is interpreted as a time interval.

Expiration date and reload value are rounded to an integer count of system clock ticks (see note in section [Clocks and timers services](#) for details on the duration of the system tick).

**Parameters:**

- timerid* identifier of the timer to be started or stopped;
- flags* one of 0 or `TIMER_ABSTIME`;
- value* address where the specified timer expiration date and reload value are read;
- ovalue* address where the specified timer previous expiration date and reload value are stored if not `NULL`.

**Return values:**

- 0 on success;
- 1 with *errno* set if:
- `EPERM`, the caller context is invalid;
  - `EINVAL`, the specified timer identifier, expiration date or reload value is invalid.

**Valid contexts:**

- Xenomai kernel-space POSIX skin thread,
- kernel-space thread cancellation cleanup routine,
- Xenomai POSIX skin user-space thread (switches to primary mode),
- user-space thread cancellation cleanup routine.

**See also:**

[Specification.](#)

## 3.3 Condition variables services.

### 3.3.1 Detailed Description

Condition variables services.

A condition variable is a synchronization object that allows threads to suspend execution until some predicate on shared data is satisfied. The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

Before it can be used, a condition variable has to be initialized with `pthread_cond_init()`. An attribute object, which reference may be passed to this service, allows to select the features of the created condition variable, namely the *clock* used by the `pthread_cond_timedwait()` service (`CLOCK_REALTIME` is used by default), and whether it may be shared between several processes (it may not be shared by default, see `pthread_condattr_setpshared()`).

Note that only `pthread_cond_init()` may be used to initialize a condition variable, using the static initializer `PTHREAD_COND_INITIALIZER` is not supported.

### Functions

- `int pthread_cond_init(pthread_cond_t *cnd, const pthread_condattr_t *attr)`  
*Initialize a condition variable.*
- `int pthread_cond_destroy(pthread_cond_t *cnd)`  
*Destroy a condition variable.*
- `int pthread_cond_wait(pthread_cond_t *cnd, pthread_mutex_t *mx)`  
*Wait on a condition variable.*
- `int pthread_cond_timedwait(pthread_cond_t *cnd, pthread_mutex_t *mx, const struct timespec *abstime)`  
*Wait a bounded time on a condition variable.*
- `int pthread_cond_signal(pthread_cond_t *cnd)`  
*Signal a condition variable.*
- `int pthread_cond_broadcast(pthread_cond_t *cnd)`  
*Broadcast a condition variable.*
- `int pthread_condattr_init(pthread_condattr_t *attr)`  
*Initialize a condition variable attributes object.*
- `int pthread_condattr_destroy(pthread_condattr_t *attr)`  
*Destroy a condition variable attributes object.*
- `int pthread_condattr_getclock(const pthread_condattr_t *attr, clockid_t *clk_id)`

*Get the clock selection attribute from a condition variable attributes object.*

- int [pthread\\_condattr\\_setclock](#) (pthread\_condattr\_t \*attr, clockid\_t clk\_id)  
*Set the clock selection attribute of a condition variable attributes object.*
- int [pthread\\_condattr\\_getpshared](#) (const pthread\_condattr\_t \*attr, int \*pshared)  
*Get the process-shared attribute from a condition variable attributes object.*
- int [pthread\\_condattr\\_setpshared](#) (pthread\_condattr\_t \*attr, int pshared)  
*Set the process-shared attribute of a condition variable attributes object.*

### 3.3.2 Function Documentation

#### 3.3.2.1 int pthread\_cond\_broadcast (pthread\_cond\_t \* cnd)

Broadcast a condition variable.

This service unblocks all threads blocked on the condition variable *cnd*.

**Parameters:**

*cnd* the condition variable to be signalled.

**Returns:**

- 0 on succes,  
an error number if:
- EINVAL, the condition variable is invalid.

**See also:**

[Specification.](#)

#### 3.3.2.2 int pthread\_cond\_destroy (pthread\_cond\_t \* cnd)

Destroy a condition variable.

This service destroys the condition variable *cnd*, if no thread is currently blocked on it. The condition variable becomes invalid for all condition variable services (they all return the EINVAL error) except [pthread\\_cond\\_init\(\)](#).

**Parameters:**

*cnd* the condition variable to be destroyed.

**Returns:**

- 0 on succes,  
an error number if:
- EINVAL, the condition variable *cnd* is invalid;
  - EBUSY, some thread is currently using the condition variable.

**See also:**

[Specification.](#)

### 3.3.2.3 `int pthread_cond_init (pthread_cond_t * cnd, const pthread_condattr_t * attr)`

Initialize a condition variable.

This service initializes the condition variable *cnd*, using the condition variable attributes object *attr*. If *attr* is *NULL* or this service is called from user-space, default attributes are used (see [pthread\\_condattr\\_init\(\)](#)).

**Parameters:**

- cnd* the condition variable to be initialized;
- attr* the condition variable attributes object.

**Returns:**

- 0 on succes,
- an error number if:
  - EINVAL, the condition variable attributes object *attr* is invalid or uninitialized;
  - EBUSY, the condition variable *cnd* was already initialized;
  - ENOMEM, insufficient memory exists in the system heap to initialize the condition variable, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ.

**See also:**

[Specification.](#)

### 3.3.2.4 `int pthread_cond_signal (pthread_cond_t * cnd)`

Signal a condition variable.

This service unblocks one thread blocked on the condition variable *cnd*.

If more than one thread is blocked on the specified condition variable, the highest priority thread is unblocked.

**Parameters:**

- cnd* the condition variable to be signalled.

**Returns:**

- 0 on succes,
- an error number if:
  - EINVAL, the condition variable is invalid.

**See also:**

[Specification.](#)

### 3.3.2.5 `int pthread_cond_timedwait (pthread_cond_t * cnd, pthread_mutex_t * mx, const struct timespec * abstime)`

Wait a bounded time on a condition variable.

This service is equivalent to [pthread\\_cond\\_wait\(\)](#), except that the calling thread remains blocked on the condition variable *cnd* only until the timeout specified by *abstime* expires.

The timeout *abstime* is expressed as an absolute value of the *clock* attribute passed to [pthread\\_cond\\_init\(\)](#). By default, *CLOCK\_REALTIME* is used.

**Parameters:**

- cnd* the condition variable to wait for;
- mx* the mutex associated with *cnd*;
- abstime* the timeout, expressed as an absolute value of the clock attribute passed to [pthread\\_cond\\_init\(\)](#).

**Returns:**

- 0 on success,
- an error number if:
  - EINVAL, the specified condition variable, mutex or timeout is invalid;
  - EPERM, the caller context is invalid;
  - EINVAL, another thread is currently blocked on *cnd* using another mutex than *mx*;
  - EPERM, the specified mutex is not owned by the caller;
  - ETIMEDOUT, the specified timeout expired.

**Valid contexts:**

- Xenomai kernel-space thread;
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification](#).

### 3.3.2.6 `int pthread_cond_wait(pthread_cond_t * cnd, pthread_mutex_t * mx)`

Wait on a condition variable.

This service atomically unlocks the mutex *mx*, and block the calling thread until the condition variable *cnd* is signalled using [pthread\\_cond\\_signal\(\)](#) or [pthread\\_cond\\_broadcast\(\)](#). When the condition is signaled, this service re-acquire the mutex before returning.

Spurious wakeups occur if a signal is delivered to the blocked thread, so, an application should not assume that the condition changed upon successful return from this service.

Even if the mutex *mx* is recursive and its recursion count is greater than one on entry, it is unlocked before blocking the caller, and the recursion count is restored once the mutex is re-acquired by this service before returning.

Once a thread is blocked on a condition variable, a dynamic binding is formed between the condition variable *cnd* and the mutex *mx*; if another thread calls this service specifying *cnd* as a condition variable but another mutex than *mx*, this service returns immediately with the EINVAL status.

This service is a cancellation point for Xenomai POSIX skin threads (created with the [pthread\\_create\(\)](#) service). When such a thread is cancelled while blocked in a call to this service, the mutex *mx* is re-acquired before the cancellation cleanup handlers are called.

**Parameters:**

- cnd* the condition variable to wait for;
- mx* the mutex associated with *cnd*.

**Returns:**

- 0 on success,
- an error number if:



- EINVAL, the specified condition variable or mutex is invalid;
- EPERM, the caller context is invalid;
- EINVAL, another thread is currently blocked on *cnd* using another mutex than *mx*;
- EPERM, the specified mutex is not owned by the caller.

**Valid contexts:**

- Xenomai kernel-space thread;
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.3.2.7 int pthread\_condattr\_destroy (pthread\_condattr\_t \* attr)**

Destroy a condition variable attributes object.

This service destroys the condition variable attributes object *attr*. The object becomes invalid for all condition variable services (they all return EINVAL) except [pthread\\_condattr\\_init\(\)](#).

**Parameters:**

*attr* the initialized mutex attributes object to be destroyed.

**Returns:**

- 0 on success;
- an error number if:
  - EINVAL, the mutex attributes object *attr* is invalid.

**See also:**

[Specification.](#)

**3.3.2.8 int pthread\_condattr\_getclock (const pthread\_condattr\_t \* attr, clockid\_t \* clk\_id)**

Get the clock selection attribute from a condition variable attributes object.

This service stores, at the address *clk\_id*, the value of the *clock* attribute in the condition variable attributes object *attr*.

See [pthread\\_cond\\_timedwait\(\)](#) documentation for a description of the effect of this attribute on a condition variable. The clock ID returned is *CLOCK\_REALTIME* or *CLOCK\_MONOTONIC*.

**Parameters:**

- attr* an initialized condition variable attributes object,
- clk\_id* address where the *clock* attribute value will be stored on success.

**Returns:**

- 0 on success,
- an error number if:
  - EINVAL, the attribute object *attr* is invalid.

**See also:**

[Specification.](#)

### 3.3.2.9 `int pthread_condattr_getpshared (const pthread_condattr_t * attr, int * pshared)`

Get the process-shared attribute from a condition variable attributes object.

This service stores, at the address *pshared*, the value of the *pshared* attribute in the condition variable attributes object *attr*.

The *pshared* attribute may only be one of `PTHREAD_PROCESS_PRIVATE` or `PTHREAD_PROCESS_SHARED`. See [pthread\\_condattr\\_setpshared\(\)](#) for the meaning of these two constants.

#### Parameters:

*attr* an initialized condition variable attributes object.

*pshared* address where the value of the *pshared* attribute will be stored on success.

#### Returns:

0 on success,

an error number if:

- `EINVAL`, the *pshared* address is invalid;
- `EINVAL`, the condition variable attributes object *attr* is invalid.

#### See also:

[Specification.](#)

### 3.3.2.10 `int pthread_condattr_init (pthread_condattr_t * attr)`

Initialize a condition variable attributes object.

This service initializes the condition variable attributes object *attr* with default values for all attributes. Default value are:

- for the *clock* attribute, `CLOCK_REALTIME`;
- for the *pshared* attribute `PTHREAD_PROCESS_PRIVATE`.

If this service is called specifying a condition variable attributes object that was already initialized, the attributes object is reinitialized.

#### Parameters:

*attr* the condition variable attributes object to be initialized.

#### Returns:

0 on success;

an error number if:

- `ENOMEM`, the condition variable attribute object pointer *attr* is `NULL`.

#### See also:

[Specification.](#)

### 3.3.2.11 `int pthread_condattr_setclock (pthread_condattr_t * attr, clockid_t clk_id)`

Set the clock selection attribute of a condition variable attributes object.

This service set the *clock* attribute of the condition variable attributes object *attr*.

See [pthread\\_cond\\_timedwait\(\)](#) documentation for a description of the effect of this attribute on a condition variable.

**Parameters:**

*attr* an initialized condition variable attributes object,

*clk\_id* value of the *clock* attribute, may be *CLOCK\_REALTIME* or *CLOCK\_MONOTONIC*.

**Returns:**

0 on success,

an error number if:

- EINVAL, the condition variable attributes object *attr* is invalid;
- EINVAL, the value of *clk\_id* is invalid for the *clock* attribute.

**See also:**

[Specification.](#)

### 3.3.2.12 `int pthread_condattr_setpshared (pthread_condattr_t * attr, int pshared)`

Set the process-shared attribute of a condition variable attributes object.

This service set the *pshared* attribute of the condition variable attributes object *attr*.

**Parameters:**

*attr* an initialized condition variable attributes object.

*pshared* value of the *pshared* attribute, may be one of:

- PTHREAD\_PROCESS\_PRIVATE, meaning that a condition variable created with the attributes object *attr* will only be accessible by threads within the same process as the thread that initialized the condition variable;
- PTHREAD\_PROCESS\_SHARED, meaning that a condition variable created with the attributes object *attr* will be accessible by any thread that has access to the memory where the condition variable is allocated.

**Returns:**

0 on success,

an error status if:

- EINVAL, the condition variable attributes object *attr* is invalid;
- EINVAL, the value of *pshared* is invalid.

**See also:**

[Specification.](#)

## 3.4 Interruptions management services.

### 3.4.1 Detailed Description

Interruptions management services.

The services described here allow applications written using the POSIX skin to handle interrupts, either in kernel-space or in user-space.

Note however, that it is recommended to use the standardized driver API of the RTDM skin (see `rtdm`).

### Functions

- `int pthread_intr_attach_np (pthread_intr_t *intrp, unsigned irq, int(*isr)(xnintr_t *), int(*iack)(unsigned irq))`  
*Create and attach an interrupt object.*
- `int pthread_intr_detach_np (pthread_intr_t intr)`  
*Destroy an interrupt object.*
- `int pthread_intr_control_np (pthread_intr_t intr, int cmd)`  
*Control the state of an interrupt channel.*
- `int pthread_intr_wait_np (pthread_intr_t intr, const struct timespec *to)`  
*Wait for the next interruption.*

### 3.4.2 Function Documentation

#### 3.4.2.1 `int pthread_intr_attach_np (pthread_intr_t * intrp, unsigned irq, int(*) (xnintr_t *) isr, int(*) (unsigned irq) iack)`

Create and attach an interrupt object.

This service creates and attaches an interrupt object.

#### In kernel-space:

This service installs *isr* as the handler for the interrupt *irq*. If *iack* is not null it is a custom interrupt acknowledge routine.

When called upon reception of an interrupt, the *isr* function is passed the address of an underlying `xnintr_t` object, and should use the macro `PTHREAD_IDESC()` to get the `pthread_intr_t` object. The meaning of the *isr* and *iack* function and what they should return is explained in `xnintr_init()` documentation.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*intrp* address where the created interrupt object identifier will be stored on success;

*irq* IRQ channel;  
*isr* interrupt handling routine;  
*iack* if not *NULL*, optional interrupt acknowledge routine.

#### In user-space:

The prototype of this service is :

**int pthread\_intr\_attach\_np (pthread\_intr\_t \*intrp, unsigned irq, int mode);**

This service causes the installation of a default interrupt handler which unblocks any Xenomai user-space interrupt server thread blocked in a call to [pthread\\_intr\\_wait\\_np\(\)](#), and returns a value depending on the *mode* parameter.

#### Parameters:

*intrp* and *irq* have the same meaning as in kernel-space; *mode* is a bitwise OR of the following values:

- PTHREAD\_IPROPAGATE, meaning that the interrupt should be propagated to lower priority domains;
- PTHREAD\_INOAUTOENA, meaning that the interrupt should not be automatically re-enabled.

This service is intended to be used in conjunction with the [pthread\\_intr\\_wait\\_np\(\)](#) service.

The return values are identical in kernel-space and user-space.

#### Return values:

0 on success;

-1 with *errno* set if:

- ENOMEM, insufficient memory exists in the system heap to create the interrupt object, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ;
- EINVAL, a low-level error occurred while attaching the interrupt;
- EBUSY, an interrupt handler was already registered for the irq line *irq*;
- ENOSYS, kernel-space Xenomai POSIX skin was built without support for interrupts, use RTDM or enable CONFIG\_XENO\_OPT\_POSIX\_INTR in kernel configuration.

#### 3.4.2.2 int pthread\_intr\_control\_np (pthread\_intr\_t intr, int cmd)

Control the state of an interrupt channel.

This service allow to enable or disable an interrupt channel.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*intr* identifier of the interrupt to be enabled or disabled.

*cmd* one of PTHREAD\_IENABLE or PTHREAD\_IDISABLE.

#### Return values:

0 on success;

-1 with *errno* set if:

- EINVAL, the identifier *intr* or *cmd* is invalid;
- ENOSYS, kernel-space Xenomai POSIX skin was built without support for interrupts, use RTDM or enable CONFIG\_XENO\_OPT\_POSIX\_INTR in kernel configuration.

### 3.4.2.3 int pthread\_intr\_detach\_np (pthread\_intr\_t intr)

Destroy an interrupt object.

This service destroys the interrupt object *intr*. The memory allocated for this object is returned to the system heap, so further references using the same object identifier are not guaranteed to fail.

If a user-space interrupt server is blocked in a call to [pthread\\_intr\\_wait\\_np\(\)](#), it is unblocked and the blocking service returns with an error of EIDRM.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*intr* identifier of the interrupt object to be destroyed.

#### Return values:

0 on success;

-1 with *errno* set if:

- EINVAL, the interrupt object *intr* is invalid;
- ENOSYS, kernel-space Xenomai POSIX skin was built without support for interrupts, use RTDM or enable CONFIG\_XENO\_OPT\_POSIX\_INTR in kernel configuration.

### 3.4.2.4 int pthread\_intr\_wait\_np (pthread\_intr\_t intr, const struct timespec \* to)

Wait for the next interruption.

This service is used by user-space interrupt server threads, to wait, if no interrupt is pending, for the next interrupt.

This service is a cancelation point. If a thread is canceled while blocked in a call to this service, no interruption notification is lost.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*intr* interrupt object identifier;

*to* if not *NULL*, timeout, expressed as a time interval.

#### Returns:

the number of interrupt received on success;

-1 with *errno* set if:

- EIDRM, the interrupt object was deleted;
- ETIMEDOUT, the timeout specified by *to* expired;
- EINTR, [pthread\\_intr\\_wait\\_np\(\)](#) was interrupted by a signal;
- ENOSYS, kernel-space Xenomai POSIX skin was built without support for interrupts, use RTDM or enable CONFIG\_XENO\_OPT\_POSIX\_INTR in kernel configuration.

## 3.5 POSIX skin.

### 3.5.1 Detailed Description

Xenomai POSIX skin is an implementation of a small subset of the Single Unix specification over Xenomai generic RTOS core.

The following table gives equivalence between native API services and POSIX services.

Native API services	POSIX API services
alarm	<a href="#">Clocks and timers services.</a>
cond	<a href="#">Condition variables services.</a>
event	no direct equivalence, see <a href="#">Condition variables services.</a>
native_heap	<a href="#">Shared memory services.</a>
interrupt	<a href="#">Interruptions management services.</a>
mutex	<a href="#">Mutex services.</a>
pipe	no direct equivalence, see <a href="#">Message queues services.</a>
native_queue	<a href="#">Message queues services.</a>
semaphore	<a href="#">Semaphores services.</a>
task	<a href="#">Threads management services.</a>
native_timer	<a href="#">Clocks and timers services.</a>

## Modules

- [Clocks and timers services.](#)  
*Clocks and timers services.*
- [Condition variables services.](#)  
*Condition variables services.*
- [Interruptions management services.](#)  
*Interruptions management services.*
- [Message queues services.](#)  
*Message queues services.*
- [Mutex services.](#)  
*Mutex services.*
- [Semaphores services.](#)  
*Semaphores services.*
- [Shared memory services.](#)  
*Shared memory services.*
- [Signals services.](#)  
*Signals management services.*

- [Threads management services.](#)

*Threads management services.*

- [Thread-specific data.](#)

*Thread-specific data.*



## 3.6 Message queues services.

### 3.6.1 Detailed Description

Message queues services.

A message queue allow exchanging data between real-time threads. For a POSIX message queue, maximum message length and maximum number of messages are fixed when it is created with [mq\\_open\(\)](#).

### Functions

- [mqd\\_t mq\\_open](#) (const char \*name, int oflags,...)  
*Open a message queue.*
- [int mq\\_close](#) (mqd\_t fd)  
*Close a message queue.*
- [int mq\\_unlink](#) (const char \*name)  
*Unlink a message queue.*
- [int mq\\_send](#) (mqd\_t fd, const char \*buffer, size\_t len, unsigned prio)  
*Send a message to a message queue.*
- [int mq\\_timedsend](#) (mqd\_t fd, const char \*buffer, size\_t len, unsigned prio, const struct timespec \*abs\_timeout)  
*Attempt, during a bounded time, to send a message to a message queue.*
- [ssize\\_t mq\\_receive](#) (mqd\_t fd, char \*buffer, size\_t len, unsigned \*priop)  
*Receive a message from a message queue.*
- [ssize\\_t mq\\_timedreceive](#) (mqd\_t fd, char \*\_\_restrict\_\_ buffer, size\_t len, unsigned \*\_\_restrict\_\_ priop, const struct timespec \*\_\_restrict\_\_ abs\_timeout)  
*Attempt, during a bounded time, to receive a message from a message queue.*
- [int mq\\_getattr](#) (mqd\_t fd, struct mq\_attr \*attr)  
*Get the attributes object of a message queue.*
- [int mq\\_setattr](#) (mqd\_t fd, const struct mq\_attr \*\_\_restrict\_\_ attr, struct mq\_attr \*\_\_restrict\_\_ oattr)  
*Set flags of a message queue.*
- [int mq\\_notify](#) (mqd\_t fd, const struct sigevent \*evp)  
*Register the current thread to be notified of message arrival at an empty message queue.*

## 3.6.2 Function Documentation

### 3.6.2.1 `int mq_close (mqd_t fd)`

Close a message queue.

This service closes the message queue descriptor *fd*. The message queue is destroyed only when all open descriptors are closed, and when unlinked with a call to the [mq\\_unlink\(\)](#) service.

**Parameters:**

*fd* message queue descriptor.

**Return values:**

0 on success;

-1 with *errno* set if:

- EBADF, *fd* is an invalid message queue descriptor;
- EPERM, the caller context is invalid.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- kernel-space cancellation cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode);
- user-space cancellation cleanup routine.

**See also:**

[Specification.](#)

### 3.6.2.2 `int mq_getattr (mqd_t fd, struct mq_attr * attr)`

Get the attributes object of a message queue.

This service stores, at the address *attr*, the attributes of the messages queue descriptor *fd*.

The following attributes are set:

- *mq\_flags*, flags of the message queue descriptor *fd*;
- *mq\_maxmsg*, maximum number of messages in the message queue;
- *mq\_msgsize*, maximum message size;
- *mq\_curmsgs*, number of messages currently in the queue.

**Parameters:**

*fd* message queue descriptor;

*attr* address where the message queue attributes will be stored on success.

**Return values:**

0 on success;

-1 with *errno* set if:

- EBADF, *fd* is not a valid descriptor.

**See also:**

[Specification.](#)

**3.6.2.3 int mq\_notify (mqd\_t fd, const struct sigevent \* evp)**

Register the current thread to be notified of message arrival at an empty message queue.

If *evp* is not *NULL* and is the address of a **sigevent** structure with the *sigev\_notify* member set to *SIGEV\_SIGNAL*, the current thread will be notified by a signal when a message is sent to the message queue *fd*, the queue is empty, and no thread is blocked in call to [mq\\_receive\(\)](#) or [mq\\_timedreceive\(\)](#). After the notification, the thread is unregistered.

If *evp* is *NULL* or the *sigev\_notify* member is *SIGEV\_NONE*, the current thread is unregistered.

Only one thread may be registered at a time.

If the current thread is not a Xenomai POSIX skin thread (created with [pthread\\_create\(\)](#)), this service fails.

Note that signals sent to user-space Xenomai POSIX skin threads will cause them to switch to secondary mode.

**Parameters:**

- fd* message queue descriptor;
- evp* pointer to an event notification structure.

**Return values:**

- 0 on success;
- 1 with *errno* set if:
  - *EINVAL*, *evp* is invalid;
  - *EPERM*, the caller context is invalid;
  - *EBADF*, *fd* is not a valid message queue descriptor;
  - *EBUSY*, another thread is already registered.

**Valid contexts:**

- Xenomai kernel-space POSIX skin thread,
- Xenomai user-space POSIX skin thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.6.2.4 mqd\_t mq\_open (const char \* name, int oflags, ...)**

Open a message queue.

This service establishes a connection between the message queue named *name* and the calling context (kernel-space as a whole, or user-space process).

One of the following values should be set in *oflags*:

- *O\_RDONLY*, meaning that the returned queue descriptor may only be used for receiving messages;
- *O\_WRONLY*, meaning that the returned queue descriptor may only be used for sending messages;
- *O\_RDWR*, meaning that the returned queue descriptor may be used for both sending and receiving messages.

If no message queue named *name* exists, and *oflags* has the *O\_CREAT* bit set, the message queue is created by this function, taking two more arguments:

- a *mode* argument, of type **mode\_t**, currently ignored;
- an *attr* argument, pointer to an **mq\_attr** structure, specifying the attributes of the new message queue.

If *oflags* has the two bits *O\_CREAT* and *O\_EXCL* set and the message queue already exists, this service fails.

If the *O\_NONBLOCK* bit is set in *oflags*, the **mq\_send()**, **mq\_receive()**, **mq\_timedsend()** and **mq\_timedreceive()** services return -1 with *errno* set to *EAGAIN* instead of blocking their caller.

The following arguments of the **mq\_attr** structure at the address *attr* are used when creating a message queue:

- *mq\_maxmsg* is the maximum number of messages in the queue (128 by default);
- *mq\_msgsize* is the maximum size of each message (128 by default).

*name* may be any arbitrary string, in which slashes have no particular meaning. However, for portability, using a name which starts with a slash and contains no other slash is recommended.

#### Parameters:

*name* name of the message queue to open;

*oflags* flags.

#### Returns:

a message queue descriptor on success;

-1 with *errno* set if:

- *ENAMETOOLONG*, the length of the *name* argument exceeds 64 characters;
- *EEXIST*, the bits *O\_CREAT* and *O\_EXCL* were set in *oflags* and the message queue already exists;
- *ENOENT*, the bit *O\_CREAT* is not set in *oflags* and the message queue does not exist;
- *ENOSPC*, allocation of system memory failed, or insufficient memory exists in the system heap to create the queue, try increasing *CONFIG\_XENO\_OPT\_SYS\_HEAPSZ*;
- *EPERM*, attempting to create a message queue from an invalid context;
- *EINVAL*, the *attr* argument is invalid;
- *EMFILE*, too many descriptors are currently open.

#### Valid contexts:

When creating a message queue, only the following contexts are valid:

- kernel module initialization or cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode).

#### See also:

[Specification.](#)

**3.6.2.5 ssize\_t mq\_receive (mqd\_t fd, char \* buffer, size\_t len, unsigned \* priop)**

Receive a message from a message queue.

If the message queue *fd* is not empty and if *len* is greater than the *mq\_msgsize* of the message queue, this service copies, at the address *buffer*, the queued message with the highest priority.

If the queue is empty and the flag *O\_NONBLOCK* is not set for the descriptor *fd*, the calling thread is suspended until some message is sent to the queue. If the queue is empty and the flag *O\_NONBLOCK* is set for the descriptor *fd*, this service returns immediately a value of -1 with *errno* set to *EAGAIN*.

**Parameters:**

- fd* the queue descriptor;
- buffer* the address where the received message will be stored on success;
- len* *buffer* length;
- priop* address where the priority of the received message will be stored on success.

**Returns:**

- the message length, and copy a message at the address *buffer* on success;
- 1 with no message unqueued and *errno* set if:
  - *EBADF*, *fd* is not a valid descriptor open for reading;
  - *EMSGSIZE*, the length *len* is lesser than the message queue *mq\_msgsize* attribute;
  - *EAGAIN*, the queue is empty, and the flag *O\_NONBLOCK* is set for the descriptor *fd*;
  - *EPERM*, the caller context is invalid;
  - *EINTR*, the service was interrupted by a signal.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.6.2.6 int mq\_send (mqd\_t fd, const char \* buffer, size\_t len, unsigned prio)**

Send a message to a message queue.

If the message queue *fd* is not full, this service sends the message of length *len* pointed to by the argument *buffer*, with priority *prio*. A message with greater priority is inserted in the queue before a message with lower priority.

If the message queue is full and the flag *O\_NONBLOCK* is not set, the calling thread is suspended until the queue is not full. If the message queue is full and the flag *O\_NONBLOCK* is set, the message is not sent and the service returns immediately a value of -1 with *errno* set to *EAGAIN*.

**Parameters:**

- fd* message queue descriptor;
- buffer* pointer to the message to be sent;
- len* length of the message;

*prio* priority of the message.

**Returns:**

0 and send a message on success;

-1 with no message sent and *errno* set if:

- EBADF, *fd* is not a valid message queue descriptor open for writing;
- EMSGSIZE, the message length *len* exceeds the *mq\_msgsize* attribute of the message queue;
- EAGAIN, the flag O\_NONBLOCK is set for the descriptor *fd* and the message queue is full;
- EPERM, the caller context is invalid;
- EINTR, the service was interrupted by a signal.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.6.2.7** `int mq_setattr (mqd_t fd, const struct mq_attr *__restrict__ attr, struct mq_attr *__restrict__ oattr)`

Set flags of a message queue.

This service sets the flags of the *fd* descriptor to the value of the member *mq\_flags* of the **mq\_attr** structure pointed to by *attr*.

The previous value of the message queue attributes are stored at the address *oattr* if it is not *NULL*.

Only setting or clearing the O\_NONBLOCK flag has an effect.

**Parameters:**

*fd* message queue descriptor;

*attr* pointer to new attributes (only *mq\_flags* is used);

*oattr* if not *NULL*, address where previous message queue attributes will be stored on success.

**Return values:**

0 on success;

-1 with *errno* set if:

- EBADF, *fd* is not a valid message queue descriptor.

**See also:**

[Specification.](#)

### 3.6.2.8 `ssize_t mq_timedreceive (mqd_t fd, char *__restrict__ buffer, size_t len, unsigned *__restrict__ priop, const struct timespec *__restrict__ abs_timeout)`

Attempt, during a bounded time, to receive a message from a message queue.

This service is equivalent to `mq_receive()`, except that if the flag `O_NONBLOCK` is not set for the descriptor *fd* and the message queue is empty, the calling thread is only suspended until the timeout *abs\_timeout* expires.

#### Parameters:

- fd* the queue descriptor;
- buffer* the address where the received message will be stored on success;
- len* *buffer* length;
- priop* address where the priority of the received message will be stored on success.
- abs\_timeout* the timeout, expressed as an absolute value of the `CLOCK_REALTIME` clock.

#### Returns:

- the message length, and copy a message at the address *buffer* on success;
- 1 with no message unqueued and *errno* set if:
  - `EBADF`, *fd* is not a valid descriptor open for reading;
  - `EMSGSIZE`, the length *len* is lesser than the message queue *mq\_msgsize* attribute;
  - `EAGAIN`, the queue is empty, and the flag `O_NONBLOCK` is set for the descriptor *fd*;
  - `EPERM`, the caller context is invalid;
  - `EINTR`, the service was interrupted by a signal;
  - `ETIMEDOUT`, the specified timeout expired.

#### Valid contexts:

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

#### See also:

[Specification.](#)

### 3.6.2.9 `int mq_timedsend (mqd_t fd, const char * buffer, size_t len, unsigned prio, const struct timespec * abs_timeout)`

Attempt, during a bounded time, to send a message to a message queue.

This service is equivalent to `mq_send()`, except that if the message queue is full and the flag `O_NONBLOCK` is not set for the descriptor *fd*, the calling thread is only suspended until the timeout specified by *abs\_timeout* expires.

#### Parameters:

- fd* message queue descriptor;
- buffer* pointer to the message to be sent;
- len* length of the message;
- prio* priority of the message;
- abs\_timeout* the timeout, expressed as an absolute value of the `CLOCK_REALTIME` clock.

**Returns:**

0 and send a message on success;

-1 with no message sent and *errno* set if:

- EBADF, *fd* is not a valid message queue descriptor open for writing;
- EMSGSIZE, the message length exceeds the *mq\_msgsize* attribute of the message queue;
- EAGAIN, the flag O\_NONBLOCK is set for the descriptor *fd* and the message queue is full;
- EPERM, the caller context is invalid;
- ETIMEDOUT, the specified timeout expired;
- EINTR, the service was interrupted by a signal.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.6.2.10 int mq\_unlink (const char \* name)**

Unlink a message queue.

This service unlinks the message queue named *name*. The message queue is not destroyed until all queue descriptors obtained with the [mq\\_open\(\)](#) service are closed with the [mq\\_close\(\)](#) service. However, after a call to this service, the unlinked queue may no longer be reached with the [mq\\_open\(\)](#) service.

**Parameters:**

*name* name of the message queue to be unlinked.

**Return values:**

0 on success;

-1 with *errno* set if:

- EPERM, the caller context is invalid;
- ENAMETOOLONG, the length of the *name* argument exceeds 64 characters;
- ENOENT, the message queue does not exist.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- kernel-space cancellation cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode);
- user-space cancellation cleanup routine.

**See also:**

[Specification.](#)



## 3.7 Mutex services.

### 3.7.1 Detailed Description

Mutex services.

A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors.

A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

Before it can be used, a mutex has to be initialized with [pthread\\_mutex\\_init\(\)](#). An attribute object, which reference may be passed to this service, allows to select the features of the created mutex, namely its *type* (see [pthread\\_mutexattr\\_settype\(\)](#)), the priority *protocol* it uses (see [pthread\\_mutexattr\\_setprotocol\(\)](#)) and whether it may be shared between several processes (see [pthread\\_mutexattr\\_setpshared\(\)](#)).

By default, Xenomai POSIX skin mutexes are of the normal type, use no priority protocol and may not be shared between several processes.

Note that only [pthread\\_mutex\\_init\(\)](#) may be used to initialize a mutex, using the static initializer `PTHREAD_MUTEX_INITIALIZER` is not supported.

### Functions

- int [pthread\\_mutex\\_init](#) (pthread\_mutex\_t \*mx, const pthread\_mutexattr\_t \*attr)  
*Initialize a mutex.*
- int [pthread\\_mutex\\_destroy](#) (pthread\_mutex\_t \*mx)  
*Destroy a mutex.*
- int [pthread\\_mutex\\_trylock](#) (pthread\_mutex\_t \*mx)  
*Attempt to lock a mutex.*
- int [pthread\\_mutex\\_lock](#) (pthread\_mutex\_t \*mx)  
*Lock a mutex.*
- int [pthread\\_mutex\\_timedlock](#) (pthread\_mutex\_t \*mx, const struct timespec \*to)  
*Attempt, during a bounded time, to lock a mutex.*
- int [pthread\\_mutex\\_unlock](#) (pthread\_mutex\_t \*mx)  
*Unlock a mutex.*
- int [pthread\\_mutexattr\\_init](#) (pthread\_mutexattr\_t \*attr)  
*Initialize a mutex attributes object.*
- int [pthread\\_mutexattr\\_destroy](#) (pthread\_mutexattr\_t \*attr)  
*Destroy a mutex attributes object.*
- int [pthread\\_mutexattr\\_gettype](#) (const pthread\_mutexattr\_t \*attr, int \*type)

*Get the mutex type attribute from a mutex attributes object.*

- `int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int type)`  
*Set the mutex type attribute of a mutex attributes object.*
- `int pthread_mutexattr_getprotocol (const pthread_mutexattr_t *attr, int *proto)`  
*Get the protocol attribute from a mutex attributes object.*
- `int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr, int proto)`  
*Set the protocol attribute of a mutex attributes object.*
- `int pthread_mutexattr_getpshared (const pthread_mutexattr_t *attr, int *pshared)`  
*Get the process-shared attribute of a mutex attributes object.*
- `int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared)`  
*Set the process-shared attribute of a mutex attributes object.*

## 3.7.2 Function Documentation

### 3.7.2.1 `int pthread_mutex_destroy (pthread_mutex_t *mx)`

Destroy a mutex.

This service destroys the mutex *mx*, if it is unlocked and not referenced by any condition variable. The mutex becomes invalid for all mutex services (they all return the EINVAL error) except `pthread_mutex_init()`.

#### Parameters:

*mx* the mutex to be destroyed.

#### Returns:

- 0 on success,  
an error number if:
- EINVAL, the mutex *mx* is invalid;
  - EBUSY, the mutex is locked, or used by a condition variable.

#### See also:

[Specification.](#)

### 3.7.2.2 `int pthread_mutex_init (pthread_mutex_t *mx, const pthread_mutexattr_t *attr)`

Initialize a mutex.

This services initializes the mutex *mx*, using the mutex attributes object *attr*. If *attr* is *NULL*, default attributes are used (see `pthread_mutexattr_init()`).

#### Parameters:

- mx* the mutex to be initialized;  
*attr* the mutex attributes object.

**Returns:**

0 on success,  
an error number if:

- EINVAL, the mutex attributes object *attr* is invalid or uninitialized;
- EBUSY, the mutex *mx* was already initialized;
- ENOMEM, insufficient memory exists in the system heap to initialize the mutex, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ.

**See also:**

[Specification.](#)

**3.7.2.3 int pthread\_mutex\_lock (pthread\_mutex\_t \* mx)**

Lock a mutex.

This service attempts to lock the mutex *mx*. If the mutex is free, it becomes locked. If it was locked by another thread than the current one, the current thread is suspended until the mutex is unlocked. If it was already locked by the current mutex, the behaviour of this service depends on the mutex type :

- for mutexes of the *PTHREAD\_MUTEX\_NORMAL* type, this service deadlocks;
- for mutexes of the *PTHREAD\_MUTEX\_ERRORCHECK* type, this service returns the EDEADLK error number;
- for mutexes of the *PTHREAD\_MUTEX\_RECURSIVE* type, this service increments the lock recursion count and returns 0.

**Parameters:**

*mx* the mutex to be locked.

**Returns:**

0 on success  
an error number if:

- EPERM, the caller context is invalid;
- EINVAL, the mutex *mx* is invalid;
- EDEADLK, the mutex is of the *PTHREAD\_MUTEX\_ERRORCHECK* type and the mutex was already locked by the current thread;
- EAGAIN, the mutex is of the *PTHREAD\_MUTEX\_RECURSIVE* type and the maximum number of recursive locks has been exceeded.

**Valid contexts:**

- Xenomai kernel-space thread;
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

### 3.7.2.4 `int pthread_mutex_timedlock (pthread_mutex_t * mx, const struct timespec * to)`

Attempt, during a bounded time, to lock a mutex.

This service is equivalent to `pthread_mutex_lock()`, except that if the mutex *mx* is locked by another thread than the current one, this service only suspends the current thread until the timeout specified by *to* expires.

#### Parameters:

- mx* the mutex to be locked;
- to* the timeout, expressed as an absolute value of the CLOCK\_REALTIME clock.

#### Returns:

- 0 on success;
- an error number if:
  - EPERM, the caller context is invalid;
  - EINVAL, the mutex *mx* is invalid;
  - ETIMEDOUT, the mutex could not be locked and the specified timeout expired;
  - EDEADLK, the mutex is of the `PTHREAD_MUTEX_ERRORCHECK` type and the mutex was already locked by the current thread;
  - EAGAIN, the mutex is of the `PTHREAD_MUTEX_RECURSIVE` type and the maximum number of recursive locks has been exceeded.

#### Valid contexts:

- Xenomai kernel-space thread;
- Xenomai user-space thread (switches to primary mode).

#### See also:

[Specification.](#)

### 3.7.2.5 `int pthread_mutex_trylock (pthread_mutex_t * mx)`

Attempt to lock a mutex.

This service is equivalent to `pthread_mutex_lock()`, except that if the mutex *mx* is locked by another thread than the current one, this service returns immediately.

#### Parameters:

- mx* the mutex to be locked.

#### Returns:

- 0 on success;
- an error number if:
  - EPERM, the caller context is invalid;
  - EINVAL, the mutex is invalid;
  - EBUSY, the mutex was locked by another thread than the current one;
  - EAGAIN, the mutex is recursive, and the maximum number of recursive locks has been exceeded.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.7.2.6 int pthread\_mutex\_unlock (pthread\_mutex\_t \* *mx*)**

Unlock a mutex.

This service unlocks the mutex *mx*. If the mutex is of the *PTHREAD\_MUTEX\_RECURSIVE* type and the locking recursion count is greater than one, the lock recursion count is decremented and the mutex remains locked.

Attempting to unlock a mutex which is not locked or which is locked by another thread than the current one yields the EPERM error, whatever the mutex *type* attribute.

**Parameters:**

*mx* the mutex to be released.

**Returns:**

0 on success;

an error number if:

- EPERM, the caller context is invalid;
- EINVAL, the mutex *mx* is invalid;
- EPERM, the mutex was not locked by the current thread.

**Valid contexts:**

- Xenomai kernel-space thread,
- kernel-space cancellation cleanup routine,
- Xenomai user-space thread (switches to primary mode),
- user-space cancellation cleanup routine.

**See also:**

[Specification.](#)

**3.7.2.7 int pthread\_mutexattr\_destroy (pthread\_mutexattr\_t \* *attr*)**

Destroy a mutex attributes object.

This service destroys the mutex attributes object *attr*. The object becomes invalid for all mutex services (they all return EINVAL) except [pthread\\_mutexattr\\_init\(\)](#).

**Parameters:**

*attr* the initialized mutex attributes object to be destroyed.

**Returns:**

0 on success;

an error number if:

- EINVAL, the mutex attributes object *attr* is invalid.

See also:

[Specification.](#)

### 3.7.2.8 int pthread\_mutexattr\_getprotocol (const pthread\_mutexattr\_t \* attr, int \* proto)

Get the protocol attribute from a mutex attributes object.

This service stores, at the address *proto*, the value of the *protocol* attribute in the mutex attributes object *attr*.

The *protocol* attribute may only be one of *PTHREAD\_PRIO\_NONE* or *PTHREAD\_PRIO\_INHERIT*. See [pthread\\_mutexattr\\_setprotocol\(\)](#) for the meaning of these two constants.

**Parameters:**

*attr* an initialized mutex attributes object;

*proto* address where the value of the *protocol* attribute will be stored on success.

**Returns:**

0 on success,

an error number if:

- EINVAL, the *proto* address is invalid;
- EINVAL, the mutex attributes object *attr* is invalid.

See also:

[Specification.](#)

### 3.7.2.9 int pthread\_mutexattr\_getpshared (const pthread\_mutexattr\_t \* attr, int \* pshared)

Get the process-shared attribute of a mutex attributes object.

This service stores, at the address *pshared*, the value of the *pshared* attribute in the mutex attributes object *attr*.

The *pshared* attribute may only be one of *PTHREAD\_PROCESS\_PRIVATE* or *PTHREAD\_PROCESS\_SHARED*. See [pthread\\_mutexattr\\_setpshared\(\)](#) for the meaning of these two constants.

**Parameters:**

*attr* an initialized mutex attributes object;

*pshared* address where the value of the *pshared* attribute will be stored on success.

**Returns:**

0 on success;

an error number if:

- EINVAL, the *pshared* address is invalid;
- EINVAL, the mutex attributes object *attr* is invalid.

See also:

[Specification.](#)

### 3.7.2.10 `int pthread_mutexattr_gettype (const pthread_mutexattr_t * attr, int * type)`

Get the mutex type attribute from a mutex attributes object.

This service stores, at the address *type*, the value of the *type* attribute in the mutex attributes object *attr*.

See [pthread\\_mutex\\_lock\(\)](#) and [pthread\\_mutex\\_unlock\(\)](#) documentations for a description of the values of the *type* attribute and their effect on a mutex.

**Parameters:**

*attr* an initialized mutex attributes object,

*type* address where the *type* attribute value will be stored on success.

**Returns:**

0 on success,

an error number if:

- EINVAL, the *type* address is invalid;
- EINVAL, the mutex attributes object *attr* is invalid.

**See also:**

[Specification.](#)

### 3.7.2.11 `int pthread_mutexattr_init (pthread_mutexattr_t * attr)`

Initialize a mutex attributes object.

This service initializes the mutex attributes object *attr* with default values for all attributes. Default value are :

- for the *type* attribute, *PTHREAD\_MUTEX\_NORMAL*;
- for the *protocol* attribute, *PTHREAD\_PRIO\_NONE*;
- for the *pshared* attribute, *PTHREAD\_PROCESS\_PRIVATE*.

If this service is called specifying a mutex attributes object that was already initialized, the attributes object is reinitialized.

**Parameters:**

*attr* the mutex attributes object to be initialized.

**Returns:**

0 on success;

an error number if:

- ENOMEM, the mutex attributes object pointer *attr* is *NULL*.

**See also:**

[Specification.](#)

### 3.7.2.12 `int pthread_mutexattr_setprotocol(pthread_mutexattr_t * attr, int proto)`

Set the protocol attribute of a mutex attributes object.

This service set the *type* attribute of the mutex attributes object *attr*.

**Parameters:**

*attr* an initialized mutex attributes object,

*proto* value of the *protocol* attribute, may be one of:

- `PTHREAD_PRIO_NONE`, meaning that a mutex created with the attributes object *attr* will not follow any priority protocol;
- `PTHREAD_PRIO_INHERIT`, meaning that a mutex created with the attributes object *attr*, will follow the priority inheritance protocol.

The value `PTHREAD_PRIO_PROTECT` (priority ceiling protocol) is unsupported.

**Returns:**

0 on success,

an error number if:

- `EINVAL`, the mutex attributes object *attr* is invalid;
- `ENOTSUP`, the value of *proto* is unsupported;
- `EINVAL`, the value of *proto* is invalid.

**See also:**

[Specification](#).

### 3.7.2.13 `int pthread_mutexattr_setpshared(pthread_mutexattr_t * attr, int pshared)`

Set the process-shared attribute of a mutex attributes object.

This service set the *pshared* attribute of the mutex attributes object *attr*.

**Parameters:**

*attr* an initialized mutex attributes object.

*pshared* value of the *pshared* attribute, may be one of:

- `PTHREAD_PROCESS_PRIVATE`, meaning that a mutex created with the attributes object *attr* will only be accessible by threads within the same process as the thread that initialized the mutex;
- `PTHREAD_PROCESS_SHARED`, meaning that a mutex created with the attributes object *attr* will be accessible by any thread that has access to the memory where the mutex is allocated.

**Returns:**

0 on success,

an error status if:

- `EINVAL`, the mutex attributes object *attr* is invalid;
- `EINVAL`, the value of *pshared* is invalid.

**See also:**

[Specification](#).



#### 3.7.2.14 `int pthread_mutexattr_settype (pthread_mutexattr_t * attr, int type)`

Set the mutex type attribute of a mutex attributes object.

This service set the *type* attribute of the mutex attributes object *attr*.

See [pthread\\_mutex\\_lock\(\)](#) and [pthread\\_mutex\\_unlock\(\)](#) documentations for a description of the values of the *type* attribute and their effect on a mutex.

The `PTHREAD_MUTEX_DEFAULT` default *type* is the same as `PTHREAD_MUTEX_NORMAL`. Note that using a Xenomai POSIX skin recursive mutex with a Xenomai POSIX skin condition variable is safe (see [pthread\\_cond\\_wait\(\)](#) documentation).

##### Parameters:

- attr* an initialized mutex attributes object,
- type* value of the *type* attribute.

##### Returns:

- 0 on success,
- an error number if:
  - `EINVAL`, the mutex attributes object *attr* is invalid;
  - `EINVAL`, the value of *type* is invalid for the *type* attribute.

##### See also:

[Specification.](#)

## 3.8 Threads scheduling services.

### 3.8.1 Detailed Description

Thread scheduling services.

Xenomai POSIX skin supports the scheduling policies `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`.

The `SCHED_OTHER` policy is mainly useful for user-space non-realtime activities that need to synchronize with real-time activities.

The `SCHED_RR` policy is only effective if the time base is periodic (i.e. if configured with the compilation constant `CONFIG_XENO_OPT_POSIX_PERIOD` or the `xeno_nucleus` module parameter `tick_arg` set to a non null value). The `SCHED_RR` round-robin time slice is configured with the `xeno_posix` module parameter `time_slice`, as a count of system timer clock ticks.

The `SCHED_SPORADIC` policy is not supported.

The scheduling policy and priority of a thread is set when creating a thread, by using thread creation attributes (see [pthread\\_attr\\_setinheritsched\(\)](#), [pthread\\_attr\\_setschedpolicy\(\)](#) and [pthread\\_attr\\_setschedparam\(\)](#)), or when the thread is already running by using the service [pthread\\_setschedparam\(\)](#).

See also:

[Specification.](#)

### Functions

- int [sched\\_get\\_priority\\_min](#) (int policy)  
*Get minimum priority of the specified scheduling policy.*
- int [sched\\_get\\_priority\\_max](#) (int policy)  
*Get maximum priority of the specified scheduling policy.*
- int [sched\\_rr\\_get\\_interval](#) (int pid, struct timespec \*interval)  
*Get the round-robin scheduling time slice.*
- int [pthread\\_getschedparam](#) (pthread\_t tid, int \*pol, struct sched\_param \*par)  
*Get the scheduling policy and parameters of the specified thread.*
- int [pthread\\_setschedparam](#) (pthread\_t tid, int pol, const struct sched\_param \*par)  
*Set the scheduling policy and parameters of the specified thread.*
- int [sched\\_yield](#) (void)  
*Yield the processor.*

### 3.8.2 Function Documentation

#### 3.8.2.1 int pthread\_getschedparam (pthread\_t tid, int \*pol, struct sched\_param \*par)

Get the scheduling policy and parameters of the specified thread.

This service returns, at the addresses *pol* and *par*, the current scheduling policy and scheduling parameters (i.e. priority) of the Xenomai POSIX skin thread *tid*. If this service is called from user-space and *tid* is not the identifier of a Xenomai POSIX skin thread, this service fallback to Linux regular `pthread_getschedparam` service.

**Parameters:**

- tid* target thread;
- pol* address where the scheduling policy of *tid* is stored on success;
- par* address where the scheduling parameters of *tid* is stored on success.

**Returns:**

- 0 on success;
- an error number if:
  - ESRCH, *tid* is invalid.

**See also:**

[Specification.](#)

### 3.8.2.2 `int pthread_setschedparam(pthread_t tid, int pol, const struct sched_param * par)`

Set the scheduling policy and parameters of the specified thread.

This service set the scheduling policy of the Xenomai POSIX skin thread *tid* to the value *pol*, and its scheduling parameters (i.e. its priority) to the value pointed to by *par*.

When used in user-space, passing the current thread ID as *tid* argument, this service turns the current thread into a Xenomai POSIX skin thread. If *tid* is neither the identifier of the current thread nor the identifier of a Xenomai POSIX skin thread this service falls back to the regular [pthread\\_setschedparam\(\)](#) service, hereby causing the current thread to switch to secondary mode if it is Xenomai thread.

**Parameters:**

- tid* target thread;
- pol* scheduling policy, one of SCHED\_FIFO, SCHED\_RR or SCHED\_OTHER;
- par* scheduling parameters address.

**Returns:**

- 0 on success;
- an error number if:
  - ESRCH, *tid* is invalid;
  - EINVAL, *pol* or *par->sched\_priority* is invalid;
  - EAGAIN, in user-space, insufficient memory exists in the system heap, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ;
  - EFAULT, in user-space, *par* is an invalid address;
  - EPERM, in user-space, the calling process does not have superuser permissions.

**See also:**

[Specification.](#)

### 3.8.2.3 `int sched_get_priority_max (int policy)`

Get maximum priority of the specified scheduling policy.

This service returns the maximum priority of the scheduling policy *policy*.

**Parameters:**

*policy* scheduling policy, one of SCHED\_FIFO, SCHED\_RR, or SCHED\_OTHER.

**Return values:**

- 0* on success;
- 1* with *errno* set if:
  - EINVAL, *policy* is invalid.

**See also:**

[Specification.](#)

### 3.8.2.4 `int sched_get_priority_min (int policy)`

Get minimum priority of the specified scheduling policy.

This service returns the minimum priority of the scheduling policy *policy*.

**Parameters:**

*policy* scheduling policy, one of SCHED\_FIFO, SCHED\_RR, or SCHED\_OTHER.

**Return values:**

- 0* on success;
- 1* with *errno* set if:
  - EINVAL, *policy* is invalid.

**See also:**

[Specification.](#)

### 3.8.2.5 `int sched_rr_get_interval (int pid, struct timespec * interval)`

Get the round-robin scheduling time slice.

This service returns the time quantum used by Xenomai POSIX skin SCHED\_RR scheduling policy.

In kernel-space, this service only works if *pid* is zero, in user-space, round-robin scheduling policy is not supported, and this service not implemented.

**Parameters:**

- pid* must be zero;
- interval* address where the round-robin scheduling time quantum will be returned on success.

**Return values:**

- 0* on success;

-1 with *errno* set if:

- ESRCH, *pid* is invalid (not 0).

See also:

[Specification.](#)

#### 3.8.2.6 int sched\_yield (void)

Yield the processor.

This function move the current thread at the end of its priority group.

Return values:

0

See also:

[Specification.](#)

## 3.9 Semaphores services.

### 3.9.1 Detailed Description

Semaphores services.

Semaphores are counters for resources shared between threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically.

Semaphores have a maximum value past which they cannot be incremented. The macro `SEM_VALUE_MAX` is defined to be this maximum value.

### Functions

- `int sem_init (sem_t *sm, int pshared, unsigned value)`  
*Initialize an unnamed semaphore.*
- `int sem_destroy (sem_t *sm)`  
*Destroy an unnamed semaphore.*
- `sem_t * sem_open (const char *name, int oflags,...)`  
*Open a named semaphore.*
- `int sem_close (sem_t *sm)`  
*Close a named semaphore.*
- `int sem_unlink (const char *name)`  
*Unlink a named semaphore.*
- `int sem_trywait (sem_t *sm)`  
*Attempt to lock a semaphore.*
- `int sem_wait (sem_t *sm)`  
*Lock a semaphore.*
- `int sem_timedwait (sem_t *sm, const struct timespec *abs_timeout)`  
*Attempt, during a bounded time, to lock a semaphore.*
- `int sem_post (sem_t *sm)`  
*Unlock a semaphore.*
- `int sem_getvalue (sem_t *sm, int *value)`  
*Get the value of a semaphore.*

### 3.9.2 Function Documentation

#### 3.9.2.1 `int sem_close (sem_t * sm)`

Close a named semaphore.

This service closes the semaphore *sm*. The semaphore is destroyed only when unlinked with a call to the [sem\\_unlink\(\)](#) service and when each call to [sem\\_open\(\)](#) matches a call to this service.

When a semaphore is destroyed, the memory it used is returned to the system heap, so that further references to this semaphore are not guaranteed to fail, as is the case for unnamed semaphores.

This service fails if *sm* is an unnamed semaphore.

**Parameters:**

*sm* the semaphore to be closed.

**Return values:**

0 on success;

-1 with *errno* set if:

- EINVAL, the semaphore *sm* is invalid or is an unnamed semaphore.

**See also:**

[Specification.](#)

### 3.9.2.2 int sem\_destroy (sem\_t \* *sm*)

Destroy an unnamed semaphore.

This service destroys the semaphore *sm*. Threads currently blocked on *sm* are unblocked and the service they called return -1 with *errno* set to EINVAL. The semaphore is then considered invalid by all semaphore services (they all fail with *errno* set to EINVAL) except [sem\\_init\(\)](#).

This service fails if *sm* is a named semaphore.

**Parameters:**

*sm* the semaphore to be destroyed.

**Return values:**

0 on success,

-1 with *errno* set if:

- EINVAL, the semaphore *sm* is invalid or a named semaphore.

**See also:**

[Specification.](#)

### 3.9.2.3 int sem\_getvalue (sem\_t \* *sm*, int \* *value*)

Get the value of a semaphore.

This service stores at the address *value*, the current count of the semaphore *sm*. The state of the semaphore is unchanged.

If the semaphore is currently locked, the value stored is zero.

**Parameters:**

*sm* a semaphore;

*value* address where the semaphore count will be stored on success.

**Return values:**

- 0* on success;
- 1* with *errno* set if:
  - EINVAL, the semaphore is invalid or uninitialized.

**See also:**

[Specification.](#)

### 3.9.2.4 `int sem_init (sem_t * sm, int pshared, unsigned value)`

Initialize an unnamed semaphore.

This service initializes the semaphore *sm*, with the value *value*.

This service fails if *sm* is already initialized or is a named semaphore.

**Parameters:**

- sm* the semaphore to be initialized;
- pshared* if zero, means that the new semaphore may only be used by threads in the same process as the thread calling `sem_init()`; if non zero, means that the new semaphore may be used by any thread that has access to the memory where the semaphore is allocated.
- value* the semaphore initial value.

**Return values:**

- 0* on success,
- 1* with *errno* set if:
  - EBUSY, the semaphore *sm* was already initialized;
  - ENOSPC, insufficient memory exists in the system heap to initialize the semaphore, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ;
  - EINVAL, the *value* argument exceeds SEM\_VALUE\_MAX.

**See also:**

[Specification.](#)

### 3.9.2.5 `sem_t* sem_open (const char * name, int oflags, ...)`

Open a named semaphore.

This service establishes a connection between the semaphore named *name* and the calling context (kernel-space as a whole, or user-space process).

If no semaphore named *name* exists and *oflags* has the *O\_CREAT* bit set, the semaphore is created by this function, using two more arguments:

- a *mode* argument, of type **mode\_t**, currently ignored;
- a *value* argument, of type **unsigned**, specifying the initial value of the created semaphore.



If *oflags* has the two bits *O\_CREAT* and *O\_EXCL* set and the semaphore already exists, this service fails.

*name* may be any arbitrary string, in which slashes have no particular meaning. However, for portability, using a name which starts with a slash and contains no other slash is recommended.

If `sem_open()` is called from the same context (kernel-space as a whole, or user-space process) several times with the same value of *name*, the same address is returned.

**Parameters:**

*name* the name of the semaphore to be created;

*oflags* flags.

**Returns:**

the address of the named semaphore on success;

SEM\_FAILED with *errno* set if:

- ENAMETOOLONG, the length of the *name* argument exceeds 64 characters;
- EEXIST, the bits *O\_CREAT* and *O\_EXCL* were set in *oflags* and the named semaphore already exists;
- ENOENT, the bit *O\_CREAT* is not set in *oflags* and the named semaphore does not exist;
- ENOSPC, insufficient memory exists in the system heap to create the semaphore, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ;
- EINVAL, the *value* argument exceeds SEM\_VALUE\_MAX.

**See also:**

[Specification.](#)

**3.9.2.6 int sem\_post (sem\_t \* sm)**

Unlock a semaphore.

This service unlocks the semaphore *sm*.

If no thread is currently blocked on this semaphore, its count is incremented, otherwise the highest priority thread is unblocked.

**Parameters:**

*sm* the semaphore to be unlocked.

**Return values:**

0 on success;

-1 with *errno* set if:

- EINVAL, the specified semaphore is invalid or uninitialized;
- EAGAIN, the semaphore count is SEM\_VALUE\_MAX.

**See also:**

[Specification.](#)

### 3.9.2.7 `int sem_timedwait (sem_t * sm, const struct timespec * abs_timeout)`

Attempt, during a bounded time, to lock a semaphore.

This service is equivalent to [sem\\_wait\(\)](#), except that the caller is only blocked until the timeout *abs\_timeout* expires.

**Parameters:**

*sm* the semaphore to be locked;

*abs\_timeout* the timeout, expressed as an absolute value of the CLOCK\_REALTIME clock.

**Return values:**

0 on success;

-1 with *errno* set if:

- EPERM, the caller context is invalid;
- EINVAL, the semaphore is invalid or uninitialized;
- EINVAL, the specified timeout is invalid;
- EINTR, the caller was interrupted by a signal while blocked in this service;
- ETIMEDOUT, the semaphore could not be locked and the specified timeout expired.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification](#).

### 3.9.2.8 `int sem_trywait (sem_t * sm)`

Attempt to lock a semaphore.

This service is equivalent to [sem\\_wait\(\)](#), except that it returns immediately if the semaphore *sm* is currently locked, and that it is not a cancellation point.

**Parameters:**

*sm* the semaphore to be locked.

**Return values:**

0 on success;

-1 with *errno* set if:

- EINVAL, the specified semaphore is invalid or uninitialized;
- EAGAIN, the specified semaphore is currently locked.

\*

**See also:**

[Specification](#).

### 3.9.2.9 `int sem_unlink (const char * name)`

Unlink a named semaphore.

This service unlinks the semaphore named *name*. This semaphore is not destroyed until all references obtained with `sem_open()` are closed by calling `sem_close()`. However, the unlinked semaphore may no longer be reached with the `sem_open()` service.

When a semaphore is destroyed, the memory it used is returned to the system heap, so that further references to this semaphore are not guaranteed to fail, as is the case for unnamed semaphores.

**Parameters:**

*name* the name of the semaphore to be unlinked.

**Return values:**

0 on success;

-1 with *errno* set if:

- ENAMETOOLONG, the length of the *name* argument exceeds 64 characters;
- ENOENT, the named semaphore does not exist.

**See also:**

[Specification.](#)

### 3.9.2.10 `int sem_wait (sem_t * sm)`

Lock a semaphore.

This service locks the semaphore *sm* if it is currently unlocked (i.e. if its value is greater than 0). If the semaphore is currently locked, the calling thread is suspended until the semaphore is unlocked, or a signal is delivered to the calling thread.

This service is a cancellation point for Xenomai POSIX skin threads (created with the `pthread_create()` service). When such a thread is cancelled while blocked in a call to this service, the semaphore state is left unchanged before the cancellation cleanup handlers are called.

**Parameters:**

*sm* the semaphore to be locked.

**Return values:**

0 on success;

-1 with *errno* set if:

- EPERM, the caller context is invalid;
- EINVAL, the semaphore is invalid or uninitialized;
- EINTR, the caller was interrupted by a signal while blocked in this service.

**Valid contexts:**

- Xenomai kernel-space thread,
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

## 3.10 Shared memory services.

### 3.10.1 Detailed Description

Shared memory services.

Shared memory objects are memory regions that can be mapped into processes address space, allowing them to share these regions as well as to share them with kernel-space modules.

Shared memory are also the only mean by which anonymous POSIX skin synchronization objects (mutexes, condition variables or semaphores) may be shared between kernel-space modules and user-space processes, or between several processes.

### Functions

- `int shm_open` (const char \*name, int oflags, mode\_t mode)  
*Open a shared memory object.*
- `int close` (int fd)  
*Close a file descriptor.*
- `int shm_unlink` (const char \*name)  
*Unlink a shared memory object.*
- `int ftruncate` (int fd, off\_t len)  
*Truncate a file or shared memory object to a specified length.*
- `void * mmap` (void \*addr, size\_t len, int prot, int flags, int fd, off\_t off)  
*Map pages of memory.*
- `int munmap` (void \*addr, size\_t len)  
*Unmap pages of memory.*

### 3.10.2 Function Documentation

#### 3.10.2.1 `int close (int fd)`

Close a file descriptor.

This service closes the file descriptor *fd*. In kernel-space, this service only works for file descriptors opened with `shm_open()`, i.e. shared memory objects. A shared memory object is only destroyed once all file descriptors are closed with this service, it is unlinked with the `shm_unlink()` service, and all mappings are unmapped with the `munmap()` service.

#### Parameters:

*fd* file descriptor.

#### Return values:

0 on success;

-1 with *errno* set if:

- EBADF, *fd* is not a valid file descriptor (in kernel-space, it was not obtained with [shm\\_open\(\)](#));
- EPERM, the caller context is invalid.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- kernel-space cancellation cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode);
- user-space cancellation cleanup routine.

**See also:**

[Specification.](#)

**3.10.2.2 int ftruncate (int *fd*, off\_t *len*)**

Truncate a file or shared memory object to a specified length.

When used in kernel-space, this service set to *len* the size of a shared memory object opened with the [shm\\_open\(\)](#) service. In user-space this service falls back to Linux regular ftruncate service for file descriptors not obtained with [shm\\_open\(\)](#). When this service is used to increase the size of a shared memory object, the added space is zero-filled.

Shared memory are suitable for direct memory access (allocated in physically contiguous memory) if their size is less than or equal to 128 K.

Shared memory objects may only be resized if they are not currently mapped.

**Parameters:**

*fd* file descriptor;

*len* new length of the underlying file or shared memory object.

**Return values:**

0 on success;

-1 with *errno* set if:

- EBADF, *fd* is not a valid file descriptor;
- EPERM, the caller context is invalid;
- EINVAL, the specified length is invalid;
- EINTR, this service was interrupted by a signal;
- EBUSY, *fd* is a shared memory object descriptor and the underlying shared memory is currently mapped;
- EFBIG, allocation of system memory failed.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode).

**See also:**

[Specification.](#)

### 3.10.2.3 void\* mmap (void \* addr, size\_t len, int prot, int flags, int fd, off\_t off)

Map pages of memory.

This service allow shared memory regions to be accessed by the caller.

When used in kernel-space, this service returns the address of the offset *off* of the shared memory object underlying *fd*. The protection flags *prot*, are only checked for consistency with *fd* open flags, but memory protection is unsupported. An existing shared memory region exists before it is mapped, this service only increments a reference counter.

The only supported value for *flags* is *MAP\_SHARED*.

When used in user-space, this service maps the specified shared memory region into the caller address-space. If *fd* is not a shared memory object descriptor (i.e. not obtained with [shm\\_open\(\)](#)), this service falls back to the regular Linux mmap service.

#### Parameters:

*addr* ignored.

*len* size of the shared memory region to be mapped.

*prot* protection bits, checked in kernel-space, but only useful in user-space, are a bitwise or of the following values:

- PROT\_NONE, meaning that the mapped region can not be accessed;
- PROT\_READ, meaning that the mapped region can be read;
- PROT\_WRITE, meaning that the mapped region can be written;
- PROT\_EXEC, meaning that the mapped region can be executed.

*flags* only MAP\_SHARED is accepted, meaning that the mapped memory region is shared.

*fd* file descriptor, obtained with [shm\\_open\(\)](#).

*off* offset in the shared memory region.

#### Return values:

0 on success;

MAP\_FAILED with *errno* set if:

- EINVAL, *len* is null or *addr* is not a multiple of *PAGE\_SIZE*;
- EBADF, *fd* is not a shared memory object descriptor (obtained with [shm\\_open\(\)](#));
- EPERM, the caller context is invalid;
- ENOTSUP, *flags* is not *MAP\_SHARED*;
- EACCES, *fd* is not opened for reading or is not open for writing and PROT\_WRITE is set in *prot*;
- EINTR, this service was interrupted by a signal;
- ENXIO, the range [*off*;*off*+*len*) is invalid for the shared memory region specified by *fd*;
- EAGAIN, insufficient memory exists in the system heap to create the mapping, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ.

#### Valid contexts:

- kernel module initialization or cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode).

#### See also:

[Specification](#).

### 3.10.2.4 `int munmap (void * addr, size_t len)`

Unmap pages of memory.

This service unmaps the shared memory region [*addr*;*addr*+*len*) from the caller address-space.

When called from kernel-space the memory region remain accessible as long as it exists, and this service only decrements a reference counter.

When called from user-space, if the region is not a shared memory region, this service falls back to the regular Linux [munmap\(\)](#) service.

#### Parameters:

*addr* start address of shared memory area;

*len* length of the shared memory area.

#### Return values:

0 on success;

-1 with *errno* set if:

- EINVAL, *len* is null, *addr* is not a multiple of the page size or the range [*addr*;*addr*+*len*) is not a mapped region;
- ENXIO, *addr* is not the address of a shared memory area;
- EPERM, the caller context is invalid;
- EINTR, this service was interrupted by a signal.

#### Valid contexts:

- kernel module initialization or cleanup routine;
- kernel-space cancellation cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode);
- user-space cancellation cleanup routine.

#### See also:

[Specification.](#)

### 3.10.2.5 `int shm_open (const char * name, int oflags, mode_t mode)`

Open a shared memory object.

This service establishes a connection between a shared memory object and a file descriptor. Further use of this descriptor will allow to dimension and map the shared memory into the calling context address space.

One of the following access mode should be set in *oflags*:

- O\_RDONLY, meaning that the shared memory object may only be mapped with the PROT\_READ flag;
- O\_WRONLY, meaning that the shared memory object may only be mapped with the PROT\_WRITE flag;
- O\_RDWR, meaning that the shared memory object may be mapped with the PROT\_READ | PROT\_WRITE flag.

If no shared memory object named *name* exists, and *oflags* has the *O\_CREAT* bit set, the shared memory object is created by this function.

If *oflags* has the two bits *O\_CREAT* and *O\_EXCL* set and the shared memory object already exists, this service fails.

If *oflags* has the bit *O\_TRUNC* set, the shared memory exists and is not currently mapped, its size is truncated to 0.

*name* may be any arbitrary string, in which slashes have no particular meaning. However, for portability, using a name which starts with a slash and contains no other slash is recommended.

**Parameters:**

- name* name of the shared memory object to open;
- oflags* flags.
- mode* ignored.

**Returns:**

- a file descriptor on success;
- 1 with *errno* set if:
  - ENAMETOOLONG, the length of the *name* argument exceeds 64 characters;
  - EEXIST, the bits *O\_CREAT* and *O\_EXCL* were set in *oflags* and the shared memory object already exists;
  - ENOENT, the bit *O\_CREAT* is not set in *oflags* and the shared memory object does not exist;
  - ENOSPC, insufficient memory exists in the system heap to create the shared memory object, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ;
  - EPERM, the caller context is invalid;
  - EINVAL, the *O\_TRUNC* flag was specified and the shared memory object is currently mapped;
  - EMFILE, too many descriptors are currently open.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode).

**See also:**

[Specification.](#)

### 3.10.2.6 int shm\_unlink (const char \* name)

Unlink a shared memory object.

This service unlinks the shared memory object named *name*. The shared memory object is not destroyed until every file descriptor obtained with the [shm\\_open\(\)](#) service is closed with the [close\(\)](#) service and all mappings done with [mmap\(\)](#) are unmapped with [munmap\(\)](#). However, after a call to this service, the unlinked shared memory object may no longer be reached with the [shm\\_open\(\)](#) service.

**Parameters:**

- name* name of the shared memory object to be unlinked.



**Return values:**

*0* on success;

*-1* with *errno* set if:

- EPERM, the caller context is invalid;
- ENAMETOOLONG, the length of the *name* argument exceeds 64 characters;
- ENOENT, the shared memory object does not exist.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- kernel-space cancellation cleanup routine;
- user-space thread (Xenomai threads switch to secondary mode);
- user-space cancellation cleanup routine.

**See also:**

[Specification.](#)

## 3.11 Signals services.

### 3.11.1 Detailed Description

Signals management services.

Signals are asynchronous notifications delivered to a process or thread. Such notifications occur as the result of an exceptional event or at the request of another process.

The services documented here are reserved to Xenomai kernel-space threads, user-space threads switch to secondary mode when handling signals, and use Linux regular signals services.

Xenomai POSIX skin signals are implemented as real-time signals, meaning that they are queued when posted several times to a thread before the first notification is handled, and that each signal carry additional data in a **siginfo\_t** object. In order to ensure consistence with user-space signals, valid signals number range from 1 to SIGRTMAX, signals from SIGRTMIN to SIGRTMAX being higher priority than signals from 1 to SIGRTMIN-1. As a special case, signal 0 may be used with services [pthread\\_kill\(\)](#) and [pthread\\_sigqueue\\_np\(\)](#) to check if a thread exists, but entails no other action.

The action to be taken upon reception of a signal depends on the thread signal mask, (see [pthread\\_sigmask\(\)](#)), and on the settings described by a **sigaction** structure (see [sigaction\(\)](#)).

### Functions

- [int sigemptyset](#) (sigset\_t \*set)  
*Initialize and empty a signal set.*
- [int sigfillset](#) (sigset\_t \*set)  
*Initialize and fill a signal set.*
- [int sigaddset](#) (sigset\_t \*set, int sig)  
*Add a signal to a signal set.*
- [int sigdelset](#) (sigset\_t \*set, int sig)  
*Delete a signal from a signal set.*
- [int sigismember](#) (const sigset\_t \*set, int sig)  
*Test for a signal in a signal set.*
- [int sigaction](#) (int sig, const struct sigaction \*act, struct sigaction \*oact)  
*Examine and change a signal action.*
- [int pthread\\_kill](#) (pthread\_t thread, int sig)  
*Send a signal to a thread.*
- [int pthread\\_sigqueue\\_np](#) (pthread\_t thread, int sig, union sigval value)  
*Queue a signal to a thread.*
- [int sigpending](#) (sigset\_t \*set)  
*Examine pending signals.*

- `int pthread_sigmask (int how, const sigset_t *set, sigset_t *oset)`  
*Examine and change the set of signals blocked by a thread.*
- `int sigwait (const sigset_t *set, int *sig)`  
*Wait for signals.*
- `int sigwaitinfo (const sigset_t *__restrict set, siginfo_t *__restrict info)`  
*Wait for signals.*
- `int sigtimedwait (const sigset_t *__restrict set, siginfo_t *__restrict info, const struct timespec *__restrict timeout)`  
*Wait during a bounded time for signals.*

### 3.11.2 Function Documentation

#### 3.11.2.1 `int pthread_kill (pthread_t thread, int sig)`

Send a signal to a thread.

This service send the signal *sig* to the Xenomai POSIX skin thread *thread* (created with [pthread\\_create\(\)](#)). If *sig* is zero, this service check for existence of the thread *thread*, but no signal is sent.

##### Parameters:

*thread* thread identifier;  
*sig* signal number.

##### Returns:

0 on success;  
an error number if:

- EINVAL, *sig* is an invalid signal number;
- EAGAIN, the maximum number of pending signals has been exceeded;
- ESRCH, *thread* is an invalid thread identifier.

##### See also:

[Specification.](#)

#### 3.11.2.2 `int pthread_sigmask (int how, const sigset_t *set, sigset_t *oset)`

Examine and change the set of signals blocked by a thread.

The signal mask of a thread is the set of signals that are blocked by this thread.

If *oset* is not NULL, this service stores, at the address *oset* the current signal mask of the calling thread.

If *set* is not NULL, this service sets the signal mask of the calling thread according to the value of *how*, as follow:

- if *how* is SIG\_BLOCK, the signals in *set* are added to the calling thread signal mask;

- if *how* is SIG\_SETMASK, the calling thread signal mask is set to *set*;
- if *how* is SIG\_UNBLOCK, the signals in *set* are removed from the calling thread signal mask.

If some signals are unblocked by this service, they are handled before this service returns.

**Parameters:**

- how* if *set* is not null, a value indicating how to interpret *set*;
- set* if not null, a signal set that will be used to modify the calling thread signal mask;
- oset* if not null, address where the previous value of the calling thread signal mask will be stored on success.

**Returns:**

- 0 on success;
- an error number if:
  - EPERM, the calling context is invalid;
  - EINVAL, *how* is not SIG\_BLOCK, SIG\_UNBLOCK or SIG\_SETMASK.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread.

**See also:**

[Specification.](#)

### 3.11.2.3 int pthread\_sigqueue\_np (pthread\_t *thread*, int *sig*, union sigval *value*)

Queue a signal to a thread.

This service send the signal *sig* to the Xenomai POSIX skin thread *thread* (created with [pthread\\_create\(\)](#)), with the value *value*. If *sig* is zero, this service check for existence of the thread *thread*, but no signal is sent.

This service is equivalent to the POSIX service sigqueue(), except that the signal is directed to a thread instead of being directed to a process.

**Parameters:**

- thread* thread identifier,
- sig* signal number,
- value* additional datum passed to *thread* with the signal *sig*.

**Returns:**

- 0 on success;
- an error number if:
  - EINVAL, *sig* is an invalid signal number;
  - EAGAIN, the maximum number of pending signals has been exceeded;
  - ESRCH, *thread* is an invalid thread identifier.

**See also:**

[sigqueue\(\) specification.](#)

### 3.11.2.4 `int sigaction (int sig, const struct sigaction * act, struct sigaction * oact)`

Examine and change a signal action.

The **sigaction** structure describes the actions to be taken upon signal delivery. A **sigaction** structure is associated with every signal, for the kernel-space as a whole.

If *oact* is not *NULL*, this service returns at the address *oact*, the current value of the **sigaction** structure associated with the signal *sig*.

If *act* is not *NULL*, this service set to the value pointed to by *act*, the **sigaction** structure associated with the signal *sig*.

The structure **sigaction** has the following members:

- *sa\_flags*, is a bitwise OR of the flags;
  - SA\_RESETHAND, meaning that the signal handler will be reset to SIG\_GFL and SA\_SIGINFO cleared upon reception of a signal,
  - SA\_NODEFER, meaning that the signal handler will be called with the signal *sig* not masked when handling the signal *sig*,
  - SA\_SIGINFO, meaning that the member *sa\_sigaction* of the **sigaction** structure will be used as a signal handler instead of *sa\_handler*
- *sa\_mask*, of type **sigset\_t**, is the value to which the thread signals mask will be set during execution of the signal handler (*sig* is automatically added to this set if SA\_NODEFER is not set in *sa\_flags*);
- *sa\_handler*, of type **void (\*)(int)** is the signal handler which will be called upon signal delivery if SA\_SIGINFO is not set in *sa\_flags*, or one of SIG\_IGN or SIG\_DFL, meaning that the signal will be respectively ignored or handled with the default handler;
- *sa\_sigaction*, of type **void (\*)(int, siginfo\_t \*, void \*)** is the signal handler which will be called upon signal delivery if SA\_SIGINFO is set in *sa\_flags*.

When using *sa\_handler* as a signal handler, it is passed the number of the received signal, when using *sa\_sigaction*, two additional arguments are passed:

- a pointer to a **siginfo\_t** object, containing additional information about the received signal;
- a void pointer, always null in this implementation.

The following members of the **siginfo\_t** structure are filled by this implementation:

- *si\_signo*, the signal number;
- *si\_code*, the provenance of the signal, one of:
  - SI\_QUEUE, the signal was queued with [pthread\\_sigqueue\\_np\(\)](#),
  - SI\_USER, the signal was queued with [pthread\\_kill\(\)](#),
  - SI\_TIMER, the signal was queued by a timer (see [timer\\_settime\(\)](#)),
  - SI\_MESQ, the signal was queued by a message queue (see [mq\\_notify\(\)](#));
- *si\_value*, an additional datum, of type **union signal**.

#### Parameters:

*sig* a signal number;

*act* if not null, description of the action to be taken upon notification of the signal *sig*;  
*oact* if not null, address where the previous description of the signal action is stored on success.

**Return values:**

- 0* on success;
- 1* with *errno* set if:
  - *EINVAL*, *sig* is an invalid signal number;
  - *ENOTSUP*, the *sa\_flags* member of *act* contains other flags than *SA\_RESETHAND*, *SA\_NODEFER* and *SA\_SIGINFO*;

**See also:**

[Specification.](#)

### 3.11.2.5 int sigaddset (sigset\_t \* set, int sig)

Add a signal to a signal set.

This service adds the signal number *sig* to the signal set pointed to by *set*.

**Parameters:**

- set* address of a signal set;
- sig* signal to be added to *set*.

**Return values:**

- 0* on success;
- 1* with *errno* set if:
  - *EINVAL*, *sig* is not a valid signal number.

**See also:**

[Specification.](#)

### 3.11.2.6 int sigdelset (sigset\_t \* set, int sig)

Delete a signal from a signal set.

This service remove the signal number *sig* from the signal set pointed to by *set*.

**Parameters:**

- set* address of a signal set;
- sig* signal to be removed from *set*.

**Return values:**

- 0* on success;
- 1* with *errno* set if:
  - *EINVAL*, *sig* is not a valid signal number.

**See also:**

[Specification.](#)

### 3.11.2.7 `int sigemptyset (sigset_t * set)`

Initialize and empty a signal set.

This service initializes and empties the signal set pointed to by *set*.

**Parameters:**

*set* address of a the signal set to be initialized.

**Return values:**

0

**See also:**

[Specification.](#)

### 3.11.2.8 `int sigfillset (sigset_t * set)`

Initialize and fill a signal set.

This service initializes and fills the signal set pointed to by *set*.

**Parameters:**

*set* address of a the signal set to be filled.

**Return values:**

0

**See also:**

[Specification.](#)

### 3.11.2.9 `int sigismember (const sigset_t * set, int sig)`

Test for a signal in a signal set.

This service tests whether the signal number *sig* is member of the signal set pointed to by *set*.

**Parameters:**

*set* address of a signal set;

*sig* tested signal number.

**Return values:**

0 on success;

-1 with *errno* set if:

- EINVAL, *sig* is not a valid signal number.

**See also:**

[Specification.](#)

### 3.11.2.10 `int sigpending (sigset_t * set)`

Examine pending signals.

This service stores, at the address *set*, the set of signals that are currently blocked and have been received by the calling thread.

**Parameters:**

*set* address where the set of blocked and received signals are stored on success.

**Return values:**

0 on success;

-1 with *errno* set if:

- EPERM, the calling context is invalid.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread.

**See also:**

[Specification.](#)

### 3.11.2.11 `int sigtimedwait (const sigset_t *__restrict__ set, siginfo_t *__restrict__ info, const struct timespec *__restrict__ timeout)`

Wait during a bounded time for signals.

This service is equivalent to the [sigwaitinfo\(\)](#) service, except that the calling thread is only blocked until the timeout specified by *timeout* expires.

**Parameters:**

*set* set of signals to wait for;

*info* address where the received **siginfo\_t** object will be stored on success;

*timeout* the timeout, expressed as a time interval.

**Return values:**

0 on success;

-1 with *errno* set if:

- EINVAL, the specified timeout is invalid;
- EPERM, the caller context is invalid;
- EINVAL, a signal in *set* is not currently blocked;
- EAGAIN, no signal was received and the specified timeout expired.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread.

**See also:**

[Specification.](#)



### 3.11.2.12 `int sigwait (const sigset_t * set, int * sig)`

Wait for signals.

This service blocks a Xenomai kernel-space POSIX skin thread until a signal of the set *set* is received. If a signal in *set* is not currently blocked by the calling thread, this service returns immediately with an error. The signal received is stored at the address *sig*.

If a signal of the set *set* was already pending, it is cleared and this service returns immediately.

Signals are received in priority order, i.e. from SIGRTMIN to SIGRTMAX, then from 1 to SIGRTMIN-1.

#### Parameters:

*set* set of signals to wait for;

*sig* address where the received signal will be stored on success.

#### Returns:

0 on success;

an error number if:

- EPERM, the caller context is invalid;
- EINVAL, a signal in *set* is not currently blocked.

#### Valid contexts:

- Xenomai POSIX skin kernel-space thread.

#### See also:

[Specification.](#)

### 3.11.2.13 `int sigwaitinfo (const sigset_t *__restrict set, siginfo_t *__restrict info)`

Wait for signals.

This service is equivalent to the [sigwait\(\)](#) service, except that it returns, at the address *info*, the **siginfo\_t** object associated with the received signal instead of only returning the signal number.

#### Parameters:

*set* set of signals to wait for;

*info* address where the received **siginfo\_t** object will be stored on success.

#### Return values:

0 on success;

-1 with *errno* set if:

- EPERM, the caller context is invalid;
- EINVAL, a signal in *set* is not currently blocked.

#### Valid contexts:

- Xenomai POSIX skin kernel-space thread.

#### See also:

[Specification.](#)

## 3.12 Threads management services.

### 3.12.1 Detailed Description

Threads management services.

See also:

[Specification.](#)

### Modules

- [Thread cancellation.](#)  
*Thread cancellation.*
- [Threads scheduling services.](#)  
*Thread scheduling services.*
- [Thread creation attributes.](#)  
*Thread creation attributes.*

### Functions

- int [pthread\\_once](#) (pthread\_once\_t \*once, void(\*init\_routine)(void))  
*Execute an initialization routine.*
- int [pthread\\_create](#) (pthread\_t \*tid, const pthread\_attr\_t \*attr, void \*(\*start)(void \*), void \*arg)  
*Create a thread.*
- int [pthread\\_detach](#) (pthread\_t thread)  
*Detach a running thread.*
- int [pthread\\_equal](#) (pthread\_t t1, pthread\_t t2)  
*Compare thread identifiers.*
- void [pthread\\_exit](#) (void \*value\_ptr)  
*Terminate the current thread.*
- int [pthread\\_join](#) (pthread\_t thread, void \*\*value\_ptr)  
*Wait for termination of a specified thread.*
- pthread\_t [pthread\\_self](#) (void)  
*Get the identifier of the calling thread.*
- int [pthread\\_make\\_periodic\\_np](#) (pthread\_t thread, struct timespec \*starttp, struct timespec \*periodtp)  
*Make a thread periodic.*

- int [pthread\\_wait\\_np](#) (unsigned long \*overruns\_r)  
*Wait for current thread next period.*
- int [pthread\\_set\\_mode\\_np](#) (int clrmask, int setmask)  
*Set the mode of the current thread.*
- int [pthread\\_set\\_name\\_np](#) (pthread\_t thread, const char \*name)  
*Set a thread name.*

### 3.12.2 Function Documentation

#### 3.12.2.1 int [pthread\\_create](#) (pthread\_t \* *tid*, const pthread\_attr\_t \* *attr*, void (\*)(void \*) *start*, void \* *arg*)

Create a thread.

This service create a thread. The created thread may be used with all POSIX skin services.

The new thread run the *start* routine, with the *arg* argument.

The new thread signal mask is inherited from the current thread, if it was also created with [pthread\\_create\(\)](#), otherwise the new thread signal mask is empty.

Other attributes of the new thread depend on the *attr* argument. If *attr* is null, default values for these attributes are used. See [Thread creation attributes](#). for a definition of thread creation attributes and their default values.

Returning from the *start* routine has the same effect as calling [pthread\\_exit\(\)](#) with the return value.

##### Parameters:

*tid* address where the identifier of the new thread will be stored on success;

*attr* thread attributes;

*start* thread routine;

*arg* thread routine argument.

##### Returns:

0 on success;

an error number if:

- EINVAL, *attr* is invalid;
- EAGAIN, insufficient memory exists in the system heap to create a new thread, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ;
- EINVAL, thread attribute *inheritsched* is set to PTHREAD\_INHERIT\_SCHED and the calling thread does not belong to the POSIX skin;

##### See also:

[Specification](#).

### 3.12.2.2 `int pthread_detach(pthread_t thread)`

Detach a running thread.

This service detaches a joinable thread. A detached thread is a thread which control block is automatically reclaimed when it terminates. The control block of a joinable thread, on the other hand, is only reclaimed when joined with the service `pthread_join()`.

If some threads are currently blocked in the `pthread_join()` service with *thread* as a target, they are unblocked and `pthread_join()` returns EINVAL.

**Parameters:**

*thread* target thread.

**Returns:**

- 0 on success;
- an error number if:
  - ESRCH, *thread* is an invalid thread identifier;
  - EINVAL, *thread* is not joinable.

**See also:**

[Specification.](#)

### 3.12.2.3 `int pthread_equal(pthread_t t1, pthread_t t2)`

Compare thread identifiers.

This service compare the thread identifiers *t1* and *t2*. No attempt is made to check the threads for existence. In order to check if a thread exists, the `pthread_kill()` service should be used with the signal number 0.

**Parameters:**

- t1* thread identifier;
- t2* other thread identifier.

**Returns:**

- a non zero value if the thread identifiers are equal;
- 0 otherwise.

**See also:**

[Specification.](#)

### 3.12.2.4 `void pthread_exit(void * value_ptr)`

Terminate the current thread.

This service terminate the current thread with the return value *value\_ptr*. If the current thread is joinable, the return value is returned to any thread joining the current thread with the `pthread_join()` service.

When a thread terminates, cancellation cleanup handlers are executed in the reverse order that they were pushed. Then, thread-specific data destructors are executed.

**Parameters:**

*value\_ptr* thread return value.

**See also:**

[Specification.](#)

**3.12.2.5 int pthread\_join (pthread\_t thread, void \*\* value\_ptr)**

Wait for termination of a specified thread.

If the thread *thread* is running and joinable, this service blocks the calling thread until the thread *thread* terminates or detaches. In this case, the calling context must be a blockable context (i.e. a Xenomai thread without the scheduler locked) or the root thread (i.e. a module initialization or cleanup routine). When *thread* terminates, the calling thread is unblocked and its return value is stored at\* the address *value\_ptr*.

If, on the other hand, the thread *thread* has already finished execution, its return value is stored at the address *value\_ptr* and this service returns immediately. In this case, this service may be called from any context.

This service is a cancelation point for POSIX skin threads: if the calling thread is canceled while blocked in a call to this service, the cancelation request is honored and *thread* remains joinable.

Multiple simultaneous calls to [pthread\\_join\(\)](#) specifying the same running target thread block all the callers until the target thread terminates.

**Parameters:**

*thread* identifier of the thread to wait for;

*value\_ptr* address where the target thread return value will be stored on success.

**Returns:**

0 on success;

an error number if:

- ESRCH, *thread* is invalid;
- EDEADLK, attempting to join the calling thread;
- EINVAL, *thread* is detached;
- EPERM, the caller context is invalid.

**Valid contexts, if this service has to block its caller:**

- Xenomai kernel-space thread;
- kernel module initialization or cleanup routine;
- Xenomai user-space thread (switches to primary mode).

**See also:**

[Specification.](#)

**3.12.2.6 int pthread\_make\_periodic\_np (pthread\_t thread, struct timespec \* starttp, struct timespec \* periodtp)**

Make a thread periodic.

This service make the POSIX skin thread *thread* periodic.

This service is a non-portable extension of the POSIX interface.

**Parameters:**

*thread* thread identifier. This thread is immediately delayed until the first periodic release point is reached.

*starttp* start time, expressed as an absolute value of the CLOCK\_REALTIME clock. The affected thread will be delayed until this point is reached.

*periodtp* period, expressed as a time interval.

**Returns:**

0 on success;

an error number if:

- ESRCH, *thread* is invalid;
- ETIMEDOUT, the start time has already passed.

Rescheduling: always, until the *starttp* start time has been reached.

**3.12.2.7 int pthread\_once (pthread\_once\_t \* once, void(\*) (void) init\_routine)**

Execute an initialization routine.

This service may be used by libraries which need an initialization function to be called only once.

The function *init\_routine* will only be called, with no argument, the first time this service is called specifying the address *once*.

**Returns:**

0 on success;

an error number if:

- EINVAL, the object pointed to by *once* is invalid (it must have been initialized with PTHREAD\_ONCE\_INIT).

**See also:**

[Specification.](#)

**3.12.2.8 pthread\_t pthread\_self (void)**

Get the identifier of the calling thread.

This service returns the identifier of the calling thread.

**Returns:**

identifier of the calling thread;

NULL if the calling thread is not a POSIX skin thread.

**See also:**

[Specification.](#)

### 3.12.2.9 `int pthread_set_mode_np (int clrmask, int setmask)`

Set the mode of the current thread.

This service sets the mode of the calling thread. *clrmask* and *setmask* are two bit masks which are respectively cleared and set in the calling thread status. They are a bitwise OR of the following values:

- `PTHREAD_LOCK_SCHED`, when set, locks the scheduler, which prevents the current thread from being switched out by the scheduler until the scheduler is unlocked;
- `PTHREAD_SHIELD`, when set, activates the interrupt shield, which improve the execution determinism of the current thread by blocking Linux interrupts when it runs in secondary mode;
- `PTHREAD_RPIOFF`, when set, prevents the root Linux thread from inheriting the priority of the calling thread, when this thread is running in secondary mode;
- `PTHREAD_WARN_SW`, when set, cause the signal `SIGXCPU` to be sent to the current thread, whenever it involuntary switches to secondary mode;
- `PTHREAD_PRIMARY`, cause the migration of the current thread to primary mode.

`PTHREAD_LOCK_SCHED` is valid for any Xenomai thread, the other bits are only valid for Xenomai user-space threads.

This service is a non-portable extension of the POSIX interface.

**Parameters:**

*clrmask* set of bits to be cleared;

*setmask* set of bits to be set.

**Returns:**

0 on success;

an error number if:

- `EINVAL`, some bit in *clrmask* or *setmask* is invalid.

### 3.12.2.10 `int pthread_set_name_np (pthread_t thread, const char * name)`

Set a thread name.

This service set to *name*, the name of *thread*. This name is used for displaying information in `/proc/xenomai/sched`.

This service is a non-portable extension of the POSIX interface.

**Parameters:**

*thread* target thread;

*name* name of the thread.

**Returns:**

0 on success;

an error number if:

- `ESRCH`, *thread* is invalid.

### 3.12.2.11 `int pthread_wait_np (unsigned long * overruns_r)`

Wait for current thread next period.

If it is periodic, this service blocks the calling thread until the next period elapses.

This service is a cancelation point for POSIX skin threads.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*overruns\_r* address where the overruns count is returned in case of overrun.

#### Returns:

0 on success;

an error number if:

- EPERM, the calling context is invalid;
- EWOULDBLOCK, the calling thread is not periodic;
- EINTR, this service was interrupted by a signal;
- ETIMEDOUT, at least one overrun occurred.

#### Valid contexts:

- Xenomai kernel-space thread;
- Xenomai user-space thread (switches to primary mode).



## 3.13 Thread creation attributes.

### 3.13.1 Detailed Description

Thread creation attributes.

The services described in this section allow to set the attributes of a `pthread_attr_t` object, passed to the `pthread_create()` service in order to set the attributes of a created thread.

A `pthread_attr_t` object has to be initialized with `pthread_attr_init()` first, which sets attributes to their default values, i.e. in kernel-space:

- *detachstate* to `PTHREAD_CREATE_JOINABLE`,
- *stacksize* to `PTHREAD_STACK_MIN`,
- *inheritsched* to `PTHREAD_EXPLICIT_SCHED`,
- *schedpolicy* to `SCHED_OTHER`,
- *name* to `NULL` (only available in kernel-space),
- scheduling priority to the minimum,
- floating-point hardware enabled (only available in kernel-space),
- processor affinity set to all available processors (only available as a thread attribute in kernel-space).

In user-space, the attributes and their defaults values are those documented by the underlying threading library (LinuxThreads or NPTL).

### Functions

- `int pthread_attr_init(pthread_attr_t *attr)`  
*Initialize a thread attributes object.*
- `int pthread_attr_destroy(pthread_attr_t *attr)`  
*Destroy a thread attributes object.*
- `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate)`  
*Get detachstate attribute.*
- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`  
*Set detachstate attribute.*
- `int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize)`  
*Get stacksize attribute.*
- `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)`  
*Set stacksize attribute.*
- `int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched)`  
*Get inheritsched attribute.*

- int [pthread\\_attr\\_setinheritsched](#) (pthread\_attr\_t \*attr, int inheritsched)  
*Set inheritsched attribute.*
- int [pthread\\_attr\\_getschedpolicy](#) (const pthread\_attr\_t \*attr, int \*policy)  
*Get schedpolicy attribute.*
- int [pthread\\_attr\\_setschedpolicy](#) (pthread\_attr\_t \*attr, int policy)  
*Set schedpolicy attribute.*
- int [pthread\\_attr\\_getschedparam](#) (const pthread\_attr\_t \*attr, struct sched\_param \*par)  
*Get schedparam attribute.*
- int [pthread\\_attr\\_setschedparam](#) (pthread\_attr\_t \*attr, const struct sched\_param \*par)  
*Set schedparam attribute.*
- int [pthread\\_attr\\_getscope](#) (const pthread\_attr\_t \*attr, int \*scope)  
*Get contention scope attribute.*
- int [pthread\\_attr\\_setscope](#) (pthread\_attr\_t \*attr, int scope)  
*Set contention scope attribute.*
- int [pthread\\_attr\\_getname\\_np](#) (const pthread\_attr\_t \*attr, const char \*\*name)  
*Get name attribute.*
- int [pthread\\_attr\\_setname\\_np](#) (pthread\_attr\_t \*attr, const char \*name)  
*Set name attribute.*
- int [pthread\\_attr\\_getfp\\_np](#) (const pthread\_attr\_t \*attr, int \*fp)  
*Get the floating point attribute.*
- int [pthread\\_attr\\_setfp\\_np](#) (pthread\_attr\_t \*attr, int fp)  
*Set the floating point attribute.*
- int [pthread\\_attr\\_getaffinity\\_np](#) (const pthread\_attr\_t \*attr, xnarch\_cpumask\_t \*mask)  
*Get the processor affinity attribute.*
- int [pthread\\_attr\\_setaffinity\\_np](#) (pthread\_attr\_t \*attr, xnarch\_cpumask\_t mask)  
*Set the processor affinity attribute.*

### 3.13.2 Function Documentation

#### 3.13.2.1 int pthread\_attr\_destroy (pthread\_attr\_t \* attr)

Destroy a thread attributes object.

This service invalidates the attribute object pointed to by *attr*. The object becomes invalid for all services (they all return EINVAL) except [pthread\\_attr\\_init\(\)](#).

See also:

[Specification.](#)

### 3.13.2.2 `int pthread_attr_getaffinity_np (const pthread_attr_t * attr, xnarch_cpumask_t * mask)`

Get the processor affinity attribute.

This service stores, at the address *mask*, the value of the *affinity* attribute in the attribute object *attr*.

The *affinity* attributes is a bitmask where bits set indicate processor where a thread created with the attribute *attr* may run. The least significant bit corresponds to the first logical processor.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*attr* attribute object;

*mask* address where the value of the *affinity* attribute will be stored on success.

#### Returns:

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid.

#### Valid contexts:

- kernel module initialization or cleanup routine;
- Xenomai kernel-space thread.

### 3.13.2.3 `int pthread_attr_getdetachstate (const pthread_attr_t * attr, int * detachstate)`

Get detachstate attribute.

This service returns, at the address *detachstate*, the value of the *detachstate* attribute in the thread attribute object *attr*.

Valid values of this attribute are `PTHREAD_CREATE_JOINABLE` and `PTHREAD_CREATE_DETACHED`. A detached thread is a thread which control block is automatically reclaimed when it terminates. The control block of a joinable thread, on the other hand, is only reclaimed when joined with the service [pthread\\_join\(\)](#).

A thread that was created joinable may be detached after creation by using the [pthread\\_detach\(\)](#) service.

#### Parameters:

*attr* attribute object

*detachstate* address where the value of the detachstate attribute will be stored on success.

#### Returns:

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid;

#### See also:

[Specification.](#)

### 3.13.2.4 `int pthread_attr_getfp_np (const pthread_attr_t * attr, int * fp)`

Get the floating point attribute.

This service returns, at the address *fp*, the value of the *fp* attribute in the attribute object *attr*.

The *fp* attribute is a boolean attribute indicating whether a thread created with the attribute *attr* may use floating-point hardware.

This service is a non-portable extension of the POSIX interface.

**Parameters:**

*attr* attribute object;

*fp* address where the value of the *fp* attribute will be stored on success.

**Returns:**

0 on success;

an error number if:

- EINVAL, *attr* is invalid.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- Xenomai kernel-space thread.

### 3.13.2.5 `int pthread_attr_getinheritsched (const pthread_attr_t * attr, int * inheritsched)`

Get inheritsched attribute.

This service returns at the address *inheritsched* the value of the *inheritsched* attribute in the attribute object *attr*.

Threads created with this attribute set to PTHREAD\_INHERIT\_SCHED will use the same scheduling policy and priority as the thread calling `pthread_create()`. Threads created with this attribute set to PTHREAD\_EXPLICIT\_SCHED will use the value of the *schedpolicy* attribute as scheduling policy, and the value of the *schedparam* attribute as scheduling priority.

**Parameters:**

*attr* attribute object;

*inheritsched* address where the value of the *inheritsched* attribute will be stored on success.

**Returns:**

0 on success;

an error number if:

- EINVAL, *attr* is invalid.

**See also:**

[Specification.](#)

### 3.13.2.6 `int pthread_attr_getname_np (const pthread_attr_t * attr, const char ** name)`

Get name attribute.

This service stores, at the address *name*, the value of the *name* attribute in the attribute object *attr*.

The *name* attribute is the name under which a thread created with the attribute object *attr* will appear under `/proc/xenomai/sched`.

The name returned by this function is only valid until the name is changed with `pthread_attr_setname_np()` or the *attr* object is destroyed with `pthread_attr_destroy()`.

If *name* is `NULL`, a unique default name will be used.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*attr* attribute object;

*name* address where the value of the *name* attribute will be stored on success.

#### Returns:

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid.

#### Valid contexts:

- kernel module initialization or cleanup routine;
- Xenomai kernel-space thread.

### 3.13.2.7 `int pthread_attr_getschedparam (const pthread_attr_t * attr, struct sched_param * par)`

Get schedparam attribute.

This service stores, at the address *par*, the value of the *schedparam* attribute in the attribute object *attr*.

The only member of the `sched_param` structure used by this implementation is *sched\_priority*. Threads created with *attr* will use the value of this attribute as a scheduling priority if the attribute *inheritsched* is set to `PTHREAD_EXPLICIT_SCHED`. Valid priorities range from 1 to 99.

#### Parameters:

*attr* attribute object;

*par* address where the value of the *schedparam* attribute will be stored on success.

#### Returns:

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid.

#### See also:

[Specification.](#)

### 3.13.2.8 `int pthread_attr_getschedpolicy (const pthread_attr_t * attr, int * policy)`

Get schedpolicy attribute.

This service stores, at the address *policy*, the value of the *policy* attribute in the attribute object *attr*.

Threads created with the attribute object *attr* use the value of this attribute as scheduling policy if the *inheritsched* attribute is set to `PTHREAD_EXPLICIT_SCHED`. The value of this attribute is one of `SCHED_FIFO`, `SCHED_RR` or `SCHED_OTHER`.

**Parameters:**

*attr* attribute object;

*policy* address where the value of the *policy* attribute in the attribute object *attr* will be stored on success.

**Returns:**

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid.

**See also:**

[Specification.](#)

### 3.13.2.9 `int pthread_attr_getscope (const pthread_attr_t * attr, int * scope)`

Get contention scope attribute.

This service stores, at the address *scope*, the value of the *scope* attribute in the attribute object *attr*.

The *scope* attribute represents the scheduling contention scope of threads created with the attribute object *attr*. This implementation only supports the value `PTHREAD_SCOPE_SYSTEM`.

**Parameters:**

*attr* attribute object;

*scope* address where the value of the *scope* attribute will be stored on success.

**Returns:**

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid.

**See also:**

[Specification.](#)

### 3.13.2.10 `int pthread_attr_getstacksize (const pthread_attr_t * attr, size_t * stacksize)`

Get stacksize attribute.

This service stores, at the address *stacksize*, the value of the *stacksize* attribute in the attribute object *attr*.

The *stacksize* attribute is used as the stack size of the threads created using the attribute object *attr*.

**Parameters:**

*attr* attribute object;  
*stacksize* address where the value of the *stacksize* attribute will be stored on success.

**Returns:**

0 on success;  
an error number if:  
• EINVAL, *attr* is invalid.

**See also:**

[Specification.](#)

**3.13.2.11 int pthread\_attr\_init (pthread\_attr\_t \* attr)**

Initialize a thread attributes object.

This service initializes the thread creation attributes structure pointed to by *attr*. Attributes are set to their default values (see [Thread creation attributes.](#)).

If this service is called specifying a thread attributes object that was already initialized, the attributes object is reinitialized.

**Parameters:**

*attr* address of the thread attributes object to initialize.

**Returns:**

0.

**See also:**

[Specification.](#)

**3.13.2.12 int pthread\_attr\_setaffinity\_np (pthread\_attr\_t \* attr, xnarch\_cpumask\_t mask)**

Set the processor affinity attribute.

This service sets to *mask*, the value of the *affinity* attribute in the attribute object *attr*.

The *affinity* attributes is a bitmask where bits set indicate processor where a thread created with the attribute *attr* may run. The least significant bit corresponds to the first logical processor.

This service is a non-portable extension of the POSIX interface.

**Parameters:**

*attr* attribute object;  
*mask* address where the value of the *affinity* attribute will be stored on success.

**Returns:**

0 on success;  
an error number if:  
• EINVAL, *attr* is invalid.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- Xenomai kernel-space thread.

### 3.13.2.13 `int pthread_attr_setdetachstate(pthread_attr_t * attr, int detachstate)`

Set detachstate attribute.

This service sets to *detachstate* the value of the *detachstate* attribute in the attribute object *attr*.

Valid values of this attribute are `PTHREAD_CREATE_JOINABLE` and `PTHREAD_CREATE_DETACHED`. A detached thread is a thread which control block is automatically reclaimed when it terminates. The control block of a joinable thread, on the other hand, is only reclaimed when joined with the service [pthread\\_join\(\)](#).

A thread that was created joinable may be detached after creation by using the [pthread\\_detach\(\)](#) service.

#### Parameters:

*attr* attribute object;

*detachstate* value of the detachstate attribute.

#### Returns:

0 on success;

an error number if:

- `EINVAL`, the attribute object *attr* is invalid

#### See also:

[Specification](#).

### 3.13.2.14 `int pthread_attr_setfp_np(pthread_attr_t * attr, int fp)`

Set the floating point attribute.

This service set to *fp*, the value of the *fp* attribute in the attribute object *attr*.

The *fp* attribute is a boolean attribute indicating whether a thread created with the attribute *attr* may use floating-point hardware.

This service is a non-portable extension of the POSIX interface.

#### Parameters:

*attr* attribute object;

*fp* value of the *fp* attribute.

#### Returns:

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid.

#### Valid contexts:

- kernel module initialization or cleanup routine;
- Xenomai kernel-space thread.



### 3.13.2.15 `int pthread_attr_setinheritsched (pthread_attr_t * attr, int inheritsched)`

Set inheritsched attribute.

This service set to *inheritsched* the value of the *inheritsched* attribute in the attribute object *attr*.

Threads created with this attribute set to `PTHREAD_INHERIT_SCHED` will use the same scheduling policy and priority as the thread calling `pthread_create()`. Threads created with this attribute set to `PTHREAD_EXPLICIT_SCHED` will use the value of the *schedpolicy* attribute as scheduling policy, and the value of the *schedparam* attribute as scheduling priority.

**Parameters:**

*attr* attribute object;

*inheritsched* value of the *inheritsched* attribute, `PTHREAD_INHERIT_SCHED` or `PTHREAD_EXPLICIT_SCHED`.

**Returns:**

0 on success;

an error number if:

- `EINVAL`, *attr* or *inheritsched* is invalid.

**See also:**

[Specification](#).

### 3.13.2.16 `int pthread_attr_setname_np (pthread_attr_t * attr, const char * name)`

Set name attribute.

This service set to *name*, the value of the *name* attribute in the attribute object *attr*.

The *name* attribute is the name under which a thread created with the attribute object *attr* will appear under `/proc/xenomai/sched`.

If *name* is `NULL`, a unique default name will be used.

This service is a non-portable extension of the POSIX interface.

**Parameters:**

*attr* attribute object;

*name* value of the *name* attribute.

**Returns:**

0 on success;

an error number if:

- `EINVAL`, *attr* is invalid;
- `ENOMEM`, insufficient memory exists in the system heap to duplicate the name string, increase `CONFIG_XENO_OPT_SYS_HEAPSZ`.

**Valid contexts:**

- kernel module initialization or cleanup routine;
- Xenomai kernel-space thread.

### 3.13.2.17 `int pthread_attr_setschedparam (pthread_attr_t * attr, const struct sched_param * par)`

Set schedparam attribute.

This service set to *par*, the value of the *schedparam* attribute in the attribute object *attr*.

The only member of the **sched\_param** structure used by this implementation is *sched\_priority*. Threads created with *attr* will use the value of this attribute as a scheduling priority if the attribute *inheritsched* is set to `PTHREAD_EXPLICIT_SCHED`. Valid priorities range from 1 to 99.

#### Parameters:

- attr* attribute object;
- par* value of the *schedparam* attribute.

#### Returns:

- 0 on success;
- an error number if:
  - `EINVAL`, *attr* or *par* is invalid.

#### See also:

[Specification.](#)

### 3.13.2.18 `int pthread_attr_setschedpolicy (pthread_attr_t * attr, int policy)`

Set schedpolicy attribute.

This service set to *policy* the value of the *policy* attribute in the attribute object *attr*.

Threads created with the attribute object *attr* use the value of this attribute as scheduling policy if the *inheritsched* attribute is set to `PTHREAD_EXPLICIT_SCHED`. The value of this attribute is one of `SCHED_FIFO`, `SCHED_RR` or `SCHED_OTHER`.

#### Parameters:

- attr* attribute object;
- policy* value of the *policy* attribute.

#### Returns:

- 0 on success;
- an error number if:
  - `EINVAL`, *attr* or *policy* is invalid.

#### See also:

[Specification.](#)

### 3.13.2.19 `int pthread_attr_setscope (pthread_attr_t * attr, int scope)`

Set contention scope attribute.

This service set to *scope* the value of the *scope* attribute in the attribute object *attr*.

The *scope* attribute represents the scheduling contention scope of threads created with the attribute object *attr*. This implementation only supports the value `PTHREAD_SCOPE_SYSTEM`.

**Parameters:**

*attr* attribute object;  
*scope* value of the *scope* attribute.

**Returns:**

0 on success;  
an error number if:

- ENOTSUP, *scope* is an unsupported value of the *scope* attribute.
- EINVAL, *attr* is invalid.

**See also:**

[Specification.](#)

**3.13.2.20 int pthread\_attr\_setstacksize (pthread\_attr\_t \* attr, size\_t stacksize)**

Set stacksize attribute.

This service set to *stacksize*, the value of the *stacksize* attribute in the attribute object *attr*.

The *stacksize* attribute is used as the stack size of the threads created using the attribute object *attr*.

The minimum value for this attribute is PTHREAD\_STACK\_MIN.

**Parameters:**

*attr* attribute object;  
*stacksize* value of the *stacksize* attribute.

**Returns:**

0 on success;  
an error number if:

- EINVAL, *attr* or *stacksize* is invalid.

**See also:**

[Specification.](#)

## 3.14 Thread-specific data.

### 3.14.1 Detailed Description

Thread-specific data.

Programs often need global or static variables that have different values in different threads. Since threads share one memory space, this cannot be achieved with regular variables. Thread-specific data is the POSIX threads answer to this need.

Each thread possesses a private memory block, the thread-specific data area, or TSD area for short. This area is indexed by TSD keys. The TSD area associates values of type 'void \*' to TSD keys. TSD keys are common to all threads, but the value associated with a given TSD key can be different in each thread.

When a thread is created, its TSD area initially associates *NULL* with all keys.

The services documented here are valid in kernel-space context; when called in user-space, the underlying Linux threading library (LinuxThreads or NPTL) services are used.

### Functions

- int [pthread\\_key\\_create](#) (pthread\_key\_t \*key, void(\*destructor)(void \*))  
*Create a thread-specific data key.*
- int [pthread\\_setspecific](#) (pthread\_key\_t key, const void \*value)  
*Associate a thread-specific value with the specified key.*
- void \* [pthread\\_getspecific](#) (pthread\_key\_t key)  
*Get the thread-specific value bound to the specified key.*
- int [pthread\\_key\\_delete](#) (pthread\_key\_t key)  
*Delete a thread-specific data key.*

### 3.14.2 Function Documentation

#### 3.14.2.1 void\* pthread\_getspecific (pthread\_key\_t key)

Get the thread-specific value bound to the specified key.

This service returns the value associated, for the calling thread, with the key *key*.

#### Parameters:

*key* TSD key, obtained with [pthread\\_key\\_create\(\)](#).

#### Returns:

the value associated with *key*;  
NULL if the context is invalid.

#### Valid contexts:

- Xenomai POSIX skin kernel-space thread.

See also:

[Specification.](#)

#### 3.14.2.2 `int pthread_key_create (pthread_key_t * key, void(*) (void *) destructor)`

Create a thread-specific data key.

This service create a TSD key. The NULL value is associated for all threads with the new key and the new key is returned at the address *key*. If *destructor* is not null, it is executed when a thread is terminated as long as the datum associated with the key is not NULL, up to PTHREAD\_DESTRUCTOR\_ITERATIONS times.

##### Parameters:

*key* address where the new key will be stored on success;

*destructor* function to be invoked when a thread terminates and has a non NULL value associated with the new key.

##### Returns:

0 on success;

an error number if:

- EAGAIN, the total number of keys PTHREAD\_KEYS\_MAX TSD has been exceeded;
- ENOMEM, insufficient memory exists in the system heap to create a new key, increase CONFIG\_XENO\_OPT\_SYS\_HEAPSZ.

See also:

[Specification.](#)

#### 3.14.2.3 `int pthread_key_delete (pthread_key_t key)`

Delete a thread-specific data key.

This service deletes the TSD key *key*. Note that the key destructor function is not called, so, if any thread has a value associated with *key* that is a pointer to dynamically allocated memory, the application has to manage to free that memory by other means.

##### Parameters:

*key* the TSD key to be destroyed.

##### Returns:

0 on success;

an error number if:

- EINVAL, *key* is invalid.

See also:

[Specification.](#)

#### 3.14.2.4 `int pthread_setspecific(pthread_key_t key, const void * value)`

Associate a thread-specific value with the specified key.

This service associates, for the calling thread, the value *value* to the key *key*.

**Parameters:**

*key* TSD key, obtained with [pthread\\_key\\_create\(\)](#);

*value* value.

**Returns:**

0 on success;

an error number if:

- EPERM, the caller context is invalid;
- EINVAL, *key* is invalid.

**Valid contexts:**

- Xenomai POSIX skin kernel-space thread.

**See also:**

[Specification](#).

## Chapter 4

# Xenomai POSIX skin API File Documentation

### 4.1 ksrc/skins/posix/syscall.c File Reference

#### 4.1.1 Detailed Description

This file is part of the Xenomai project.

Copyright (C) 2005 Philippe Gerum <[rpm@xenomai.org](mailto:rpm@xenomai.org)> Copyright (C) 2005 Gilles Chantepredrix <[gilles.chantepredrix@laposte.net](mailto:gilles.chantepredrix@laposte.net)>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

```
#include <asm/xenomai/wrappers.h>
#include <nucleus/jhash.h>
#include <nucleus/ppd.h>
#include <posix/syscall.h>
#include <posix/posix.h>
#include <posix/thread.h>
#include <posix/mutex.h>
#include <posix/cond.h>
#include <posix/mq.h>
#include <posix/intr.h>
#include <posix/registry.h>
```

```
#include <posix/sem.h>
#include <posix/shm.h>
#include <posix/timer.h>
```



# Index

- clock\_getres
  - [posix\\_time, 11](#)
- clock\_gettime
  - [posix\\_time, 11](#)
- clock\_nanosleep
  - [posix\\_time, 12](#)
- clock\_settime
  - [posix\\_time, 12](#)
- Clocks and timers services., [10](#)
- close
  - [posix\\_shm, 56](#)
- Condition variables services., [17](#)
- ftruncate
  - [posix\\_shm, 57](#)
- Interruptions management services., [24](#)
- ksrc/skins/posix/syscall.c, [91](#)
- Message queues services., [29](#)
- mmap
  - [posix\\_shm, 57](#)
- mq\_close
  - [posix\\_mq, 30](#)
- mq\_getattr
  - [posix\\_mq, 30](#)
- mq\_notify
  - [posix\\_mq, 30](#)
- mq\_open
  - [posix\\_mq, 31](#)
- mq\_receive
  - [posix\\_mq, 32](#)
- mq\_send
  - [posix\\_mq, 33](#)
- mq\_setattr
  - [posix\\_mq, 34](#)
- mq\_timedreceive
  - [posix\\_mq, 34](#)
- mq\_timedsend
  - [posix\\_mq, 35](#)
- mq\_unlink
  - [posix\\_mq, 36](#)
- munmap
  - [posix\\_shm, 58](#)
- Mutex services., [37](#)
- nanosleep
  - [posix\\_time, 13](#)
- POSIX skin., [27](#)
- posix\_cancel
  - [pthread\\_cancel, 6](#)
  - [pthread\\_cleanup\\_pop, 6](#)
  - [pthread\\_cleanup\\_push, 7](#)
  - [pthread\\_setcancelstate, 7](#)
  - [pthread\\_setcanceltype, 8](#)
  - [pthread\\_testcancel, 8](#)
- posix\_cond
  - [pthread\\_cond\\_broadcast, 18](#)
  - [pthread\\_cond\\_destroy, 18](#)
  - [pthread\\_cond\\_init, 18](#)
  - [pthread\\_cond\\_signal, 19](#)
  - [pthread\\_cond\\_timedwait, 19](#)
  - [pthread\\_cond\\_wait, 20](#)
  - [pthread\\_condattr\\_destroy, 21](#)
  - [pthread\\_condattr\\_getclock, 21](#)
  - [pthread\\_condattr\\_getpshared, 21](#)
  - [pthread\\_condattr\\_init, 22](#)
  - [pthread\\_condattr\\_setclock, 22](#)
  - [pthread\\_condattr\\_setpshared, 23](#)
- posix\_intr
  - [pthread\\_intr\\_attach\\_np, 24](#)
  - [pthread\\_intr\\_control\\_np, 25](#)
  - [pthread\\_intr\\_detach\\_np, 26](#)
  - [pthread\\_intr\\_wait\\_np, 26](#)
- posix\_mq
  - [mq\\_close, 30](#)
  - [mq\\_getattr, 30](#)
  - [mq\\_notify, 30](#)
  - [mq\\_open, 31](#)
  - [mq\\_receive, 32](#)
  - [mq\\_send, 33](#)
  - [mq\\_setattr, 34](#)
  - [mq\\_timedreceive, 34](#)
  - [mq\\_timedsend, 35](#)
  - [mq\\_unlink, 36](#)
- posix\_mutex
  - [pthread\\_mutex\\_destroy, 38](#)
  - [pthread\\_mutex\\_init, 38](#)

- pthread\_mutex\_lock, 39
- pthread\_mutex\_timedlock, 39
- pthread\_mutex\_trylock, 40
- pthread\_mutex\_unlock, 41
- pthread\_mutexattr\_destroy, 41
- pthread\_mutexattr\_getprotocol, 42
- pthread\_mutexattr\_getpshared, 42
- pthread\_mutexattr\_gettype, 42
- pthread\_mutexattr\_init, 43
- pthread\_mutexattr\_setprotocol, 43
- pthread\_mutexattr\_setpshared, 44
- pthread\_mutexattr\_settype, 44
- posix\_sched
  - pthread\_getschedparam, 46
  - pthread\_setschedparam, 47
  - sched\_get\_priority\_max, 47
  - sched\_get\_priority\_min, 48
  - sched\_rr\_get\_interval, 48
  - sched\_yield, 49
- posix\_sem
  - sem\_close, 50
  - sem\_destroy, 51
  - sem\_getvalue, 51
  - sem\_init, 52
  - sem\_open, 52
  - sem\_post, 53
  - sem\_timedwait, 53
  - sem\_trywait, 54
  - sem\_unlink, 54
  - sem\_wait, 55
- posix\_shm
  - close, 56
  - ftruncate, 57
  - mmap, 57
  - munmap, 58
  - shm\_open, 59
  - shm\_unlink, 60
- posix\_signal
  - pthread\_kill, 63
  - pthread\_sigmask, 63
  - pthread\_sigqueue\_np, 64
  - sigaction, 64
  - sigaddset, 66
  - sigdelset, 66
  - sigemptyset, 66
  - sigfillset, 67
  - sigismember, 67
  - sigpending, 67
  - sigtimedwait, 68
  - sigwait, 68
  - sigwaitinfo, 69
- posix\_thread
  - pthread\_create, 71
  - pthread\_detach, 71
  - pthread\_equal, 72
  - pthread\_exit, 72
  - pthread\_join, 73
  - pthread\_make\_periodic\_np, 73
  - pthread\_once, 74
  - pthread\_self, 74
  - pthread\_set\_mode\_np, 74
  - pthread\_set\_name\_np, 75
  - pthread\_wait\_np, 75
- posix\_threadattr
  - pthread\_attr\_destroy, 78
  - pthread\_attr\_getaffinity\_np, 78
  - pthread\_attr\_getdetachstate, 79
  - pthread\_attr\_getfp\_np, 79
  - pthread\_attr\_getinheritsched, 80
  - pthread\_attr\_getname\_np, 80
  - pthread\_attr\_getschedparam, 81
  - pthread\_attr\_getschedpolicy, 81
  - pthread\_attr\_getscope, 82
  - pthread\_attr\_getstacksize, 82
  - pthread\_attr\_init, 83
  - pthread\_attr\_setaffinity\_np, 83
  - pthread\_attr\_setdetachstate, 83
  - pthread\_attr\_setfp\_np, 84
  - pthread\_attr\_setinheritsched, 84
  - pthread\_attr\_setname\_np, 85
  - pthread\_attr\_setschedparam, 85
  - pthread\_attr\_setschedpolicy, 86
  - pthread\_attr\_setscope, 86
  - pthread\_attr\_setstacksize, 87
- posix\_time
  - clock\_getres, 11
  - clock\_gettime, 11
  - clock\_nanosleep, 12
  - clock\_settime, 12
  - nanosleep, 13
  - timer\_create, 13
  - timer\_delete, 14
  - timer\_getoverrun, 14
  - timer\_gettime, 15
  - timer\_settime, 15
- posix\_tsd
  - pthread\_getspecific, 88
  - pthread\_key\_create, 89
  - pthread\_key\_delete, 89
  - pthread\_setspecific, 89
- pthread\_attr\_destroy
  - posix\_threadattr, 78
- pthread\_attr\_getaffinity\_np
  - posix\_threadattr, 78
- pthread\_attr\_getdetachstate
  - posix\_threadattr, 79
- pthread\_attr\_getfp\_np
  - posix\_threadattr, 79

- pthread\_attr\_getinheritsched
  - posix\_threadattr, [80](#)
- pthread\_attr\_getname\_np
  - posix\_threadattr, [80](#)
- pthread\_attr\_getschedparam
  - posix\_threadattr, [81](#)
- pthread\_attr\_getschedpolicy
  - posix\_threadattr, [81](#)
- pthread\_attr\_getscope
  - posix\_threadattr, [82](#)
- pthread\_attr\_getstacksize
  - posix\_threadattr, [82](#)
- pthread\_attr\_init
  - posix\_threadattr, [83](#)
- pthread\_attr\_setaffinity\_np
  - posix\_threadattr, [83](#)
- pthread\_attr\_setdetachstate
  - posix\_threadattr, [83](#)
- pthread\_attr\_setfp\_np
  - posix\_threadattr, [84](#)
- pthread\_attr\_setinheritsched
  - posix\_threadattr, [84](#)
- pthread\_attr\_setname\_np
  - posix\_threadattr, [85](#)
- pthread\_attr\_setschedparam
  - posix\_threadattr, [85](#)
- pthread\_attr\_setschedpolicy
  - posix\_threadattr, [86](#)
- pthread\_attr\_setscope
  - posix\_threadattr, [86](#)
- pthread\_attr\_setstacksize
  - posix\_threadattr, [87](#)
- pthread\_cancel
  - posix\_cancel, [6](#)
- pthread\_cleanup\_pop
  - posix\_cancel, [6](#)
- pthread\_cleanup\_push
  - posix\_cancel, [7](#)
- pthread\_cond\_broadcast
  - posix\_cond, [18](#)
- pthread\_cond\_destroy
  - posix\_cond, [18](#)
- pthread\_cond\_init
  - posix\_cond, [18](#)
- pthread\_cond\_signal
  - posix\_cond, [19](#)
- pthread\_cond\_timedwait
  - posix\_cond, [19](#)
- pthread\_cond\_wait
  - posix\_cond, [20](#)
- pthread\_condattr\_destroy
  - posix\_cond, [21](#)
- pthread\_condattr\_getclock
  - posix\_cond, [21](#)
- pthread\_condattr\_getpshared
  - posix\_cond, [21](#)
- pthread\_condattr\_init
  - posix\_cond, [22](#)
- pthread\_condattr\_setclock
  - posix\_cond, [22](#)
- pthread\_condattr\_setpshared
  - posix\_cond, [23](#)
- pthread\_create
  - posix\_thread, [71](#)
- pthread\_detach
  - posix\_thread, [71](#)
- pthread\_equal
  - posix\_thread, [72](#)
- pthread\_exit
  - posix\_thread, [72](#)
- pthread\_getschedparam
  - posix\_sched, [46](#)
- pthread\_getspecific
  - posix\_tsd, [88](#)
- pthread\_intr\_attach\_np
  - posix\_intr, [24](#)
- pthread\_intr\_control\_np
  - posix\_intr, [25](#)
- pthread\_intr\_detach\_np
  - posix\_intr, [26](#)
- pthread\_intr\_wait\_np
  - posix\_intr, [26](#)
- pthread\_join
  - posix\_thread, [73](#)
- pthread\_key\_create
  - posix\_tsd, [89](#)
- pthread\_key\_delete
  - posix\_tsd, [89](#)
- pthread\_kill
  - posix\_signal, [63](#)
- pthread\_make\_periodic\_np
  - posix\_thread, [73](#)
- pthread\_mutex\_destroy
  - posix\_mutex, [38](#)
- pthread\_mutex\_init
  - posix\_mutex, [38](#)
- pthread\_mutex\_lock
  - posix\_mutex, [39](#)
- pthread\_mutex\_timedlock
  - posix\_mutex, [39](#)
- pthread\_mutex\_trylock
  - posix\_mutex, [40](#)
- pthread\_mutex\_unlock
  - posix\_mutex, [41](#)
- pthread\_mutexattr\_destroy
  - posix\_mutex, [41](#)
- pthread\_mutexattr\_getprotocol
  - posix\_mutex, [42](#)

- pthread\_mutexattr\_getpshared
  - posix\_mutex, [42](#)
- pthread\_mutexattr\_gettype
  - posix\_mutex, [42](#)
- pthread\_mutexattr\_init
  - posix\_mutex, [43](#)
- pthread\_mutexattr\_setprotocol
  - posix\_mutex, [43](#)
- pthread\_mutexattr\_setpshared
  - posix\_mutex, [44](#)
- pthread\_mutexattr\_settype
  - posix\_mutex, [44](#)
- pthread\_once
  - posix\_thread, [74](#)
- pthread\_self
  - posix\_thread, [74](#)
- pthread\_set\_mode\_np
  - posix\_thread, [74](#)
- pthread\_set\_name\_np
  - posix\_thread, [75](#)
- pthread\_setcancelstate
  - posix\_cancel, [7](#)
- pthread\_setcanceltype
  - posix\_cancel, [8](#)
- pthread\_setschedparam
  - posix\_sched, [47](#)
- pthread\_setspecific
  - posix\_tsd, [89](#)
- pthread\_sigmask
  - posix\_signal, [63](#)
- pthread\_sigqueue\_np
  - posix\_signal, [64](#)
- pthread\_testcancel
  - posix\_cancel, [8](#)
- pthread\_wait\_np
  - posix\_thread, [75](#)
- posix\_sem, [52](#)
- sem\_post
  - posix\_sem, [53](#)
- sem\_timedwait
  - posix\_sem, [53](#)
- sem\_trywait
  - posix\_sem, [54](#)
- sem\_unlink
  - posix\_sem, [54](#)
- sem\_wait
  - posix\_sem, [55](#)
- Semaphores services., [50](#)
- Shared memory services., [56](#)
- shm\_open
  - posix\_shm, [59](#)
- shm\_unlink
  - posix\_shm, [60](#)
- sigaction
  - posix\_signal, [64](#)
- sigaddset
  - posix\_signal, [66](#)
- sigdelset
  - posix\_signal, [66](#)
- sigemptyset
  - posix\_signal, [66](#)
- sigfillset
  - posix\_signal, [67](#)
- sigismember
  - posix\_signal, [67](#)
- Signals services., [62](#)
- sigpending
  - posix\_signal, [67](#)
- sigtimedwait
  - posix\_signal, [68](#)
- sigwait
  - posix\_signal, [68](#)
- sigwaitinfo
  - posix\_signal, [69](#)
- Thread cancellation., [5](#)
- Thread creation attributes., [77](#)
- Thread-specific data., [88](#)
- Threads management services., [70](#)
- Threads scheduling services., [46](#)
- timer\_create
  - posix\_time, [13](#)
- timer\_delete
  - posix\_time, [14](#)
- timer\_getoverrun
  - posix\_time, [14](#)
- timer\_gettime
  - posix\_time, [15](#)
- timer\_settime
  - posix\_time, [15](#)
- posix\_sched, [47](#)
- posix\_sched, [48](#)
- posix\_sched, [48](#)
- posix\_sched, [49](#)
- sem\_close
  - posix\_sem, [50](#)
- sem\_destroy
  - posix\_sem, [51](#)
- sem\_getvalue
  - posix\_sem, [51](#)
- sem\_init
  - posix\_sem, [52](#)
- sem\_open