Xenomai Native skin API

2.5.0

Generated by Doxygen 1.6.1

Fri Jan 1 10:42:47 2010

Contents

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

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

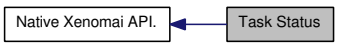
Chapter 4

Module Documentation

4.1 Task Status

Defines used to specify task state and/or mode.

Collaboration diagram for Task Status:



Defines

- #define T_BLOCKED XNPEND
  See XNPEND.
- #define T_DELAYED XNDELAY
  See XNDELAY.
- #define T_READY XNREADY
  See XNREADY.
- #define T_DORMANT XNDORMANT
  See XNDORMANT.
- #define T_STARTED XNSTARTED
  See XNSTARTED.
- #define T_BOOST XNBOOST
  See XNBOOST.
- #define T_LOCK XNLOCK
  See XNLOCK.
- #define T_NOSIG XNASDI
  See XNASDI.

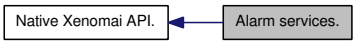
- #define T_WARNSW XNTRAPSW
  See XNTRAPSW.
- #define T_RPIOFF XNRPIOFF
  See XNRPIOFF.

4.1.1 Detailed Description

Defines used to specify task state and/or mode.

4.2 Alarm services.

Collaboration diagram for Alarm services.:



Files

- file alarm.c

  This file is part of the Xenomai project.

Functions

- int rt_alarm_create (RT_ALARM *alarm, const char *name, rt_alarm_t handler, void *cookie)

  Create an alarm object from kernel space.

- int rt_alarm_delete (RT_ALARM *alarm)

  Delete an alarm.

- int rt_alarm_start (RT_ALARM *alarm, RTIME value, RTIME interval)

  Start an alarm.

- int rt_alarm_stop (RT_ALARM *alarm)

  Stop an alarm.

- int rt_alarm_inquire (RT_ALARM *alarm, RT_ALARM_INFO *info)

  Inquire about an alarm.

- int rt_alarm_create (RT_ALARM *alarm, const char *name)

  Create an alarm object from user-space.

- int rt_alarm_wait (RT_ALARM *alarm)

  Wait for the next alarm shot.

4.2.1 Detailed Description

Alarms are general watchdog timers. Any Xenomai task may create any number of alarms and use them to run a user-defined handler, after a specified initial delay has elapsed. Alarms can be either one shot or periodic; in the latter case, the real-time kernel automatically reprograms the alarm for the next shot according to a user-defined interval value.

4.2.2 Function Documentation

4.2.2.1 int rt_alarm_create (RT_ALARM * *alarm*, const char * *name*)

Create an alarm object from user-space. Initializes an alarm object from a user-space application. Alarms can be made periodic or oneshot, depending on the reload interval value passed to rt_alarm_start() for them. In this mode, the basic principle is to define some alarm server task which routinely waits for the next incoming alarm event through the rt_alarm_wait() syscall.

Parameters:

alarm The address of an alarm descriptor Xenomai will use to store the alarm-related data. This descriptor must always be valid while the alarm is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the alarm. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created alarm.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the alarm.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- User-space task

Rescheduling: possible.

Note:

It is possible to combine kernel-based alarm handling with waiter threads pending on the same alarm object from user-space through the rt_alarm_wait() service. For this purpose, the rt_alarm_handler() routine which is internally invoked to wake up alarm servers in user-space is accessible to user-provided alarm handlers in kernel space, and should be called from there in order to unblock any thread sleeping on the rt_alarm_wait() service.

4.2.2.2 int rt_alarm_create (RT_ALARM * *alarm*, const char * *name*, rt_alarm_t *handler*, void * *cookie*)

Create an alarm object from kernel space. Create an object triggering an alarm routine at a specified time in the future. Alarms can be made periodic or oneshot, depending on the reload interval value passed to rt_alarm_start() for them. In kernel space, alarms are immediately notified on behalf of the timer interrupt to a user-defined handler.

Parameters:

alarm The address of an alarm descriptor Xenomai will use to store the alarm-related data. This descriptor must always be valid while the alarm is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the alarm. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created alarm.

handler The address of the routine to call when the alarm expires. This routine will be passed the address of the current alarm descriptor, and the opaque *cookie*.

cookie A user-defined opaque cookie the real-time kernel will pass to the alarm handler as its second argument.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the alarm.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task

Rescheduling: possible.

Note:

It is possible to combine kernel-based alarm handling with waiter threads pending on the same alarm object from user-space through the rt_alarm_wait() service. For this purpose, the rt_alarm_handler() routine which is internally invoked to wake up alarm servers in user-space is accessible to user-provided alarm handlers in kernel space, and should be called from there in order to unblock any thread sleeping on the rt_alarm_wait() service.

References rt_alarm_delete().

4.2.2.3 int rt_alarm_delete (RT_ALARM * *alarm*)

Delete an alarm. Destroy an alarm. An alarm exists in the system since rt_alarm_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

alarm The descriptor address of the affected alarm.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *alarm* is not a alarm descriptor.
- -EIDRM is returned if *alarm* is a deleted alarm descriptor.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: never.

Referenced by rt_alarm_create().

4.2.2.4 int rt_alarm_inquire (RT_ALARM * *alarm*, RT_ALARM_INFO * *info*)

Inquire about an alarm. Return various information about the status of a given alarm.

Parameters:

alarm The descriptor address of the inquired alarm.
info The address of a structure the alarm information will be written to.

The expiration date returned in the information block is converted to the current time unit. The special value TM_INFINITE is returned if *alarm* is currently inactive/stopped. In single-shot mode, it might happen that the alarm has already expired when this service is run (even if the associated handler has not been fired yet); in such a case, 1 is returned.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *alarm* is not a alarm descriptor.
- -EIDRM is returned if *alarm* is a deleted alarm descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.2.2.5 int rt_alarm_start (RT_ALARM * *alarm*, RTIME *value*, RTIME *interval*)

Start an alarm. Program the trigger date of an alarm object. An alarm can be either periodic or oneshot, depending on the reload value passed to this routine. The given alarm must have been previously created by a call to rt_alarm_create().

Alarm handlers are always called on behalf of Xenomai's internal timer tick handler, so the Xenomai services which can be called from such handlers are restricted to the set of services available on behalf of any ISR.

This service overrides any previous setup of the expiry date and reload interval for the given alarm.

Parameters:

alarm The descriptor address of the affected alarm.

value The relative date of the initial alarm shot, expressed in clock ticks (see note).

interval The reload value of the alarm. It is a periodic interval value to be used for reprogramming the next alarm shot, expressed in clock ticks (see note). If *interval* is equal to TM_INFINITE, the alarm will not be reloaded after it has expired.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *alarm* is not a alarm descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

Note:

The initial *value* and *interval* will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.2.2.6 int rt_alarm_stop (RT_ALARM * *alarm*)

Stop an alarm. Disarm an alarm object previously armed using rt_alarm_start() so that it will not trigger until is is re-armed.

Parameters:

alarm The descriptor address of the released alarm.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *alarm* is not a alarm descriptor.
- -EIDRM is returned if *alarm* is a deleted alarm descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.2.2.7 int rt_alarm_wait (RT_ALARM * *alarm*)

Wait for the next alarm shot. This user-space only call allows the current task to suspend execution until the specified alarm triggers. The priority of the current task is raised above all other Xenomai tasks - except those also undergoing an alarm or interrupt wait (see rt_intr_wait()) - so that it would preempt any of them under normal circumstances (i.e. no scheduler lock).

Parameters:

alarm The descriptor address of the awaited alarm.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *alarm* is not an alarm descriptor.
- -EPERM is returned if this service was called from a context which cannot sleep (e.g. interrupt, non-realtime or scheduler locked).
- -EIDRM is returned if *alarm* is a deleted alarm descriptor, including if the deletion occurred while the caller was waiting for its next shot.
- -EINTR is returned if rt_task_unblock() has been called for the current task before the next alarm shot.

Environments:

This service can be called from:

- User-space task

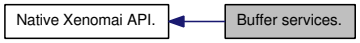
Rescheduling: always.

Examples:

user_alarm.c.

4.3 Buffer services.

Collaboration diagram for Buffer services.:



Files

- file buffer.c

  This file is part of the Xenomai project.

Functions

- int rt_buffer_create (RT_BUFFER *bf, const char *name, size_t bufsz, int mode)

  Create a buffer.

- int rt_buffer_delete (RT_BUFFER *bf)

  Delete a buffer.

- ssize_t rt_buffer_write (RT_BUFFER *bf, const void *ptr, size_t len, RTIME timeout)

  Write to a buffer.

- ssize_t rt_buffer_write_until (RT_BUFFER *bf, const void *ptr, size_t len, RTIME timeout)

  Write to a buffer (with absolute timeout date).

- ssize_t rt_buffer_read (RT_BUFFER *bf, void *ptr, size_t len, RTIME timeout)

  Read from a buffer.

- int rt_buffer_clear (RT_BUFFER *bf)

  Clear a buffer.

- int rt_buffer_inquire (RT_BUFFER *bf, RT_BUFFER_INFO *info)

  Inquire about a buffer.

- int rt_buffer_bind (RT_BUFFER *bf, const char *name, RTIME timeout)

  Bind to a buffer.

- static int rt_buffer_unbind (RT_BUFFER *bf)

  Unbind from a buffer.

4.3.1 Detailed Description

Buffer services.

A buffer is a lightweight IPC object, implementing a fast, one-way Producer-Consumer data path. All messages written are buffered in a single memory area in strict FIFO order, until read either in blocking or non-blocking mode.

Message are always atomically handled on the write side (i.e. no interleave, no short writes), whilst only complete messages are normally returned to the read side. However, short reads may happen under a well-defined situation (see note in rt_buffer_read()), albeit they can be fully avoided by proper use of the buffer.

4.3.2 Function Documentation

4.3.2.1 int rt_buffer_bind (RT_BUFFER * *bf*, const char * *name*, RTIME *timeout*)

Bind to a buffer. This user-space only service retrieves the uniform descriptor of a given Xenomai buffer identified by its symbolic name. If the buffer does not exist on entry, this service blocks the caller until a buffer of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the buffer to bind to.

bf The address of a buffer descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *bf* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanosebfs otherwise.

4.3.2.2 int rt_buffer_clear (RT_BUFFER * *bf*)

Clear a buffer. Empties a buffer from any data.

Parameters:

bf The descriptor address of the cleared buffer.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *bf* is not a buffer descriptor.
- -EIDRM is returned if *bf* is a deleted buffer descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible, as a consequence of resuming tasks that wait for buffer space in rt_buffer_write().

4.3.2.3 int rt_buffer_create (RT_BUFFER * *bf*, const char * *name*, size_t *bufsz*, int *mode*)

Create a buffer. Create a synchronization object that allows tasks to send and receive data asynchronously via a memory buffer. Data may be of an arbitrary length, albeit this IPC is best suited for small to medium-sized messages, since data always have to be copied to the buffer during transit. Large messages may be more efficiently handled by message queues (RT_QUEUE) via rt_queue_send()/rt_queue_receive() services.

Parameters:

bf The address of a buffer descriptor Xenomai will use to store the buffer-related data. This descriptor must always be valid while the buffer is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the buffer. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created buffer.

bufsz The size of the buffer space available to hold data. The required memory is obtained from the system heap.

mode The buffer creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new buffer:

- B_FIFO makes tasks pend in FIFO order for reading data from the buffer.
- B_PRIO makes tasks pend in priority order for reading data from the buffer.

This parameter also applies to tasks blocked on the buffer's output queue (see rt_buffer_write()).

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the buffer.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task (switches to secondary mode)

Rescheduling: possible.

References rt_buffer_delete().

4.3.2.4 int rt_buffer_delete (RT_BUFFER * *bf*)

Delete a buffer. Destroy a buffer and release all the tasks currently pending on it. A buffer exists in the system since rt_buffer_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

bf The descriptor address of the buffer to delete.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *bf* is not a buffer descriptor.
- -EIDRM is returned if *bf* is a deleted buffer descriptor.

- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task (switches to secondary mode)

Rescheduling: possible.

Referenced by rt_buffer_create().

4.3.2.5 int rt_buffer_inquire (RT_BUFFER * *bf*, RT_BUFFER_INFO * *info*)

Inquire about a buffer. Return various information about the status of a given buffer.

Parameters:

bf The descriptor address of the inquired buffer.

info The address of a structure the buffer information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *bf* is not a buffer descriptor.
- -EIDRM is returned if *bf* is a deleted buffer descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.3.2.6 ssize_t rt_buffer_read (RT_BUFFER * *bf*, void * *ptr*, size_t *len*, RTIME *timeout*)

Read from a buffer. Reads the next message from the specified buffer. If no message is available on entry, the caller is allowed to block until enough data is written to the buffer.

Parameters:

bf The descriptor address of the buffer to read from.

ptr A pointer to a memory area which will be written upon success with the received data.

len The length in bytes of the memory area pointed to by *ptr*. Under normal circumstances, rt_buffer_read() only returns entire messages as specified by the *len* argument, or an error value. However, short reads are allowed when a potential deadlock situation is detected (see note below).

timeout The number of clock ticks to wait for a message to be available from the buffer (see note). Passing TM_INFINITE causes the caller to block indefinitely until enough data is available. Passing TM_NONBLOCK causes the service to return immediately without blocking in case not enough data is available.

Returns:

The number of bytes read from the buffer is returned upon success. Otherwise:

- -ETIMEDOUT is returned if *timeout* is different from TM_NONBLOCK and not enough data is available within the specified amount of time to form a complete message.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and not enough data is immediately available on entry to form a complete message.
- -EINTR is returned if rt_task_unblock() has been called for the reading task before enough data became available to form a complete message.
- -EINVAL is returned if *bf* is not a buffer descriptor, or *len* is greater than the actual buffer length.
- -EIDRM is returned if *bf* is a deleted buffer descriptor.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).
- -ENOMEM is returned if not enough memory is available from the system heap to hold a temporary copy of the message (user-space call only).

Note:

A short read (i.e. fewer bytes returned than requested by *len*) may happen whenever a pathological use of the buffer is encountered. This condition only arises when the system detects that one or more writers are waiting for sending data, while a reader would have to wait for receiving a complete message at the same time. For instance, consider the following sequence, involving a 1024-byte buffer (bf) and two threads:

writer thread > rt_write_buffer(&bf, ptr, 1, TM_INFINITE); (one byte to read, 1023 bytes available for sending) writer thread > rt_write_buffer(&bf, ptr, 1024, TM_INFINITE); (writer blocks - no space for another 1024-byte message) reader thread > rt_read_buffer(&bf, ptr, 1024, TM_-INFINITE); (short read - a truncated (1-byte) message is returned)

In order to prevent both threads to wait for each other indefinitely, a short read is allowed, which may be completed by a subsequent call to rt_buffer_read() or rt_buffer_read_until(). If that case arises, thread priorities, buffer and/or message lengths should likely be fixed, in order to eliminate such condition.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine (non-blocking call only)
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied and no task is waiting for buffer space to be released for the same buffer (see rt_buffer_write()), or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.3.2.7 int rt_buffer_unbind (RT_BUFFER * *bf*) [inline, static]

Unbind from a buffer. This user-space only service unbinds the calling task from the buffer object previously retrieved by a call to rt_buffer_bind().

Parameters:

bf The address of a buffer descriptor to unbind from.

Returns:

0 is always returned.

This service can be called from:

- User-space task.

Rescheduling: never.

4.3.2.8 ssize_t rt_buffer_write (RT_BUFFER * *bf*, const void * *ptr*, size_t *len*, RTIME *timeout*)

Write to a buffer. Writes a message to the specified buffer. If not enough buffer space is available on entry to hold the message, the caller is allowed to block until enough room is freed. Data written by rt_buffer_write() calls can be read in FIFO order by subsequent rt_buffer_read() calls. Messages sent via rt_buffer_write() are handled atomically (no interleave, no short writes).

Parameters:

bf The descriptor address of the buffer to write to.

ptr The address of the message data to be written to the buffer.

len The length in bytes of the message data. Zero is a valid value, in which case the buffer is left untouched, and zero is returned to the caller. No partial message is ever sent.

timeout The number of clock ticks to wait for enough buffer space to be available to hold the message (see note). Passing TM_INFINITE causes the caller to block indefinitely until enough buffer space is available. Passing TM_NONBLOCK causes the service to return immediately without blocking in case of buffer space shortage.

Returns:

The number of bytes written to the buffer is returned upon success. Otherwise:

- -ETIMEDOUT is returned if *timeout* is different from TM_NONBLOCK and no buffer space is available within the specified amount of time to hold the message.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no buffer space is immediately available on entry to hold the message.
- -EINTR is returned if rt_task_unblock() has been called for the writing task before enough buffer space became available to hold the message.
- -EINVAL is returned if *bf* is not a buffer descriptor, or *len* is greater than the actual buffer length.
- -EIDRM is returned if *bf* is a deleted buffer descriptor.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).
- -ENOMEM is returned if not enough memory is available from the system heap to hold a temporary copy of the message (user-space call only).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine (non-blocking call only)
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied and no task is waiting for messages on the same buffer, or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.3.2.9 ssize_t rt_buffer_write_until (RT_BUFFER * *bf*, const void * *ptr*, size_t *len*, RTIME *timeout*)

Write to a buffer (with absolute timeout date). Writes a message to the specified buffer. If not enough buffer space is available on entry to hold the message, the caller is allowed to block until enough room is freed, or a timeout elapses.

Parameters:

bf The descriptor address of the buffer to write to.

ptr The address of the message data to be written to the buffer.

len The length in bytes of the message data. Zero is a valid value, in which case the buffer is left untouched, and zero is returned to the caller.

timeout The absolute date specifying a time limit to wait for enough buffer space to be available to hold the message (see note). Passing TM_INFINITE causes the caller to block indefinitely until enough buffer space is available. Passing TM_NONBLOCK causes the service to return immediately without blocking in case of buffer space shortage.

Returns:

The number of bytes written to the buffer is returned upon success. Otherwise:

- -ETIMEDOUT is returned if the absolute *timeout* date is reached before enough buffer space is available to hold the message.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no buffer space is immediately available on entry to hold the message.
- -EINTR is returned if rt_task_unblock() has been called for the writing task before enough buffer space became available to hold the message.
- -EINVAL is returned if *bf* is not a buffer descriptor, or *len* is greater than the actual buffer length.
- -EIDRM is returned if *bf* is a deleted buffer descriptor.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).
- -ENOMEM is returned if not enough memory is available from the system heap to hold a temporary copy of the message (user-space call only).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine (non-blocking call only)
- Kernel-based task
- User-space task (switches to primary mode)

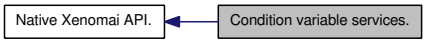
Rescheduling: always unless the request is immediately satisfied and no task is waiting for messages on the same buffer, or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.4 Condition variable services.

Collaboration diagram for Condition variable services.:



Files

- file cond.c
  This file is part of the Xenomai project.

Functions

- int rt_cond_create (RT_COND *cond, const char *name)
  Create a condition variable.
- int rt_cond_delete (RT_COND *cond)
  Delete a condition variable.
- int rt_cond_signal (RT_COND *cond)
  Signal a condition variable.
- int rt_cond_broadcast (RT_COND *cond)
  Broadcast a condition variable.
- int rt_cond_wait (RT_COND *cond, RT_MUTEX *mutex, RTIME timeout)
  Wait on a condition.
- int rt_cond_wait_until (RT_COND *cond, RT_MUTEX *mutex, RTIME timeout)
  Wait on a condition (with absolute timeout date).
- int rt_cond_inquire (RT_COND *cond, RT_COND_INFO *info)
  Inquire about a condition variable.
- int rt_cond_bind (RT_COND *cond, const char *name, RTIME timeout)
  Bind to a condition variable.
- static int rt_cond_unbind (RT_COND *cond)
  Unbind from a condition variable.

4.4.1 Detailed Description

Condition variable services.

A condition variable is a synchronization object which allows tasks 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, blocking the task execution until another task signals the condition. A condition variable must always be associated with a mutex, to avoid a well-known race condition where a task prepares to wait on a condition variable and another task signals the condition just before the first task actually waits on it.

4.4.2 Function Documentation

4.4.2.1 int rt_cond_bind (RT_COND * *cond*, const char * *name*, RTIME *timeout*)

Bind to a condition variable. This user-space only service retrieves the uniform descriptor of a given Xenomai condition variable identified by its symbolic name. If the condition variable does not exist on entry, this service blocks the caller until a condition variable of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the condition variable to bind to.

cond The address of a condition variable descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *cond* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.4.2.2 int rt_cond_broadcast (RT_COND * *cond*)

Broadcast a condition variable. If the condition variable is pended, all tasks currently waiting on it are immediately unblocked.

Parameters:

cond The descriptor address of the affected condition variable.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *cond* is not a condition variable descriptor.
- -EIDRM is returned if *cond* is a deleted condition variable descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.4.2.3 int rt_cond_create (RT_COND * *cond*, const char * *name*)

Create a condition variable. Create a synchronization object that allows tasks to suspend execution until some predicate on shared data is satisfied.

Parameters:

cond The address of a condition variable descriptor Xenomai will use to store the variable-related data. This descriptor must always be valid while the variable is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the condition variable. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created variable.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the condition variable.
- -EEXIST is returned if the *name* is already in use by some registered object.

- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

References rt_cond_delete().

4.4.2.4 int rt_cond_delete (RT_COND * *cond*)

Delete a condition variable. Destroy a condition variable and release all the tasks currently pending on it. A condition variable exists in the system since rt_cond_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

cond The descriptor address of the affected condition variable.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *cond* is not a condition variable descriptor.
- -EIDRM is returned if *cond* is a deleted condition variable descriptor.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Referenced by rt_cond_create().

4.4.2.5 int rt_cond_inquire (RT_COND * *cond*, RT_COND_INFO * *info*)

Inquire about a condition variable. Return various information about the status of a given condition variable.

Parameters:

cond The descriptor address of the inquired condition variable.

info The address of a structure the condition variable information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *cond* is not a condition variable descriptor.
- -EIDRM is returned if *cond* is a deleted condition variable descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.4.2.6 int rt_cond_signal (RT_COND * *cond*)

Signal a condition variable. If the condition variable is pended, the first waiting task (by queuing priority order) is immediately unblocked.

Parameters:

cond The descriptor address of the affected condition variable.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *cond* is not a condition variable descriptor.
- -EIDRM is returned if *cond* is a deleted condition variable descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.4.2.7 int rt_cond_unbind (RT_COND * *cond*) [inline, static]

Unbind from a condition variable. This user-space only service unbinds the calling task from the condition variable object previously retrieved by a call to rt_cond_bind().

Parameters:

cond The address of a condition variable descriptor to unbind from.

Returns:

0 is always returned.

This service can be called from:

- User-space task.

Rescheduling: never.

4.4.2.8 int rt_cond_wait (RT_COND * *cond*, RT_MUTEX * *mutex*, RTIME *timeout*)

Wait on a condition. This service atomically release the mutex and causes the calling task to block on the specified condition variable. The caller will be unblocked when the variable is signaled, and the mutex re-acquired before returning from this service.

Tasks pend on condition variables by priority order.

Parameters:

cond The descriptor address of the affected condition variable.

mutex The descriptor address of the mutex protecting the condition variable.

timeout The number of clock ticks to wait for the condition variable to be signaled (see note). Passing TM_INFINITE causes the caller to block indefinitely until the condition variable is signaled.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *mutex* is not a mutex descriptor, or *cond* is not a condition variable descriptor.
- -EIDRM is returned if *mutex* or *cond* is a deleted object descriptor, including if the deletion occurred while the caller was sleeping on the variable.

- -ETIMEDOUT is returned if *timeout* expired before the condition variable has been signaled.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the condition variable has been signaled.
- -EWOULDBLOCK is returned if *timeout* equals TM_NONBLOCK.

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.4.2.9 int rt_cond_wait_until (RT_COND * *cond*, RT_MUTEX * *mutex*, RTIME *timeout*)

Wait on a condition (with absolute timeout date). This service atomically release the mutex and causes the calling task to block on the specified condition variable. The caller will be unblocked when the variable is signaled, and the mutex re-acquired before returning from this service.

Tasks pend on condition variables by priority order.

Parameters:

cond The descriptor address of the affected condition variable.

mutex The descriptor address of the mutex protecting the condition variable.

timeout The absolute date specifying a time limit to wait for the condition variable to be signaled (see note). Passing TM_INFINITE causes the caller to block indefinitely until the condition variable is signaled.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *mutex* is not a mutex descriptor, or *cond* is not a condition variable descriptor.
- -EIDRM is returned if *mutex* or *cond* is a deleted object descriptor, including if the deletion occurred while the caller was sleeping on the variable.
- -ETIMEDOUT is returned if the absolute *timeout* date is reached before the condition variable is signaled.

- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the condition variable has been signaled.
- -EWOULDBLOCK is returned if *timeout* equals TM_NONBLOCK.

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

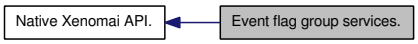
Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.5 Event flag group services.

Collaboration diagram for Event flag group services.:



Files

- file event.c

  This file is part of the Xenomai project.

Functions

- int rt_event_create (RT_EVENT *event, const char *name, unsigned long ivalue, int mode)

  Create an event group.

- int rt_event_delete (RT_EVENT *event)

  Delete an event group.

- int rt_event_signal (RT_EVENT *event, unsigned long mask)

  Post an event group.

- int rt_event_wait (RT_EVENT *event, unsigned long mask, unsigned long *mask_r, int mode, RTIME timeout)

  Pend on an event group.

- int rt_event_wait_until (RT_EVENT *event, unsigned long mask, unsigned long *mask_r, int mode, RTIME timeout)

  Pend on an event group (with absolute timeout date).

- int rt_event_clear (RT_EVENT *event, unsigned long mask, unsigned long *mask_r)

  Clear an event group.

- int rt_event_inquire (RT_EVENT *event, RT_EVENT_INFO *info)

  Inquire about an event group.

- int rt_event_bind (RT_EVENT *event, const char *name, RTIME timeout)

  Bind to an event flag group.

- static int rt_event_unbind (RT_EVENT *event)

  Unbind from an event flag group.

4.5.1 Detailed Description

An event flag group is a synchronization object represented by a long-word structure; every available bit in such word can be used to map a user-defined event flag. When a flag is set, the associated event is said to have occurred. Xenomai tasks and interrupt handlers can use event flags to signal the occurrence of events to other tasks; those tasks can either wait for the events to occur in a conjunctive manner (all awaited events must have occurred to wake up), or in a disjunctive way (at least one of the awaited events must have occurred to wake up).

4.5.2 Function Documentation

4.5.2.1 int rt_event_bind (RT_EVENT * *event*, const char * *name*, RTIME *timeout*)

Bind to an event flag group. This user-space only service retrieves the uniform descriptor of a given Xenomai event flag group identified by its symbolic name. If the event flag group does not exist on entry, this service blocks the caller until a event flag group of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the event flag group to bind to.

event The address of an event flag group descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *event* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.5.2.2 int rt_event_clear (RT_EVENT * *event*, unsigned long *mask*, unsigned long * *mask_r*)

Clear an event group. Clears a set of flags from an event mask.

Parameters:

event The descriptor address of the affected event.

mask The set of events to be cleared.

mask_r If non-NULL, *mask_r* is the address of a memory location which will be written upon success with the previous value of the event group before the flags are cleared.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *event* is not an event group descriptor.
- -EIDRM is returned if *event* is a deleted event group descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.5.2.3 int rt_event_create (RT_EVENT * *event*, const char * *name*, unsigned long *ivalue*, int *mode*)

Create an event group. Event groups provide for task synchronization by allowing a set of flags (or "events") to be waited for and posted atomically. An event group contains a mask of received events; any set of bits from the event mask can be pended or posted in a single operation.

Tasks can wait for a conjunctive (AND) or disjunctive (OR) set of events to occur. A task pending on an event group in conjunctive mode is woken up as soon as all awaited events are set in the event mask. A task pending on an event group in disjunctive mode is woken up as soon as any awaited event is set in the event mask.

Parameters:

event The address of an event group descriptor Xenomai will use to store the event-related data. This descriptor must always be valid while the group is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the group. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created event group.

ivalue The initial value of the group's event mask.

mode The event group creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new group:

- EV_FIFO makes tasks pend in FIFO order on the event group.
- EV_PRIO makes tasks pend in priority order on the event group.

Returns:

0 is returned upon success. Otherwise:

- -EEXIST is returned if the *name* is already in use by some registered object.
- -EPERM is returned if this service was called from an asynchronous context.
- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the event group.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

References rt_event_delete().

4.5.2.4 int rt_event_delete (RT_EVENT * *event*)

Delete an event group. Destroy an event group and release all the tasks currently pending on it. An event group exists in the system since rt_event_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

event The descriptor address of the affected event group.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *event* is not a event group descriptor.
- -EIDRM is returned if *event* is a deleted event group descriptor.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Referenced by rt_event_create().

4.5.2.5 int rt_event_inquire (RT_EVENT * *event*, RT_EVENT_INFO * *info*)

Inquire about an event group. Return various information about the status of a specified event group.

Parameters:

event The descriptor address of the inquired event group.

info The address of a structure the event group information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *event* is not a event group descriptor.
- -EIDRM is returned if *event* is a deleted event group descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.5.2.6 int rt_event_signal (RT_EVENT * *event*, unsigned long *mask*)

Post an event group. Post a set of bits to the event mask. All tasks having their wait request fulfilled by the posted events are resumed.

Parameters:

event The descriptor address of the affected event.

mask The set of events to be posted.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *event* is not an event group descriptor.
- -EIDRM is returned if *event* is a deleted event group descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.5.2.7 int rt_event_unbind (RT_EVENT * *event*) [inline, static]

Unbind from an event flag group. This user-space only service unbinds the calling task from the event flag group object previously retrieved by a call to rt_event_bind().

Parameters:

event The address of an event flag group descriptor to unbind from.

Returns:

0 is always returned.

This service can be called from:

- User-space task.

Rescheduling: never.

4.5.2.8 int rt_event_wait (RT_EVENT * *event*, unsigned long *mask*, unsigned long * *mask_r*, int *mode*, RTIME *timeout*)

Pend on an event group. Waits for one or more events on the specified event group, either in conjunctive or disjunctive mode.

If the specified set of bits is not set, the calling task is blocked. The task is not resumed until the request is fulfilled. The event bits are NOT cleared from the event group when a request is satisfied; rt_event_wait() will return immediately with success for the same event mask until rt_event_clear() is called to clear those bits.

Parameters:

event The descriptor address of the affected event group.

mask The set of bits to wait for. Passing zero causes this service to return immediately with a success value; the current value of the event mask is also copied to *mask_r*.

mask_r The value of the event mask at the time the task was readied.

mode The pend mode. The following flags can be OR'ed into this bitmask, each of them affecting the operation:

- EV_ANY makes the task pend in disjunctive mode (i.e. OR); this means that the request is fulfilled when at least one bit set into *mask* is set in the current event mask.

- EV_ALL makes the task pend in conjunctive mode (i.e. AND); this means that the request is fulfilled when at all bits set into *mask* are set in the current event mask.

Parameters:

timeout The number of clock ticks to wait for fulfilling the request (see note). Passing TM_-INFINITE causes the caller to block indefinitely until the request is fulfilled. Passing TM_NONBLOCK causes the service to return immediately without waiting if the request cannot be satisfied immediately.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *event* is not a event group descriptor.

- -EIDRM is returned if *event* is a deleted event group descriptor, including if the deletion occurred while the caller was sleeping on it before the request has been satisfied.

- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the current event mask value does not satisfy the request.

- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the request has been satisfied.

- -ETIMEDOUT is returned if the request has not been satisfied within the specified amount of time.

- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code or Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.5.2.9 int rt_event_wait_until (RT_EVENT * *event*, unsigned long *mask*, unsigned long * *mask_r*, int *mode*, RTIME *timeout*)

Pend on an event group (with absolute timeout date). Waits for one or more events on the specified event group, either in conjunctive or disjunctive mode.

If the specified set of bits is not set, the calling task is blocked. The task is not resumed until the request is fulfilled. The event bits are NOT cleared from the event group when a request is satisfied; rt_event_wait() will return immediately with success for the same event mask until rt_event_clear() is called to clear those bits.

Parameters:

event The descriptor address of the affected event group.

mask The set of bits to wait for. Passing zero causes this service to return immediately with a success value; the current value of the event mask is also copied to *mask_r*.

mask_r The value of the event mask at the time the task was readied.

mode The pend mode. The following flags can be OR'ed into this bitmask, each of them affecting the operation:

- EV_ANY makes the task pend in disjunctive mode (i.e. OR); this means that the request is fulfilled when at least one bit set into *mask* is set in the current event mask.
- EV_ALL makes the task pend in conjunctive mode (i.e. AND); this means that the request is fulfilled when at all bits set into *mask* are set in the current event mask.

Parameters:

timeout The absolute date specifying a time limit to wait for fulfilling the request (see note). Passing TM_INFINITE causes the caller to block indefinitely until the request is fulfilled. Passing TM_NONBLOCK causes the service to return immediately without waiting if the request cannot be satisfied immediately.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *event* is not a event group descriptor.
- -EIDRM is returned if *event* is a deleted event group descriptor, including if the deletion occurred while the caller was sleeping on it before the request has been satisfied.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the current event mask value does not satisfy the request.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the request has been satisfied.
- -ETIMEDOUT is returned if the absolute *timeout* date is reached before the request is satisfied.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code or Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.6 Memory heap services.

Collaboration diagram for Memory heap services.:

Files

- file heap.c
  This file is part of the Xenomai project.

Functions

- int rt_heap_create (RT_HEAP *heap, const char *name, size_t heapsize, int mode)
  Create a memory heap or a shared memory segment.
- int rt_heap_delete (RT_HEAP *heap)
  Delete a real-time heap.
- int rt_heap_alloc (RT_HEAP *heap, size_t size, RTIME timeout, void **blockp)
  Allocate a block or return the single segment base.
- int rt_heap_free (RT_HEAP *heap, void *block)
  Free a block.
- int rt_heap_inquire (RT_HEAP *heap, RT_HEAP_INFO *info)
  Inquire about a heap.
- int rt_heap_bind (RT_HEAP *heap, const char *name, RTIME timeout)
  Bind to a mappable heap.
- int rt_heap_unbind (RT_HEAP *heap)
  Unbind from a mappable heap.

4.6.1 Detailed Description

Memory heaps are regions of memory used for dynamic memory allocation in a time-bounded fashion. Blocks of memory are allocated and freed in an arbitrary order and the pattern of allocation and size of blocks is not known until run time.

The implementation of the memory allocator follows the algorithm described in a USENIX 1988 paper called "Design of a General Purpose Memory Allocator for the 4.3BSD Unix Kernel" by Marshall K. McKusick and Michael J. Karels.

Xenomai memory heaps are built over the nucleus's heap objects, which in turn provide the needed support for sharing a memory area between kernel and user-space using direct memory mapping.

4.6.2 Function Documentation

4.6.2.1 int rt_heap_alloc (RT_HEAP * *heap*, size_t *size*, RTIME *timeout*, void ** *blockp*)

Allocate a block or return the single segment base. This service allocates a block from the heap's internal pool, or returns the address of the single memory segment in the caller's address space. Tasks may wait for some requested amount of memory to become available from local heaps.

Parameters:

heap The descriptor address of the heap to allocate a block from.

size The requested size in bytes of the block. If the heap is managed as a single-block area (H_SINGLE), this value can be either zero, or the same value given to rt_heap_create(). In that case, the same block covering the entire heap space will always be returned to all callers of this service.

timeout The number of clock ticks to wait for a block of sufficient size to be available from a local heap (see note). Passing TM_INFINITE causes the caller to block indefinitely until some block is eventually available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no block is available on entry. This parameter has no influence if the heap is managed as a single-block area since the entire heap space is always available.

blockp A pointer to a memory location which will be written upon success with the address of the allocated block, or the start address of the single memory segment. In the former case, the block should be freed using rt_heap_free().

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *heap* is not a heap descriptor, or *heap* is managed as a single-block area (i.e. H_SINGLE mode) and *size* is non-zero but does not match the original heap size passed to rt_heap_create().
- -EIDRM is returned if *heap* is a deleted heap descriptor.
- -ETIMEDOUT is returned if *timeout* is different from TM_NONBLOCK and no block is available within the specified amount of time.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no block is immediately available on entry.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before any block was available.
- -EPERM is returned if this service should block but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK, or the heap is managed as a single-block area.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation. Operations on single-block heaps never start the rescheduling procedure.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.6.2.2 int rt_heap_bind (RT_HEAP * *heap*, const char * *name*, RTIME *timeout*)

Bind to a mappable heap. This user-space only service retrieves the uniform descriptor of a given mappable Xenomai heap identified by its symbolic name. If the heap does not exist on entry, this service blocks the caller until a heap of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the heap to bind to.

heap The address of a heap descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *heap* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).
- -ENOENT is returned if the special file /dev/rtheap (character-mode, major 10, minor 254) is not available from the filesystem. This device is needed to map the shared heap memory into the caller's address space. udev-based systems should not need manual creation of such device entry. Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

Examples:

shared_mem.c.

4.6.2.3 int rt_heap_create (RT_HEAP * *heap*, const char * *name*, size_t *heapsize*, int *mode*)

Create a memory heap or a shared memory segment. Initializes a memory heap suitable for time-bounded allocation requests of dynamic memory. Memory heaps can be local to the kernel address space, or mapped to user-space.

In their simplest form, heaps are only accessible from kernel space, and are merely usable as regular memory allocators.

Heaps existing in kernel space can be mapped by user-space processes to their own address space provided H_MAPPABLE has been passed into the *mode* parameter.

By default, heaps support allocation of multiple blocks of memory in an arbitrary order. However, it is possible to ask for single-block management by passing the H_SINGLE flag into the *mode* parameter, in which case the entire memory space managed by the heap is made available as a unique block. In this mode, all allocation requests made through rt_heap_alloc() will then return the same block address, pointing at the beginning of the heap memory.

H_SHARED is a shorthand for creating shared memory segments transparently accessible from kernel and user-space contexts, which are basically single-block, mappable heaps. By proper use of a common *name*, all tasks can bind themselves to the same heap and thus share the same memory space, which start address should be subsequently retrieved by a call to rt_heap_alloc().

Parameters:

heap The address of a heap descriptor Xenomai will use to store the heap-related data. This descriptor must always be valid while the heap is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the heap. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created heap. Mappable heaps must be given a valid name.

heapsize The size (in bytes) of the block pool which is going to be pre-allocated to the heap. Memory blocks will be claimed and released to this pool. The block pool is not extensible, so this value must be compatible with the highest memory pressure that could be expected. A minimum of 2 * PAGE_SIZE will be enforced for mappable heaps, 2 * XNHEAP_PAGE_SIZE otherwise.

mode The heap creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new heap:

- H_FIFO makes tasks pend in FIFO order on the heap when waiting for available blocks.
- H_PRIO makes tasks pend in priority order on the heap when waiting for available blocks.
- H_MAPPABLE causes the heap to be sharable between kernel and user-space contexts. Otherwise, the new heap is only available for kernel-based usage. This flag is implicitely set when the caller is running in user-space. This feature requires the real-time support in user-space to be configured in (CONFIG_XENO_OPT_PERVASIVE).
- H_SINGLE causes the entire heap space to be managed as a single memory block.
- H_SHARED is a shorthand for H_MAPPABLE|H_SINGLE, creating a global shared memory segment accessible from both the kernel and user-space contexts.
- H_DMA causes the block pool associated to the heap to be allocated in physically contiguous memory, suitable for DMA operations with I/O devices. The physical address of the heap can be obtained by a call to rt_heap_inquire().
- H_NONCACHED causes the heap not to be cached. This is necessary on platforms such as ARM to share a heap between kernel and user-space. Note that this flag is not compatible with the H_DMA flag.

Returns:

0 is returned upon success. Otherwise:

- -EEXIST is returned if the *name* is already in use by some registered object.
- -EINVAL is returned if *heapsize* is null, greater than the system limit, or *name* is null or empty for a mappable heap.
- -ENOMEM is returned if not enough system memory is available to create or register the heap. Additionally, and if H_MAPPABLE has been passed in *mode*, errors while mapping the block pool in the caller's address space might beget this return code too.
- -EPERM is returned if this service was called from an invalid context.
- -ENOSYS is returned if *mode* specifies H_MAPPABLE, but the real-time support in user-space is unavailable.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task (switches to secondary mode)

Rescheduling: possible.

References rt_heap_delete().

4.6.2.4 int rt_heap_delete (RT_HEAP * *heap*)

Delete a real-time heap. Destroy a heap and release all the tasks currently pending on it. A heap exists in the system since rt_heap_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

heap The descriptor address of the affected heap.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *heap* is not a heap descriptor.
- -EIDRM is returned if *heap* is a deleted heap descriptor.
- -EPERM is returned if this service was called from an invalid context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task (switches to secondary mode).

Rescheduling: possible.

Referenced by rt_heap_create().

4.6.2.5 int rt_heap_free (RT_HEAP * *heap*, void * *block*)

Free a block. This service releases a block to the heap's internal pool. If some task is currently waiting for a block so that it's pending request could be satisfied as a result of the release, it is immediately resumed.

If the heap is defined as a single-block area (i.e. H_SINGLE mode), this service leads to a null-effect and always returns successfully.

Parameters:

heap The address of the heap descriptor to which the block *block* belong.

block The address of the block to free.

Returns:

0 is returned upon success, or -EINVAL if *block* is not a valid block previously allocated by the rt_heap_alloc() service.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.6.2.6 int rt_heap_inquire (RT_HEAP * *heap*, RT_HEAP_INFO * *info*)

Inquire about a heap. Return various information about the status of a given heap.

Parameters:

heap The descriptor address of the inquired heap.

info The address of a structure the heap information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *heap* is not a message queue descriptor.
- -EIDRM is returned if *heap* is a deleted queue descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.6.2.7 int rt_heap_unbind (RT_HEAP * *heap*)

Unbind from a mappable heap. This user-space only service unbinds the calling task from the heap object previously retrieved by a call to rt_heap_bind().

Unbinding from a heap when it is no longer needed is especially important in order to properly release the mapping resources used to attach the heap memory to the caller's address space.

Parameters:

heap The address of a heap descriptor to unbind from.

Returns:

0 is always returned.

This service can be called from:

- User-space task.

Rescheduling: never.

Examples:

shared_mem.c.

4.7 Interrupt management services.

Collaboration diagram for Interrupt management services.:

Files

- file intr.c

  This file is part of the Xenomai project.

Functions

- int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq, rt_isr_t isr, rt_iack_t iack, int mode)

  Create an interrupt object from kernel space.

- int rt_intr_delete (RT_INTR *intr)

  Delete an interrupt object.

- int rt_intr_enable (RT_INTR *intr)

  Enable an interrupt object.

- int rt_intr_disable (RT_INTR *intr)

  Disable an interrupt object.

- int rt_intr_inquire (RT_INTR *intr, RT_INTR_INFO *info)

  Inquire about an interrupt object.

- int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq, int mode)

  Create an interrupt object from user-space.

- int rt_intr_wait (RT_INTR *intr, RTIME timeout)

  Wait for the next interrupt.

- int rt_intr_bind (RT_INTR *intr, const char *name, RTIME timeout)

  Bind to an interrupt object.

- static int rt_intr_unbind (RT_INTR *intr)

  Unbind from an interrupt object.

4.7.1 Function Documentation

4.7.1.1 int rt_intr_bind (RT_INTR * *intr*, const char * *name*, RTIME *timeout*)

Bind to an interrupt object. This user-space only service retrieves the uniform descriptor of a given Xenomai interrupt object identified by its IRQ number. If the object does not exist on entry, this service blocks the caller until an interrupt object of the given number is created. An interrupt is registered whenever a kernel-space task invokes the rt_intr_create() service successfully for the given IRQ line.

Parameters:

intr The address of an interrupt object descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

name An ASCII string standing for the symbolic name of the interrupt object to search for.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *intr* is referencing invalid memory.
- -EINVAL is returned if *irq* is invalid.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.7.1.2 int rt_intr_create (RT_INTR * *intr*, const char * *name*, unsigned *irq*, int *mode*)

Create an interrupt object from user-space. Initializes and associates an interrupt object with an IRQ line from a user-space application. In this mode, the basic principle is to define some interrupt server task which routinely waits for the next incoming IRQ event through the rt_intr_wait() syscall.

When an interrupt occurs on the given *irq* line, any task pending on the interrupt object through rt_intr_wait() is imediately awaken in order to deal with the hardware event. The interrupt service code may then call any Xenomai service available from user-space.

Parameters:

intr The address of a interrupt object descriptor Xenomai will use to store the object-specific data. This descriptor must always be valid while the object is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the interrupt object. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created interrupt objects.

irq The hardware interrupt channel associated with the interrupt object. This value is architecture-dependent.

mode The interrupt object creation mode. The following flags can be OR'ed into this bitmask:

- I_NOAUTOENA asks Xenomai not to re-enable the IRQ line before awakening the interrupt server task. This flag is functionally equivalent as always returning RT_INTR_NOENABLE from a kernel space interrupt handler.

- I_PROPAGATE asks Xenomai to propagate the IRQ down the pipeline; in other words, the interrupt occurrence is chained to Linux after it has been processed by the Xenomai task. This flag is functionally equivalent as always returning RT_INTR_PROPAGATE from a kernel space interrupt handler.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the interrupt object.

- -EBUSY is returned if the interrupt line is already in use by another interrupt object. Only a single interrupt object can be associated to any given interrupt line using rt_intr_create() at any time, regardless of the caller's execution space (kernel or user).

Environments:

This service can be called from:

- User-space task

Rescheduling: possible.

Note:

The interrupt source associated to the interrupt descriptor remains masked upon creation. rt_intr_enable() should be called for the new interrupt object to unmask it.

Examples:

user_irq.c.

4.7.1.3 int rt_intr_create (RT_INTR * *intr*, const char * *name*, unsigned *irq*, rt_isr_t *isr*, rt_iack_t *iack*, int *mode*)

Create an interrupt object from kernel space. Initializes and associates an interrupt object with an IRQ line. In kernel space, interrupts are immediately notified to a user-defined handler or ISR (interrupt service routine).

When an interrupt occurs on the given *irq* line, the ISR is fired in order to deal with the hardware event. The interrupt service code may call any non-suspensive Xenomai service.

Upon receipt of an IRQ, the ISR is immediately called on behalf of the interrupted stack context, the rescheduling procedure is locked, and the interrupt source is masked at hardware level. The status value returned by the ISR is then checked for the following values:

- RT_INTR_HANDLED indicates that the interrupt request has been fulfilled by the ISR.
- RT_INTR_NONE indicates the opposite to RT_INTR_HANDLED. The ISR must always return this value when it determines that the interrupt request has not been issued by the dedicated hardware device.

In addition, one of the following bits may be set by the ISR :

NOTE: use these bits with care and only when you do understand their effect on the system. The ISR is not encouraged to use these bits in case it shares the IRQ line with other ISRs in the real-time domain.

- RT_INTR_NOENABLE asks Xenomai not to re-enable the IRQ line upon return of the interrupt service routine.
- RT_INTR_PROPAGATE tells Xenomai to require the real-time control layer to forward the IRQ. For instance, this would cause the Adeos control layer to propagate the interrupt down the interrupt pipeline to other Adeos domains, such as Linux. This is the regular way to share interrupts between Xenomai and the Linux kernel. In effect, RT_INTR_-PROPAGATE implies RT_INTR_NOENABLE since it would make no sense to re-enable the interrupt channel before the next domain down the pipeline has had a chance to process the propagated interrupt.

A count of interrupt receipts is tracked into the interrupt descriptor, and reset to zero each time the interrupt object is attached. Since this count could wrap around, it should be used as an indication of interrupt activity only.

Parameters:

intr The address of a interrupt object descriptor Xenomai will use to store the object-specific data. This descriptor must always be valid while the object is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the interrupt object. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created interrupt objects.

irq The hardware interrupt channel associated with the interrupt object. This value is architecture-dependent.

isr The address of a valid interrupt service routine in kernel space. This handler will be called each time the corresponding IRQ is delivered on behalf of an interrupt context. A pointer to an internal information is passed to the routine which can use it to retrieve the descriptor address of the associated interrupt object through the I_DESC() macro.

iack The address of an optional interrupt acknowledge routine, aimed at replacing the default one. Only very specific situations actually require to override the default setting for this parameter, like having to acknowledge non-standard PIC hardware. *iack* should return a non-zero value to indicate that the interrupt has been properly acknowledged. If *iack* is NULL, the default routine will be used instead.

mode The interrupt object creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new interrupt object:

- I_SHARED enables IRQ-sharing with other interrupt objects.
- I_EDGE is an additional flag need to be set together with I_SHARED to enable IRQ-sharing of edge-triggered interrupts.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the interrupt object.
- -EBUSY is returned if the interrupt line is already in use by another interrupt object. Only a single interrupt object can be associated to any given interrupt line using rt_intr_create() at any time.
- -EEXIST is returned if *irq* is already associated to an existing interrupt object.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task (note that in user-space the interface is different, see rt_intr_create())

Rescheduling: possible.

Note:

The interrupt source associated to the interrupt descriptor remains masked upon creation. rt_intr_enable() should be called for the new interrupt object to unmask it.

References rt_intr_delete().

4.7.1.4 int rt_intr_delete (RT_INTR * *intr*)

Delete an interrupt object. Destroys an interrupt object. An interrupt exists in the system since rt_intr_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Any user-space task which might be currently pending on the interrupt object through the rt_intr_wait() service will be awaken as a result of the deletion, and return with the -EIDRM status.

Parameters:

intr The descriptor address of the affected interrupt object.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *intr* is not a interrupt object descriptor.
- -EIDRM is returned if *intr* is a deleted interrupt object descriptor.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Referenced by rt_intr_create().

4.7.1.5 int rt_intr_disable (RT_INTR * *intr*)

Disable an interrupt object. Disables the hardware interrupt line associated with an interrupt object. This operation invalidates further interrupt requests from the given source until the IRQ line is re-enabled anew through rt_intr_enable().

Parameters:

intr The descriptor address of the interrupt object to enable.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *intr* is not a interrupt object descriptor.
- -EIDRM is returned if *intr* is a deleted interrupt object descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: never.

4.7.1.6 int rt_intr_enable (RT_INTR * *intr*)

Enable an interrupt object. Enables the hardware interrupt line associated with an interrupt object. Over Adeos-based systems which mask and acknowledge IRQs upon receipt, this operation is necessary to revalidate the interrupt channel so that more interrupts from the same source can be notified.

Parameters:

intr The descriptor address of the interrupt object to enable.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *intr* is not a interrupt object descriptor.
- -EIDRM is returned if *intr* is a deleted interrupt object descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: never.

4.7.1.7 int rt_intr_inquire (RT_INTR * *intr*, RT_INTR_INFO * *info*)

Inquire about an interrupt object. Return various information about the status of a given interrupt object.

Parameters:

intr The descriptor address of the inquired interrupt object.

info The address of a structure the interrupt object information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *intr* is not a interrupt object descriptor.
- -EIDRM is returned if *intr* is a deleted interrupt object descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.7.1.8 int rt_intr_unbind (RT_INTR * *intr*) [inline, static]

Unbind from an interrupt object. This user-space only service unbinds the calling task from the interrupt object previously retrieved by a call to rt_intr_bind().

Parameters:

intr The address of a interrupt object descriptor to unbind from.

Returns:

0 is always returned.

This service can be called from:

- User-space task.

Rescheduling: never.

4.7.1.9 int rt_intr_wait (RT_INTR * *intr*, RTIME *timeout*)

Wait for the next interrupt. This user-space only call allows the current task to suspend execution until the associated interrupt event triggers. The priority of the current task is raised above all other Xenomai tasks - except those also undergoing an interrupt or alarm wait (see rt_alarm_wait()) - so that it would preempt any of them under normal circumstances (i.e. no scheduler lock).

Interrupt receipts are logged if they cannot be delivered immediately to some interrupt server task, so that a call to rt_intr_wait() might return immediately if an IRQ is already pending on entry of the service.

Parameters:

intr The descriptor address of the awaited interrupt.

timeout The number of clock ticks to wait for an interrupt to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until an interrupt triggers. Passing TM_NONBLOCK is invalid.

Returns:

A positive value is returned upon success, representing the number of pending interrupts to process. Otherwise:

- -ETIMEDOUT is returned if no interrupt occurred within the specified amount of time.
- -EINVAL is returned if *intr* is not an interrupt object descriptor, or *timeout* is equal to TM_NONBLOCK.
- -EIDRM is returned if *intr* is a deleted interrupt object descriptor, including if the deletion occurred while the caller was waiting for its next interrupt.
- -EINTR is returned if rt_task_unblock() has been called for the current task before the next interrupt occurrence.

Environments:

This service can be called from:

- User-space task

Rescheduling: always, unless an interrupt is already pending on entry.

Note:

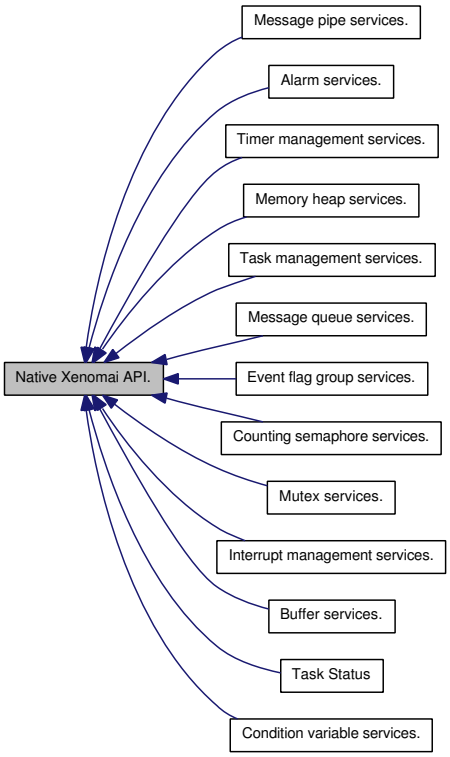
The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

Examples:

user_irq.c.

4.8 Native Xenomai API.

Collaboration diagram for Native Xenomai API.:



Modules

- Task Status
  Defines used to specify task state and/or mode.

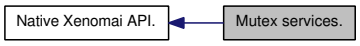
- Alarm services.
- Buffer services.
- Condition variable services.
- Event flag group services.
- Memory heap services.
- Interrupt management services.
- Mutex services.
- Message pipe services.
- Message queue services.
- Counting semaphore services.
- Task management services.
- Timer management services.

4.8.1 Detailed Description

The native Xenomai programming interface available to real-time applications. This API is built over the abstract RTOS core implemented by the Xenomai nucleus.

4.9 Mutex services.

Collaboration diagram for Mutex services.:



Files

- file mutex.c

  This file is part of the Xenomai project.

Functions

- int rt_mutex_create (RT_MUTEX *mutex, const char *name)

  Create a mutex.

- int rt_mutex_delete (RT_MUTEX *mutex)

  Delete a mutex.

- int rt_mutex_acquire (RT_MUTEX *mutex, RTIME timeout)

  Acquire a mutex.

- int rt_mutex_acquire_until (RT_MUTEX *mutex, RTIME timeout)

  Acquire a mutex (with absolute timeout date).

- int rt_mutex_release (RT_MUTEX *mutex)

  Unlock mutex.

- int rt_mutex_inquire (RT_MUTEX *mutex, RT_MUTEX_INFO *info)

  Inquire about a mutex.

- int rt_mutex_bind (RT_MUTEX *mutex, const char *name, RTIME timeout)

  Bind to a mutex.

- static int rt_mutex_unbind (RT_MUTEX *mutex)

  Unbind from a mutex.

4.9.1 Detailed Description

Mutex services.

A mutex is a MUTual EXclusion object, 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 task), and locked (owned by one task). A mutex can never be owned by two different tasks simultaneously. A task attempting to

lock a mutex that is already locked by another task is blocked until the latter unlocks the mutex first.

Xenomai mutex services enforce a priority inheritance protocol in order to solve priority inversions.

4.9.2 Function Documentation

4.9.2.1 int rt_mutex_acquire (RT_MUTEX * *mutex*, RTIME *timeout*)

Acquire a mutex. Attempt to lock a mutex. The calling task is blocked until the mutex is available, in which case it is locked again before this service returns. Mutexes have an ownership property, which means that their current owner is tracked. Xenomai mutexes are implicitely recursive and implement the priority inheritance protocol.

Since a nested locking count is maintained for the current owner, rt_mutex_acquire{_until}() and rt_mutex_release() must be used in pairs.

Tasks pend on mutexes by priority order.

Parameters:

mutex The descriptor address of the mutex to acquire.

timeout The number of clock ticks to wait for the mutex to be available to the calling task (see note). Passing TM_INFINITE causes the caller to block indefinitely until the mutex is available. Passing TM_NONBLOCK causes the service to return immediately without waiting if the mutex is still locked by another task.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *mutex* is not a mutex descriptor.
- -EIDRM is returned if *mutex* is a deleted mutex descriptor, including if the deletion occurred while the caller was sleeping on it.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the mutex is not immediately available.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the mutex has become available.
- -ETIMEDOUT is returned if the mutex cannot be made available to the calling task within the specified amount of time.
- -EPERM is returned if this service was called from a context which cannot be given the ownership of the mutex (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel-based task

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation. If the caller is blocked, the current owner's priority might be temporarily raised as a consequence of the priority inheritance protocol.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.9.2.2 int rt_mutex_acquire_until (RT_MUTEX * *mutex*, RTIME *timeout*)

Acquire a mutex (with absolute timeout date). Attempt to lock a mutex. The calling task is blocked until the mutex is available, in which case it is locked again before this service returns. Mutexes have an ownership property, which means that their current owner is tracked. Xenomai mutexes are implicitely recursive and implement the priority inheritance protocol.

Since a nested locking count is maintained for the current owner, rt_mutex_acquire{_until}() and rt_mutex_release() must be used in pairs.

Tasks pend on mutexes by priority order.

Parameters:

mutex The descriptor address of the mutex to acquire.

timeout The absolute date specifying a time limit to wait for the mutex to be available to the calling task (see note).

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *mutex* is not a mutex descriptor.
- -EIDRM is returned if *mutex* is a deleted mutex descriptor, including if the deletion occurred while the caller was sleeping on it.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the mutex is not immediately available.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the mutex has become available.
- -ETIMEDOUT is returned if the mutex cannot be made available to the calling task until the absolute timeout date is reached.
- -EPERM is returned if this service was called from a context which cannot be given the ownership of the mutex (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation. If the caller is blocked, the current owner's priority might be temporarily raised as a consequence of the priority inheritance protocol.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.9.2.3 int rt_mutex_bind (RT_MUTEX * *mutex*, const char * *name*, RTIME *timeout*)

Bind to a mutex. This user-space only service retrieves the uniform descriptor of a given Xenomai mutex identified by its symbolic name. If the mutex does not exist on entry, this service blocks the caller until a mutex of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the mutex to bind to.

mutex The address of a mutex descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *mutex* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.9.2.4 int rt_mutex_create (RT_MUTEX * *mutex*, const char * *name*)

Create a mutex. Create a mutual exclusion object that allows multiple tasks to synchronize access to a shared resource. A mutex is left in an unlocked state after creation.

Parameters:

mutex The address of a mutex descriptor Xenomai will use to store the mutex-related data. This descriptor must always be valid while the mutex is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the mutex. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created mutex.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the mutex.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

4.9.2.5 int rt_mutex_delete (RT_MUTEX * *mutex*)

Delete a mutex. Destroy a mutex and release all the tasks currently pending on it. A mutex exists in the system since rt_mutex_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

mutex The descriptor address of the affected mutex.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *mutex* is not a mutex descriptor.
- -EIDRM is returned if *mutex* is a deleted mutex descriptor.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

4.9.2.6 int rt_mutex_inquire (RT_MUTEX * *mutex*, RT_MUTEX_INFO * *info*)

Inquire about a mutex. Return various information about the status of a given mutex.

Parameters:

mutex The descriptor address of the inquired mutex.

info The address of a structure the mutex information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *mutex* is not a mutex descriptor.
- -EIDRM is returned if *mutex* is a deleted mutex descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

References rt_mutex_info::locked, rt_mutex_info::name, rt_mutex_info::nwaiters, and rt_mutex_-info::owner.

4.9.2.7 int rt_mutex_release (RT_MUTEX * *mutex*)

Unlock mutex. Release a mutex. If the mutex is pended, the first waiting task (by priority order) is immediately unblocked and transfered the ownership of the mutex; otherwise, the mutex is left in an unlocked state.

Parameters:

mutex The descriptor address of the released mutex.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *mutex* is not a mutex descriptor.
- -EIDRM is returned if *mutex* is a deleted mutex descriptor.
- -EPERM is returned if *mutex* is not owned by the current task, or more generally if this service was called from a context which cannot own any mutex (e.g. interrupt, or non-realtime context).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: possible.

4.9.2.8 int rt_mutex_unbind (RT_MUTEX * *mutex*) [inline, static]

Unbind from a mutex. This user-space only service unbinds the calling task from the mutex object previously retrieved by a call to rt_mutex_bind().

Parameters:

mutex The address of a mutex descriptor to unbind from.

Returns:

0 is always returned.

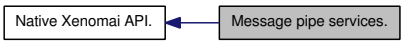
This service can be called from:

- User-space task.

Rescheduling: never.

4.10 Message pipe services.

Collaboration diagram for Message pipe services.:



Files

- file pipe.c

  This file is part of the Xenomai project.

Functions

- int rt_pipe_create (RT_PIPE *pipe, const char *name, int minor, size_t poolsize)

  Create a message pipe.

- int rt_pipe_delete (RT_PIPE *pipe)

  Delete a message pipe.

- ssize_t rt_pipe_receive (RT_PIPE *pipe, RT_PIPE_MSG **msgp, RTIME timeout)

  Receive a message from a pipe.

- ssize_t rt_pipe_read (RT_PIPE *pipe, void *buf, size_t size, RTIME timeout)

  Read a message from a pipe.

- ssize_t rt_pipe_send (RT_PIPE *pipe, RT_PIPE_MSG *msg, size_t size, int mode)

  Send a message through a pipe.

- ssize_t rt_pipe_write (RT_PIPE *pipe, const void *buf, size_t size, int mode)

  Write a message to a pipe.

- ssize_t rt_pipe_stream (RT_PIPE *pipe, const void *buf, size_t size)

  Stream bytes to a pipe.

- RT_PIPE_MSG * rt_pipe_alloc (RT_PIPE *pipe, size_t size)

  Allocate a message pipe buffer.

- int rt_pipe_free (RT_PIPE *pipe, RT_PIPE_MSG *msg)

  Free a message pipe buffer.

- int rt_pipe_flush (RT_PIPE *pipe, int mode)

  Flush the i/o queues associated with the kernel endpoint of a message pipe.

- int rt_pipe_monitor (RT_PIPE *pipe, int(*fn)(RT_PIPE *pipe, int event, long arg))

  Monitor a message pipe asynchronously.

4.10.1 Detailed Description

Message pipe services.

A message pipe is a two-way communication channel between Xenomai tasks and standard Linux processes using regular file I/O operations on a pseudo-device. Pipes can be operated in a message-oriented fashion so that message boundaries are preserved, and also in byte streaming mode from real-time to standard Linux processes for optimal throughput.

Xenomai tasks open their side of the pipe using the rt_pipe_create() service; standard Linux processes do the same by opening one of the /dev/rtpN special devices, where N is the minor number agreed upon between both ends of each pipe. Additionally, named pipes are available through the registry support, which automatically creates a symbolic link from entries under /proc/xenomai/registry/native/pipes/ to the corresponding special device file.

4.10.2 Function Documentation

4.10.2.1 RT_PIPE_MSG* rt_pipe_alloc (RT_PIPE * *pipe*, size_t *size*)

Allocate a message pipe buffer. This service allocates a message buffer from the pipe's heap which can be subsequently filled by the caller then passed to rt_pipe_send() for sending. The beginning of the available data area of *size* contiguous bytes is accessible from P_MSGPTR(msg).

Parameters:

pipe The descriptor address of the affected pipe.

size The requested size in bytes of the buffer. This value should represent the size of the payload data.

Returns:

The address of the allocated message buffer upon success, or NULL if the allocation fails.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task

Rescheduling: never.

Referenced by rt_pipe_write().

4.10.2.2 int rt_pipe_create (RT_PIPE * *pipe*, const char * *name*, int *minor*, size_t *poolsize*)

Create a message pipe. This service opens a bi-directional communication channel allowing data exchange between Xenomai tasks and standard Linux processes. Pipes natively preserve message boundaries, but can also be used in byte stream mode from Xenomai tasks to standard Linux processes.

rt_pipe_create() always returns immediately, even if no Linux process has opened the associated special device file yet. On the contrary, the non real-time side could block upon attempt to open the special device file until rt_pipe_create() is issued on the same pipe from a Xenomai task, unless O_NONBLOCK has been specified to the open(2) system call.

Parameters:

pipe The address of a pipe descriptor Xenomai will use to store the pipe-related data. This descriptor must always be valid while the pipe is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the message pipe. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created pipe.

Named pipes are supported through the use of the registry. When the registry support is enabled, passing a valid *name* parameter when creating a message pipe subsequently allows standard Linux processes to follow a symbolic link from /proc/xenomai/registry/pipes/*name* in order to reach the associated special device (i.e. /dev/rtp*), so that the specific *minor* information does not need to be known from those processes for opening the proper device file. In such a case, both sides of the pipe only need to agree upon a symbolic name to refer to the same data path, which is especially useful whenever the *minor* number is picked up dynamically using an adaptive algorithm, such as passing P_MINOR_AUTO as *minor* value.

Parameters:

minor The minor number of the device associated with the pipe. Passing P_MINOR_AUTO causes the minor number to be auto-allocated. In such a case, the *name* parameter must be valid so that user-space processes may subsequently follow the symbolic link that will be automatically created from /proc/xenomai/registry/pipes/*name* to the allocated pipe device entry (i.e. /dev/rtp*).

poolsize Specifies the size of a dedicated buffer pool for the pipe. Passing 0 means that all message allocations for this pipe are performed on the system heap.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the pipe, or if not enough memory could be obtained from the selected buffer pool for allocating the internal streaming buffer.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -ENODEV is returned if *minor* is different from P_MINOR_AUTO and is not a valid minor number for the pipe special device either (i.e. /dev/rtp*).
- -EBUSY is returned if *minor* is already open.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

References rt_pipe_delete().

4.10.2.3 int rt_pipe_delete (RT_PIPE * *pipe*)

Delete a message pipe. This service deletes a pipe previously created by rt_pipe_create(). Data pending for transmission to non real-time processes are lost.

Parameters:

pipe The descriptor address of the affected pipe.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *pipe* is not a pipe descriptor.
- -EIDRM is returned if *pipe* is a closed pipe descriptor.
- -ENODEV or -EBADF can be returned if *pipe* is scrambled.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Referenced by rt_pipe_create().

4.10.2.4 int rt_pipe_flush (RT_PIPE * *pipe*, int *mode*)

Flush the i/o queues associated with the kernel endpoint of a message pipe. This service flushes all data pending for consumption by the remote side in user-space for the given message pipe. Upon success, no data remains to be read from the remote side of the connection.

Parameters:

pipe The descriptor address of the pipe to flush.

mode A mask indicating which queues need to be flushed; the following flags may be combined in a single flush request:

- XNPIPE_OFLUSH causes the output queue to be flushed (i.e. unread data sent from the real-time endpoint in kernel-space to the non real-time endpoint in user-space will be discarded). This is equivalent to calling ioctl(pipefd, XNPIPEIOC_OFLUSH, 0) from user-space.

- XNPIPE_IFLUSH causes the input queue to be flushed (i.e. unread data sent from the non real-time endpoint in user-space to the real-time endpoint in kernel-space will be discarded). This is equivalent to calling ioctl(pipefd, XNPIPEIOC_IFLUSH, 0) from user-space.

Returns:

Zero is returned upon success. Otherwise:

- -EINVAL is returned if *pipe* is not a pipe descriptor.

- -EIDRM is returned if *pipe* is a closed pipe descriptor.

- -ENODEV or -EBADF are returned if *pipe* is scrambled.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task

Rescheduling: never.

4.10.2.5 int rt_pipe_free (RT_PIPE * *pipe*, RT_PIPE_MSG * *msg*)

Free a message pipe buffer. This service releases a message buffer returned by rt_pipe_receive() to the pipe's heap.

Parameters:

pipe The descriptor address of the affected pipe.

msg The address of the message buffer to free.

Returns:

0 is returned upon success, or -EINVAL if *msg* is not a valid message buffer previously allocated by the rt_pipe_alloc() service.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Interrupt service routine
- Kernel-based task

Rescheduling: never.

Referenced by rt_pipe_read(), and rt_pipe_write().

4.10.2.6 int rt_pipe_monitor (RT_PIPE * *pipe*, int(*)(RT_PIPE *pipe, int event, long arg) *fn*)

Monitor a message pipe asynchronously. This service registers a notifier callback that will be called upon specific events occurring on the channel. rt_pipe_monitor() is particularly useful to monitor a channel asynchronously while performing other tasks.

Parameters:

pipe The descriptor address of the pipe to monitor.

fn The notification handler. This user-provided routine will be passed the address of the message pipe descriptor receiving the event, the event code, and an optional argument. Four events are currently defined:

- P_EVENT_INPUT is sent when the user-space endpoint writes to the pipe, which means that some input is pending for the kernel-based endpoint. The argument is the size of the incoming message.
- P_EVENT_OUTPUT is sent when the user-space endpoint successfully reads a complete buffer from the pipe. The argument is the size of the outgoing message.
- P_EVENT_CLOSE is sent when the user-space endpoint is closed. The argument is always 0.
- P_EVENT_NOBUF is sent when no memory is available from the kernel pool to hold the message currently sent from the user-space endpoint. The argument is the size of the failed allocation. Upon return from the handler, the caller will block and retry until enough space is available from the pool; during that process, the handler might be called multiple times, each time a new attempt to get the required memory fails.

The P_EVENT_INPUT and P_EVENT_OUTPUT events are fired on behalf of a fully atomic context; therefore, care must be taken to keep their overhead low. In those cases, the Xenomai services that may be called from the handler are restricted to the set allowed to a real-time interrupt handler.

Returns:

Zero is returned upon success. Otherwise:

- -EINVAL is returned if *pipe* is not a pipe descriptor.
- -EIDRM is returned if *pipe* is a closed pipe descriptor.
- -ENODEV or -EBADF are returned if *pipe* is scrambled.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task

Rescheduling: never.

4.10.2.7 ssize_t rt_pipe_read (RT_PIPE * *pipe*, void * *buf*, size_t *size*, RTIME *timeout*)

Read a message from a pipe. This service retrieves the next message written to the associated special device in user-space. rt_pipe_read() always preserves message boundaries, which means that all data sent through the same write(2) operation to the special device will be gathered in a single message by this service. This services differs from rt_pipe_receive() in that it copies back the payload data to a user-defined memory area, instead of returning a pointer to the internal message buffer holding such data.

Unless otherwise specified, the caller is blocked for a given amount of time if no data is immediately available on entry.

Parameters:

pipe The descriptor address of the pipe to read from.

buf A pointer to a memory location which will be written upon success with the read message contents.

size The count of bytes from the received message to read up into *buf*. If *size* is lower than the actual message size, -ENOBUFS is returned since the incompletely received message would be lost. If *size* is zero, this call returns immediately with no other action.

timeout The number of clock ticks to wait for some message to arrive (see note). Passing TM_-INFINITE causes the caller to block indefinitely until some data is eventually available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no data is available on entry.

Returns:

The number of read bytes copied to the *buf* is returned upon success. Otherwise:

- 0 is returned if the peer closed the channel while rt_pipe_read() was reading from it. There is no way to distinguish this situation from an empty message return using rt_pipe_read(). One should rather call rt_pipe_receive() whenever this information is required.
- -EINVAL is returned if *pipe* is not a pipe descriptor.
- -EIDRM is returned if *pipe* is a closed pipe descriptor.
- -ENODEV or -EBADF are returned if *pipe* is scrambled.
- -ETIMEDOUT is returned if *timeout* is different from TM_NONBLOCK and no data is available within the specified amount of time.

- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no data is immediately available on entry.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before any data was available.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).
- -ENOBUFS is returned if *size* is not large enough to collect the message data.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

References rt_pipe_free(), and rt_pipe_receive().

4.10.2.8 ssize_t rt_pipe_receive (RT_PIPE * *pipe*, RT_PIPE_MSG ** *msgp*, RTIME *timeout*)

Receive a message from a pipe. This service retrieves the next message written to the associated special device in user-space. rt_pipe_receive() always preserves message boundaries, which means that all data sent through the same write(2) operation to the special device will be gathered in a single message by this service. This service differs from rt_pipe_read() in that it returns a pointer to the internal buffer holding the message, which improves performances by saving a data copy to a user-provided buffer, especially when large messages are involved.

Unless otherwise specified, the caller is blocked for a given amount of time if no data is immediately available on entry.

Parameters:

pipe The descriptor address of the pipe to receive from.

msgp A pointer to a memory location which will be written upon success with the address of the received message. Once consumed, the message space should be freed using rt_pipe_free(). The application code can retrieve the actual data and size carried by the message by respectively using the P_MSGPTR() and P_MSGSIZE() macros. *msgp is set to NULL and zero is returned to the caller, in case the peer closed the channel while rt_pipe_receive() was reading from it.

timeout The number of clock ticks to wait for some message to arrive (see note). Passing TM_-INFINITE causes the caller to block indefinitely until some data is eventually available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no data is available on entry.

Returns:

The number of read bytes available from the received message is returned upon success; this value will be equal to P_MSGSIZE(*msgp). Otherwise:

- 0 is returned and *msgp is set to NULL if the peer closed the channel while rt_pipe_receive() was reading from it. This is to be distinguished from an empty message return, where *msgp points to a valid - albeit empty - message block (i.e. P_MSGSIZE(*msgp) == 0).
- -EINVAL is returned if *pipe* is not a pipe descriptor.
- -ENODEV or -EBADF are returned if *pipe* is scrambled.
- -ETIMEDOUT is returned if *timeout* is different from TM_NONBLOCK and no data is available within the specified amount of time.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no data is immediately available on entry.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before any data was available.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

Referenced by rt_pipe_read().

4.10.2.9 ssize_t rt_pipe_send (RT_PIPE * *pipe*, RT_PIPE_MSG * *msg*, size_t *size*, int *mode*)

Send a message through a pipe. This service writes a complete message to be received from the associated special device. rt_pipe_send() always preserves message boundaries, which means that all data sent through a single call of this service will be gathered in a single read(2) operation from the special device. This service differs from rt_pipe_write() in that it accepts a canned message buffer, instead of a pointer to the raw data to be sent. This call is useful whenever the caller wants to prepare the message contents separately from its sending, which does not require to have all the data to be sent available at once but allows for incremental updates of the message, and also saves a message copy, since rt_pipe_send() deals internally with message buffers.

Parameters:

pipe The descriptor address of the pipe to send to.

msg The address of the message to be sent. The message space must have been allocated using the rt_pipe_alloc() service. Once passed to rt_pipe_send(), the memory pointed to by *msg* is no more under the control of the application code and thus should not be referenced by it anymore; deallocation of this memory will be automatically handled as needed. As a special exception, *msg* can be NULL and will not be dereferenced if *size* is zero.

size The size in bytes of the message (payload data only). Zero is a valid value, in which case the service returns immediately without sending any message. This parameter allows you to actually send less data than you reserved using the rt_pipe_alloc() service, which may be the case if you did not know how much space you needed at the time of allocation. In all other cases it may be more convenient to just pass P_MSGSIZE(msg).

mode A set of flags affecting the operation:

- P_URGENT causes the message to be prepended to the output queue, ensuring a LIFO ordering.
- P_NORMAL causes the message to be appended to the output queue, ensuring a FIFO ordering.

Returns:

Upon success, this service returns *size*. Upon error, one of the following error codes is returned:

- -EINVAL is returned if *pipe* is not a pipe descriptor.
- -EPIPE is returned if the associated special device is not yet open.
- -EIDRM is returned if *pipe* is a closed pipe descriptor.
- -ENODEV or -EBADF are returned if *pipe* is scrambled.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine

- Kernel-based task

Rescheduling: possible.

Referenced by rt_pipe_write().

4.10.2.10 ssize_t rt_pipe_stream (RT_PIPE * *pipe*, const void * *buf*, size_t *size*)

Stream bytes to a pipe. This service writes a sequence of bytes to be received from the associated special device. Unlike rt_pipe_send(), this service does not preserve message boundaries. Instead, an internal buffer is filled on the fly with the data, which will be consumed as soon as the receiver wakes up.

Data buffers sent by the rt_pipe_stream() service are always transmitted in FIFO order (i.e. P_-NORMAL mode).

Parameters:

pipe The descriptor address of the pipe to write to.

buf The address of the first data byte to send. The data will be copied to an internal buffer before transmission.

size The size in bytes of the buffer. Zero is a valid value, in which case the service returns immediately without buffering any data.

Returns:

The number of bytes sent upon success; this value may be lower than *size*, depending on the available space in the internal buffer. Otherwise:

- -EINVAL is returned if *pipe* is not a pipe descriptor.
- -EPIPE is returned if the associated special device is not yet open.
- -EIDRM is returned if *pipe* is a closed pipe descriptor.
- -ENODEV or -EBADF are returned if *pipe* is scrambled.
- -ENOSYS is returned if the byte streaming mode has been disabled at configuration time by nullifying the size of the pipe buffer (see CONFIG_XENO_OPT_NATIVE_PIPE_BUFSZ).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.10.2.11 ssize_t rt_pipe_write (RT_PIPE * *pipe*, const void * *buf*, size_t *size*, int *mode*)

Write a message to a pipe. This service writes a complete message to be received from the associated special device. rt_pipe_write() always preserves message boundaries, which means that all data sent through a single call of this service will be gathered in a single read(2) operation from the special device. This service differs from rt_pipe_send() in that it accepts a pointer to the raw data to be sent, instead of a canned message buffer. This call is useful whenever the caller does not need to prepare the message contents separately from its sending.

Parameters:

pipe The descriptor address of the pipe to write to.

buf The address of the first data byte to send. The data will be copied to an internal buffer before transmission.

size The size in bytes of the message (payload data only). Zero is a valid value, in which case the service returns immediately without sending any message.

mode A set of flags affecting the operation:

- P_URGENT causes the message to be prepended to the output queue, ensuring a LIFO ordering.
- P_NORMAL causes the message to be appended to the output queue, ensuring a FIFO ordering.

Returns:

Upon success, this service returns *size*. Upon error, one of the following error codes is returned:

- -EINVAL is returned if *pipe* is not a pipe descriptor.
- -EPIPE is returned if the associated special device is not yet open.
- -ENOMEM is returned if not enough buffer space is available to complete the operation.
- -EIDRM is returned if *pipe* is a closed pipe descriptor.
- -ENODEV or -EBADF are returned if *pipe* is scrambled.

Environments:

This service can be called from:

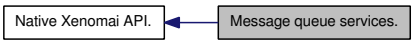
- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

References rt_pipe_alloc(), rt_pipe_free(), and rt_pipe_send().

4.11 Message queue services.

Collaboration diagram for Message queue services.:



Files

- file queue.c

  This file is part of the Xenomai project.

Functions

- int rt_queue_create (RT_QUEUE *q, const char *name, size_t poolsize, size_t qlimit, int mode)

  Create a message queue.

- int rt_queue_delete (RT_QUEUE *q)

  Delete a message queue.

- void * rt_queue_alloc (RT_QUEUE *q, size_t size)

  Allocate a message queue buffer.

- int rt_queue_free (RT_QUEUE *q, void *buf)

  Free a message queue buffer.

- int rt_queue_send (RT_QUEUE *q, void *mbuf, size_t size, int mode)

  Send a message to a queue.

- int rt_queue_write (RT_QUEUE *q, const void *buf, size_t size, int mode)

  Write a message to a queue.

- ssize_t rt_queue_receive (RT_QUEUE *q, void **bufp, RTIME timeout)

  Receive a message from a queue.

- ssize_t rt_queue_receive_until (RT_QUEUE *q, void **bufp, RTIME timeout)

  Receive a message from a queue (with absolute timeout date).

- ssize_t rt_queue_read (RT_QUEUE *q, void *buf, size_t size, RTIME timeout)

  Read a message from a queue.

- ssize_t rt_queue_read_until (RT_QUEUE *q, void *buf, size_t size, RTIME timeout)

  Read a message from a queue (with absolute timeout date).

- int rt_queue_flush (RT_QUEUE *q)

  Flush a message queue.

- int rt_queue_inquire (RT_QUEUE *q, RT_QUEUE_INFO *info)
  Inquire about a message queue.
- int rt_queue_bind (RT_QUEUE *q, const char *name, RTIME timeout)
  Bind to a shared message queue.
- int rt_queue_unbind (RT_QUEUE *q)
  Unbind from a shared message queue.

4.11.1 Detailed Description

Queue services.

Message queueing is a method by which real-time tasks can exchange or pass data through a Xenomai-managed queue of messages. Messages can vary in length and be assigned different types or usages. A message queue can be created by one task and used by multiple tasks that send and/or receive messages to the queue.

This implementation is based on a zero-copy scheme for message buffers. Message buffer pools are built over the nucleus's heap objects, which in turn provide the needed support for exchanging messages between kernel and user-space using direct memory mapping.

4.11.2 Function Documentation

4.11.2.1 void* rt_queue_alloc (RT_QUEUE * *q*, size_t *size*)

Allocate a message queue buffer. This service allocates a message buffer from the queue's internal pool which can be subsequently filled by the caller then passed to rt_queue_send() for sending.

Parameters:

q The descriptor address of the affected queue.

size The requested size in bytes of the buffer. Zero is an acceptable value, meaning that the message will not carry any payload data; the receiver will thus receive a zero-sized message.

Returns:

The address of the allocated message buffer upon success, or NULL if the allocation fails.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

Referenced by rt_queue_write().

4.11.2.2 int rt_queue_bind (RT_QUEUE * *q*, const char * *name*, RTIME *timeout*)

Bind to a shared message queue. This user-space only service retrieves the uniform descriptor of a given shared Xenomai message queue identified by its symbolic name. If the queue does not exist on entry, this service blocks the caller until a queue of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the queue to bind to.

q The address of a queue descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *q* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context). This error may also be returned whenever the call attempts to bind from a user-space application to a local queue defined from kernel space (i.e. Q_SHARED was not passed to rt_queue_create()).
- -ENOENT is returned if the special file /dev/rtheap (character-mode, major 10, minor 254) is not available from the filesystem. This device is needed to map the memory pool used by the shared queue into the caller's address space. udev-based systems should not need manual creation of such device entry.

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

Examples:

msg_queue.c.

4.11.2.3 int rt_queue_create (RT_QUEUE * *q*, const char * *name*, size_t *poolsize*, size_t *qlimit*, int *mode*)

Create a message queue. Create a message queue object that allows multiple tasks to exchange data through the use of variable-sized messages. A message queue is created empty. Message queues can be local to the kernel space, or shared between kernel and user-space.

This service needs the special character device /dev/rtheap (10,254) when called from user-space tasks.

Parameters:

q The address of a queue descriptor Xenomai will use to store the queue-related data. This descriptor must always be valid while the message queue is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the queue. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created queue. Shared queues must be given a valid name.

poolsize The size (in bytes) of the message buffer pool which is going to be pre-allocated to the queue. Message buffers will be claimed and released to this pool. The buffer pool memory is not extensible, so this value must be compatible with the highest message pressure that could be expected.

qlimit This parameter allows to limit the maximum number of messages which can be queued at any point in time. Sending to a full queue begets an error. The special value Q_UNLIMITED can be passed to specify an unlimited amount.

mode The queue creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new queue:

- Q_FIFO makes tasks pend in FIFO order on the queue for consuming messages.
- Q_PRIO makes tasks pend in priority order on the queue.
- Q_SHARED causes the queue to be sharable between kernel and user-space tasks. Otherwise, the new queue is only available for kernel-based usage. This flag is implicitely set when the caller is running in user-space. This feature requires the real-time support in user-space to be configured in (CONFIG_XENO_OPT_PERVASIVE).
- Q_DMA causes the buffer pool associated to the queue to be allocated in physically contiguous memory, suitable for DMA operations with I/O devices. A 128Kb limit exists for *poolsize* when this flag is passed.

Returns:

0 is returned upon success. Otherwise:

- -EEXIST is returned if the *name* is already in use by some registered object.

- -EINVAL is returned if *poolsize* is null, greater than the system limit, or *name* is null or empty for a shared queue.
- -ENOMEM is returned if not enough system memory is available to create or register the queue. Additionally, and if Q_SHARED has been passed in *mode*, errors while mapping the buffer pool in the caller's address space might beget this return code too.
- -EPERM is returned if this service was called from an invalid context.
- -ENOSYS is returned if *mode* specifies Q_SHARED, but the real-time support in user-space is unavailable.
- -ENOENT is returned if /dev/rtheap can't be opened.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task (switches to secondary mode)

Rescheduling: possible.

References rt_queue_delete().

4.11.2.4 int rt_queue_delete (RT_QUEUE * *q*)

Delete a message queue. Destroy a message queue and release all the tasks currently pending on it. A queue exists in the system since rt_queue_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

q The descriptor address of the affected queue.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task (switches to secondary mode).

Rescheduling: possible.

Referenced by rt_queue_create().

4.11.2.5 int rt_queue_flush (RT_QUEUE * *q*)

Flush a message queue. This service discards all unread messages from a message queue.

Parameters:

q The descriptor address of the affected queue.

Returns:

The number of messages flushed is returned upon success. Otherwise:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

References rt_queue_free().

4.11.2.6 int rt_queue_free (RT_QUEUE * *q*, void * *buf*)

Free a message queue buffer. This service releases a message buffer returned by rt_queue_receive() to the queue's internal pool.

Parameters:

q The descriptor address of the affected queue.

buf The address of the message buffer to free. Even zero-sized messages carrying no payload data must be freed, since they are assigned a valid memory space to store internal information.

Returns:

0 is returned upon success, or -EINVAL if *buf* is not a valid message buffer previously allocated by the rt_queue_alloc() service, or the caller did not get ownership of the message through a successful return from rt_queue_receive().

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

Referenced by rt_queue_flush().

4.11.2.7 int rt_queue_inquire (RT_QUEUE * *q*, RT_QUEUE_INFO * *info*)

Inquire about a message queue. Return various information about the status of a given queue.

Parameters:

q The descriptor address of the inquired queue.

info The address of a structure the queue information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.11.2.8 ssize_t rt_queue_read (RT_QUEUE * *q*, void * *buf*, size_t *size*, RTIME *timeout*)

Read a message from a queue. This service retrieves the next message available from the given queue. Unless otherwise specified, the caller is blocked for a given amount of time if no message is immediately available on entry. This services differs from rt_queue_receive() in that it copies back the payload data to a user-defined memory area, instead of returning a pointer to the message buffer holding such data.

Parameters:

q The descriptor address of the message queue to read from.

buf A pointer to a memory area which will be written upon success with the message contents. The internal message buffer conveying the data is automatically freed by this call.

size The length in bytes of the memory area pointed to by *buf*. Messages larger than *size* are truncated appropriately.

timeout The number of clock ticks to wait for a message to arrive (see note). Passing TM_INFINITE causes the caller to block indefinitely until some message is eventually available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no message is available on entry.

Returns:

The number of bytes available from the received message is returned upon success, which might be greater than the actual number of bytes copied to the destination buffer if the message has been truncated. Zero is a possible value corresponding to a zero-sized message passed to rt_queue_send() or rt_queue_write(). Otherwise:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.
- -ETIMEDOUT is returned if *timeout* is different from TM_NONBLOCK and no message is available within the specified amount of time.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no message is immediately available on entry.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before any data was available.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.11.2.9 ssize_t rt_queue_read_until (RT_QUEUE * *q*, void * *buf*, size_t *size*, RTIME *timeout*)

Read a message from a queue (with absolute timeout date). This service retrieves the next message available from the given queue. Unless otherwise specified, the caller is blocked for a given amount of time if no message is immediately available on entry. This services differs from rt_queue_receive() in that it copies back the payload data to a user-defined memory area, instead of returning a pointer to the message buffer holding such data.

Parameters:

q The descriptor address of the message queue to read from.

buf A pointer to a memory area which will be written upon success with the message contents. The internal message buffer conveying the data is automatically freed by this call.

size The length in bytes of the memory area pointed to by *buf*. Messages larger than *size* are truncated appropriately.

timeout The absolute date specifying a time limit to wait for a message to arrive (see note). Passing TM_INFINITE causes the caller to block indefinitely until some message is eventually available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no message is available on entry.

Returns:

The number of bytes available from the received message is returned upon success, which might be greater than the actual number of bytes copied to the destination buffer if the message has been truncated. Zero is a possible value corresponding to a zero-sized message passed to rt_queue_send() or rt_queue_write(). Otherwise:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.
- -ETIMEDOUT is returned if the absolute *timeout* date is reached before a message arrives.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no message is immediately available on entry.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before any data was available.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.11.2.10 ssize_t rt_queue_receive (RT_QUEUE * *q*, void ** *bufp*, RTIME *timeout*)

Receive a message from a queue. This service retrieves the next message available from the given queue. Unless otherwise specified, the caller is blocked for a given amount of time if no message is immediately available on entry.

Parameters:

q The descriptor address of the message queue to receive from.

bufp A pointer to a memory location which will be written upon success with the address of the received message. Once consumed, the message space should be freed using rt_queue_free().

timeout The number of clock ticks to wait for a message to arrive (see note). Passing TM_INFINITE causes the caller to block indefinitely until some message is eventually available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no message is available on entry.

Returns:

The number of bytes available from the received message is returned upon success. Zero is a possible value corresponding to a zero-sized message passed to rt_queue_send(). Otherwise:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.
- -ETIMEDOUT is returned if *timeout* is different from TM_NONBLOCK and no message is available within the specified amount of time.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no message is immediately available on entry.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before any data was available.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.11.2.11 ssize_t rt_queue_receive_until (RT_QUEUE * *q*, void ** *bufp*, RTIME *timeout*)

Receive a message from a queue (with absolute timeout date). This service retrieves the next message available from the given queue. Unless otherwise specified, the caller is blocked for a given amount of time if no message is immediately available on entry.

Parameters:

q The descriptor address of the message queue to receive from.

bufp A pointer to a memory location which will be written upon success with the address of the received message. Once consumed, the message space should be freed using rt_queue_free().

timeout The absolute date specifying a time limit to wait for a message to arrive (see note). Passing TM_INFINITE causes the caller to block indefinitely until some message is eventually available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no message is available on entry.

Returns:

The number of bytes available from the received message is returned upon success. Zero is a possible value corresponding to a zero-sized message passed to rt_queue_send(). Otherwise:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.
- -ETIMEDOUT is returned if the absolute *timeout* date is reached before a message arrives.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no message is immediately available on entry.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before any data was available.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.11.2.12 int rt_queue_send (RT_QUEUE * *q*, void * *mbuf*, size_t *size*, int *mode*)

Send a message to a queue. This service sends a complete message to a given queue. The message must have been allocated by a previous call to rt_queue_alloc().

Parameters:

q The descriptor address of the message queue to send to.

mbuf The address of the message buffer to be sent. The message buffer must have been allocated using the rt_queue_alloc() service. Once passed to rt_queue_send(), the memory pointed to by *mbuf* is no more under the control of the sender and thus should not be referenced by it anymore; deallocation of this memory must be handled on the receiving side.

size The size in bytes of the message. Zero is a valid value, in which case an empty message will be sent.

mode A set of flags affecting the operation:

- Q_URGENT causes the message to be prepended to the message queue, ensuring a LIFO ordering.
- Q_NORMAL causes the message to be appended to the message queue, ensuring a FIFO ordering.
- Q_BROADCAST causes the message to be sent to all tasks currently waiting for messages. The message is not copied; a reference count is maintained instead so that the message will remain valid until the last receiver releases its own reference using rt_queue_free(), after which the message space will be returned to the queue's internal pool.

Returns:

Upon success, this service returns the number of receivers which got awaken as a result of the operation. If zero is returned, no task was waiting on the receiving side of the queue, and the message has been enqueued. Upon error, one of the following error codes is returned:

- -EINVAL is returned if *q* is not a message queue descriptor, or *mbuf* is not a valid message buffer obtained from a previous call to rt_queue_alloc().
- -EIDRM is returned if *q* is a deleted queue descriptor.
- -ENOMEM is returned if queuing the message would exceed the limit defined for the queue at creation.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

Referenced by rt_queue_write().

4.11.2.13 int rt_queue_unbind (RT_QUEUE * *q*)

Unbind from a shared message queue. This user-space only service unbinds the calling task from the message queue object previously retrieved by a call to rt_queue_bind().

Unbinding from a message queue when it is no more needed is especially important in order to properly release the mapping resources used to attach the shared queue memory to the caller's address space.

Parameters:

q The address of a queue descriptor to unbind from.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *q* is invalid or not bound.

This service can be called from:

- User-space task.

Rescheduling: never.

Examples:

msg_queue.c.

4.11.2.14 int rt_queue_write (RT_QUEUE * *q*, const void * *buf*, size_t *size*, int *mode*)

Write a message to a queue. This service writes a complete message to a given queue. This service differs from rt_queue_send() in that it accepts a pointer to the raw data to be sent, instead of a canned message buffer.

Parameters:

q The descriptor address of the message queue to write to.

buf The address of the message data to be written to the queue. A message buffer will be allocated internally to convey the data.

size The size in bytes of the message data. Zero is a valid value, in which case an empty message will be sent.

mode A set of flags affecting the operation:

- Q_URGENT causes the message to be prepended to the message queue, ensuring a LIFO ordering.
- Q_NORMAL causes the message to be appended to the message queue, ensuring a FIFO ordering.
- Q_BROADCAST causes the message to be sent to all tasks currently waiting for messages. The message is not copied; a reference count is maintained instead so that the message will remain valid until all receivers get a copy of the message, after which the message space will be returned to the queue's internal pool.

Returns:

Upon success, this service returns the number of receivers which got awaken as a result of the operation. If zero is returned, no task was waiting on the receiving side of the queue, and the message has been enqueued. Upon error, one of the following error codes is returned:

- -EINVAL is returned if *q* is not a message queue descriptor.
- -EIDRM is returned if *q* is a deleted queue descriptor.
- -ENOMEM is returned if queuing the message would exceed the limit defined for the queue at creation, or if no memory can be obtained to convey the message data internally.
- -ESRCH is returned if a *q* represents a stale userland handle

Environments:

This service can be called from:

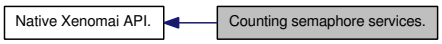
- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

References rt_queue_alloc(), and rt_queue_send().

4.12 Counting semaphore services.

Collaboration diagram for Counting semaphore services.:



Files

- file sem.c
  This file is part of the Xenomai project.

Functions

- int rt_sem_create (RT_SEM *sem, const char *name, unsigned long icount, int mode)
  Create a counting semaphore.
- int rt_sem_delete (RT_SEM *sem)
  Delete a semaphore.
- int rt_sem_p (RT_SEM *sem, RTIME timeout)
  Pend on a semaphore.
- int rt_sem_p_until (RT_SEM *sem, RTIME timeout)
  Pend on a semaphore (with absolute timeout date).
- int rt_sem_v (RT_SEM *sem)
  Signal a semaphore.
- int rt_sem_broadcast (RT_SEM *sem)
  Broadcast a semaphore.
- int rt_sem_inquire (RT_SEM *sem, RT_SEM_INFO *info)
  Inquire about a semaphore.
- int rt_sem_bind (RT_SEM *sem, const char *name, RTIME timeout)
  Bind to a semaphore.
- static int rt_sem_unbind (RT_SEM *sem)
  Unbind from a semaphore.

4.12.1 Detailed Description

A counting semaphore is a synchronization object granting Xenomai tasks a concurrent access to a given number of resources maintained in an internal counter variable. The semaphore is used through the P ("Proberen", from the Dutch "test and decrement") and V ("Verhogen", increment)

operations. The P operation waits for a unit to become available from the count, and the V operation releases a resource by incrementing the unit count by one.

If no more than a single resource is made available at any point in time, the semaphore enforces mutual exclusion and thus can be used to serialize access to a critical section. However, mutexes should be used instead in order to prevent priority inversions.

4.12.2 Function Documentation

4.12.2.1 int rt_sem_bind (RT_SEM * *sem*, const char * *name*, RTIME *timeout*)

Bind to a semaphore. This user-space only service retrieves the uniform descriptor of a given Xenomai semaphore identified by its symbolic name. If the semaphore does not exist on entry, this service blocks the caller until a semaphore of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the semaphore to bind to.

sem The address of a semaphore descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *sem* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.12.2.2 int rt_sem_broadcast (RT_SEM * *sem*)

Broadcast a semaphore. Unblock all tasks waiting on a semaphore. Awaken tasks return from rt_sem_p() as if the semaphore has been signaled. The semaphore count is zeroed as a result of the operation.

Parameters:

sem The descriptor address of the affected semaphore.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *sem* is not a semaphore descriptor.
- -EIDRM is returned if *sem* is a deleted semaphore descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.12.2.3 int rt_sem_create (RT_SEM * *sem*, const char * *name*, unsigned long *icount*, int *mode*)

Create a counting semaphore.

Parameters:

sem The address of a semaphore descriptor Xenomai will use to store the semaphore-related data. This descriptor must always be valid while the semaphore is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the semaphore. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created semaphore.

icount The initial value of the semaphore count.

mode The semaphore creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new semaphore:

- S_FIFO makes tasks pend in FIFO order on the semaphore.
- S_PRIO makes tasks pend in priority order on the semaphore.

- S_PULSE causes the semaphore to behave in "pulse" mode. In this mode, the V (signal) operation attempts to release a single waiter each time it is called, but without incrementing the semaphore count if no waiter is pending. For this reason, the semaphore count in pulse mode remains zero.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the semaphore.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -EINVAL is returned if the *icount* is non-zero and *mode* specifies a pulse semaphore.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

References rt_sem_delete().

4.12.2.4 int rt_sem_delete (RT_SEM * *sem*)

Delete a semaphore. Destroy a semaphore and release all the tasks currently pending on it. A semaphore exists in the system since rt_sem_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

sem The descriptor address of the affected semaphore.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *sem* is not a semaphore descriptor.
- -EIDRM is returned if *sem* is a deleted semaphore descriptor.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Referenced by rt_sem_create().

4.12.2.5 int rt_sem_inquire (RT_SEM * *sem*, RT_SEM_INFO * *info*)

Inquire about a semaphore. Return various information about the status of a given semaphore.

Parameters:

sem The descriptor address of the inquired semaphore.

info The address of a structure the semaphore information will be written to.

Returns:

0 is returned and status information is written to the structure pointed at by *info* upon success. Otherwise:

- -EINVAL is returned if *sem* is not a semaphore descriptor.
- -EIDRM is returned if *sem* is a deleted semaphore descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.12.2.6 int rt_sem_p (RT_SEM * *sem*, RTIME *timeout*)

Pend on a semaphore. Acquire a semaphore unit. If the semaphore value is greater than zero, it is decremented by one and the service immediately returns to the caller. Otherwise, the caller is blocked until the semaphore is either signaled or destroyed, unless a non-blocking operation has been required.

Parameters:

sem The descriptor address of the affected semaphore.

timeout The number of clock ticks to wait for a semaphore unit to be available (see note). Passing TM_INFINITE causes the caller to block indefinitely until a unit is available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no unit is available.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *sem* is not a semaphore descriptor.
- -EIDRM is returned if *sem* is a deleted semaphore descriptor, including if the deletion occurred while the caller was sleeping on it for a unit to become available.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the semaphore value is zero.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before a semaphore unit has become available.
- -ETIMEDOUT is returned if no unit is available within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.12.2.7 int rt_sem_p_until (RT_SEM * *sem*, RTIME *timeout*)

Pend on a semaphore (with absolute timeout date). Acquire a semaphore unit. If the semaphore value is greater than zero, it is decremented by one and the service immediately returns to the caller. Otherwise, the caller is blocked until the semaphore is either signaled or destroyed, unless a non-blocking operation has been required.

Parameters:

sem The descriptor address of the affected semaphore.

timeout The absolute date specifying a time limit to wait for a semaphore unit to be available (see note). Passing TM_INFINITE causes the caller to block indefinitely until a unit is available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no unit is available.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *sem* is not a semaphore descriptor.
- -EIDRM is returned if *sem* is a deleted semaphore descriptor, including if the deletion occurred while the caller was sleeping on it for a unit to become available.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the semaphore value is zero.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before a semaphore unit has become available.
- -ETIMEDOUT is returned if the absolute *timeout* date is reached before a semaphore unit is available.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *timeout* is equal to TM_NONBLOCK.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.12.2.8 int rt_sem_unbind (RT_SEM * *sem*) [inline, static]

Unbind from a semaphore. This user-space only service unbinds the calling task from the semaphore object previously retrieved by a call to rt_sem_bind().

Parameters:

sem The address of a semaphore descriptor to unbind from.

Returns:

0 is always returned.

This service can be called from:

- User-space task.

Rescheduling: never.

4.12.2.9 int rt_sem_v (RT_SEM * *sem*)

Signal a semaphore. Release a semaphore unit. If the semaphore is pended, the first waiting task (by queuing order) is immediately unblocked; otherwise, the semaphore value is incremented by one.

Parameters:

sem The descriptor address of the affected semaphore.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *sem* is not a semaphore descriptor.
- -EIDRM is returned if *sem* is a deleted semaphore descriptor.

Environments:

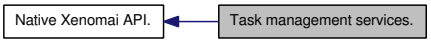
This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.13 Task management services.

Collaboration diagram for Task management services.:



Files

- file task.c
  This file is part of the Xenomai project.

Functions

- int rt_task_create (RT_TASK *task, const char *name, int stksize, int prio, int mode)
  Create a new real-time task.
- int rt_task_start (RT_TASK *task, void(*entry)(void *cookie), void *cookie)
  Start a real-time task.
- int rt_task_suspend (RT_TASK *task)
  Suspend a real-time task.
- int rt_task_resume (RT_TASK *task)
  Resume a real-time task.
- int rt_task_delete (RT_TASK *task)
  Delete a real-time task.
- int rt_task_yield (void)
  Manual round-robin.
- int rt_task_set_periodic (RT_TASK *task, RTIME idate, RTIME period)
  Make a real-time task periodic.
- int rt_task_wait_period (unsigned long *overruns_r)
  Wait for the next periodic release point.
- int rt_task_set_priority (RT_TASK *task, int prio)
  Change the base priority of a real-time task.
- int rt_task_sleep (RTIME delay)
  Delay the calling task (relative).
- int rt_task_sleep_until (RTIME date)
  Delay the calling task (absolute).

- int rt_task_unblock (RT_TASK *task)
  Unblock a real-time task.
- int rt_task_inquire (RT_TASK *task, RT_TASK_INFO *info)
  Inquire about a real-time task.
- int rt_task_add_hook (int type, void(*routine)(void *cookie))
  Install a task hook.
- int rt_task_remove_hook (int type, void(*routine)(void *cookie))
  Remove a task hook.
- int rt_task_catch (void(*handler)(rt_sigset_t))
  Install a signal handler.
- int rt_task_notify (RT_TASK *task, rt_sigset_t signals)
  Send signals to a task.
- int rt_task_set_mode (int clrmask, int setmask, int *mode_r)
  Change task mode bits.
- RT_TASK * rt_task_self (void)
  Retrieve the current task.
- int rt_task_slice (RT_TASK *task, RTIME quantum)
  Set a task's round-robin quantum.
- ssize_t rt_task_send (RT_TASK *task, RT_TASK_MCB *mcb_s, RT_TASK_MCB *mcb_r, RTIME timeout)
  Send a message to a task.
- int rt_task_receive (RT_TASK_MCB *mcb_r, RTIME timeout)
  Receive a message from a task.
- int rt_task_reply (int flowid, RT_TASK_MCB *mcb_s)
  Reply to a task.
- static int rt_task_spawn (RT_TASK *task, const char *name, int stksize, int prio, int mode, void(*entry)(void *cookie), void *cookie)
  Spawn a new real-time task.
- int rt_task_shadow (RT_TASK *task, const char *name, int prio, int mode)
  Turns the current Linux task into a native Xenomai task.
- int rt_task_bind (RT_TASK *task, const char *name, RTIME timeout)
  Bind to a real-time task.
- static int rt_task_unbind (RT_TASK *task)
  Unbind from a real-time task.

- int rt_task_join (RT_TASK *task)

  Wait on the termination of a real-time task.

4.13.1 Detailed Description

Xenomai provides a set of multitasking mechanisms. The basic process object performing actions in Xenomai is a task, a logically complete path of application code. Each Xenomai task is an independent portion of the overall application code embodied in a C procedure, which executes on its own stack context.

The Xenomai scheduler ensures that concurrent tasks are run according to one of the supported scheduling policies. Currently, the Xenomai scheduler supports fixed priority-based FIFO and round-robin policies.

4.13.2 Function Documentation

4.13.2.1 int rt_task_add_hook (int *type*, void(*)(void *cookie) *routine*)

Install a task hook. The real-time kernel allows to register user-defined routines which get called whenever a specific scheduling event occurs. Multiple hooks can be chained for a single event type, and get called on a FIFO basis.

The scheduling is locked while a hook is executing.

Parameters:

type Defines the kind of hook to install:

- T_HOOK_START: The user-defined routine will be called on behalf of the starter task whenever a new task starts. An opaque cookie is passed to the routine which can use it to retrieve the descriptor address of the started task through the T_DESC() macro.

- T_HOOK_DELETE: The user-defined routine will be called on behalf of the deletor task whenever a task is deleted. An opaque cookie is passed to the routine which can use it to retrieve the descriptor address of the deleted task through the T_DESC() macro.

- T_HOOK_SWITCH: The user-defined routine will be called on behalf of the resuming task whenever a context switch takes place. An opaque cookie is passed to the routine which can use it to retrieve the descriptor address of the task which has been switched in through the T_DESC() macro.

Parameters:

routine The address of the user-supplied routine to call.

Returns:

0 is returned upon success. Otherwise, one of the following error codes indicates the cause of the failure:

- -EINVAL is returned if *type* is incorrect.

- -ENOMEM is returned if not enough memory is available from the system heap to add the new hook.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task

Rescheduling: never.

4.13.2.2 int rt_task_bind (RT_TASK * *task*, const char * *name*, RTIME *timeout*)

Bind to a real-time task. This user-space only service retrieves the uniform descriptor of a given Xenomai task identified by its symbolic name. If the task does not exist on entry, this service blocks the caller until a task of the given name is created.

Parameters:

name A valid NULL-terminated name which identifies the task to bind to.

task The address of a task descriptor retrieved by the operation. Contents of this memory is undefined upon failure.

timeout The number of clock ticks to wait for the registration to occur (see note). Passing TM_INFINITE causes the caller to block indefinitely until the object is registered. Passing TM_NONBLOCK causes the service to return immediately without waiting if the object is not registered on entry.

Returns:

0 is returned upon success. Otherwise:

- -EFAULT is returned if *task* or *name* is referencing invalid memory.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and the searched object is not registered on entry.
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.
- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

Environments:

This service can be called from:

- User-space task (switches to primary mode)

Rescheduling: always unless the request is immediately satisfied or *timeout* specifies a non-blocking operation.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

Examples:

bound_task.c.

4.13.2.3 int rt_task_catch (void(*)(rt_sigset_t) *handler*)

Install a signal handler. This service installs a signal handler for the current task. Signals are discrete events tasks can receive each time they resume execution. When signals are pending upon resumption, *handler* is fired to process them. Signals can be sent using rt_task_notify(). A task can block the signal delivery by passing the T_NOSIG bit to rt_task_set_mode().

Calling this service implicitely unblocks the signal delivery for the caller.

Parameters:

handler The address of the user-supplied routine to fire when signals are pending for the task. This handler is passed the set of pending signals as its first and only argument.

Returns:

0 upon success, or:

- -EPERM is returned if this service was not called from a real-time task context.

Environments:

This service can be called from:

- Kernel-based task

Rescheduling: possible.

4.13.2.4 int rt_task_create (RT_TASK * *task*, const char * *name*, int *stksize*, int *prio*, int *mode*)

Create a new real-time task. Creates a real-time task, either running in a kernel module or in user-space depending on the caller's context.

Parameters:

task The address of a task descriptor Xenomai will use to store the task-related data. This descriptor must always be valid while the task is active therefore it must be allocated in permanent memory.

The task is left in an innocuous state until it is actually started by rt_task_start().

Parameters:

name An ASCII string standing for the symbolic name of the task. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created task.

stksize The size of the stack (in bytes) for the new task. If zero is passed, a reasonable pre-defined size will be substituted.

prio The base priority of the new task. This value must range from [0 .. 99] (inclusive) where 0 is the lowest effective priority.

mode The task creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new task:

- T_FPU allows the task to use the FPU whenever available on the platform. This flag is forced for user-space tasks.
- T_SUSP causes the task to start in suspended mode. In such a case, the thread will have to be explicitly resumed using the rt_task_resume() service for its execution to actually begin.
- T_CPU(cpuid) makes the new task affine to CPU # **cpuid**. CPU identifiers range from 0 to RTHAL_NR_CPUS - 1 (inclusive).
- T_JOINABLE (user-space only) allows another task to wait on the termination of the new task. This implies that rt_task_join() is actually called for this task to clean up any user-space located resources after its termination.

Passing T_FPU|T_CPU(1) in the *mode* parameter thus creates a task with FPU support enabled and which will be affine to CPU #1.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to create or register the task.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Note:

When creating or shadowing a Xenomai thread for the first time in user-space, Xenomai installs a handler for the SIGWINCH signal. If you had installed a handler before that, it will be automatically called by Xenomai for SIGWINCH signals that it has not sent.

If, however, you install a signal handler for SIGWINCH after creating or shadowing the first Xenomai thread, you have to explicitly call the function xeno_sigwinch_handler at the beginning of your signal handler, using its return to know if the signal was in fact an internal signal of Xenomai (in which case it returns 1), or if you should handle the signal (in which case it returns 0). xeno_sigwinch_handler prototype is:

int xeno_sigwinch_handler(int sig, siginfo_t *si, void *ctxt);

Which means that you should register your handler with sigaction, using the SA_SIGINFO flag, and pass all the arguments you received to xeno_sigwinch_handler.

Referenced by rt_task_spawn().

4.13.2.5 int rt_task_delete (RT_TASK * *task*)

Delete a real-time task. Terminate a task and release all the real-time kernel resources it currently holds. A task exists in the system since rt_task_create() has been called to create it, so this service must be called in order to destroy it afterwards.

Native tasks implement a mechanism by which they are immune from deletion by other tasks while they run into a deemed safe section of code. This feature is used internally by the native skin in order to prevent tasks from being deleted in the middle of a critical section, without resorting to interrupt masking when the latter is not an option. For this reason, the caller of rt_task_delete() might be blocked and a rescheduling take place, waiting for the target task to exit such critical section.

The DELETE hooks are called on behalf of the calling context (if any). The information stored in the task control block remains valid until all hooks have been called.

Parameters:

task The descriptor address of the affected task. If *task* is NULL, the current task is deleted.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor.
- -EPERM is returned if *task* is NULL but not called from a task context, or this service was called from an asynchronous context.
- -EINTR is returned if rt_task_unblock() has been invoked for the caller while it was waiting for *task* to exit a safe section. In such a case, the deletion process has been aborted and *task* remains unaffected.
- -EIDRM is returned if *task* is a deleted task descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code only if *task* is non-NULL.
- Kernel-based task
- Any user-space context (conforming call)

Rescheduling: always if *task* is NULL, and possible if the deleted task is currently running into a safe section.

References rt_task_self().

4.13.2.6 int rt_task_inquire (RT_TASK * *task*, RT_TASK_INFO * *info*)

Inquire about a real-time task. Return various information about the status of a given task.

Parameters:

task The descriptor address of the inquired task. If *task* is NULL, the current task is inquired.

info The address of a structure the task information will be written to. Passing NULL is valid, in which case the system is only probed for existence of the specified task.

Returns:

0 is returned if the task exists, and status information is written to the structure pointed at by *info* if non-NULL. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor.
- -EPERM is returned if *task* is NULL but not called from a task context.
- -EIDRM is returned if *task* is a deleted task descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *task* is non-NULL.
- Kernel-based task
- User-space task

Rescheduling: never.

References rt_task_info::bprio, rt_task_info::cprio, rt_task_info::ctxswitches, rt_task_info::exectime, rt_task_info::modeswitches, rt_task_info::name, rt_task_info::pagefaults, rt_task_info::relpoint, and rt_task_info::status.

4.13.2.7 int rt_task_join (RT_TASK * *task*)

Wait on the termination of a real-time task. This user-space only service blocks the caller in non-real-time context until *task* has terminated. Note that the specified task must have been created with the T_JOINABLE mode flag set.

Parameters:

task The address of a task descriptor to join.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if the task was not created with T_JOINABLE set or some other task is already waiting on the termination.
- -EDEADLK is returned if *task* refers to the caller.

This service can be called from:

- User-space task.

Rescheduling: always unless the task was already terminated.

4.13.2.8 int rt_task_notify (RT_TASK * *task*, rt_sigset_t *signals*)

Send signals to a task. This service sends a set of signals to a given task. A task can install a signal handler using the rt_task_catch() service to process them.

Parameters:

task The descriptor address of the affected task which must have been previously created by the rt_task_create() service.

signals The set of signals to make pending for the task. This set is OR'ed with the current set of pending signals for the task; there is no count of occurence maintained for each available signal, which is either pending or cleared.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor.
- -EPERM is returned if *task* is NULL but not called from a real-time task context.
- -EIDRM is returned if *task* is a deleted task descriptor.
- -ESRCH is returned if *task* has not set any signal handler.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *task* is non-NULL.

- Kernel-based task
- User-space task

Rescheduling: possible.

4.13.2.9 int rt_task_receive (RT_TASK_MCB * *mcb_r*, RTIME *timeout*)

Receive a message from a task. This service is part of the synchronous message passing support available to Xenomai tasks. It allows the caller to receive a variable-sized message sent from another task using the rt_task_send() service. The sending task is blocked until the caller invokes rt_task_reply() to finish the transaction.

A basic message control block is used to store the location and size of the data area to receive from the client, in addition to a user-defined operation code.

Parameters:

mcb_r The address of a message control block referring to the receive message area. The fields from this control block should be set as follows:

- mcb_r->data should contain the address of a buffer large enough to collect the data sent by the remote task;

- mcb_r->size should contain the size in bytes of the buffer space pointed at by mcb_r->data. If mcb_r->size is lower than the actual size of the received message, no data copy takes place and -ENOBUFS is returned to the caller. See note.

Upon return, mcb_r->opcode will contain the operation code sent from the remote task using rt_task_send().

Parameters:

timeout The number of clock ticks to wait for receiving a message (see note). Passing TM_INFINITE causes the caller to block indefinitely until a remote task eventually sends a message. Passing TM_NONBLOCK causes the service to return immediately without waiting if no remote task is currently waiting for sending a message.

Returns:

A strictly positive value is returned upon success, representing a flow identifier for the opening transaction; this token should be passed to rt_task_reply(), in order to send back a reply to and unblock the remote task appropriately. Otherwise:

- -ENOBUFS is returned if *mcb_r* does not point at a message area large enough to collect the remote task's message.

- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and no remote task is currently waiting for sending a message to the caller.

- -ETIMEDOUT is returned if no message was received within the *timeout*.
- -EINTR is returned if rt_task_unblock() has been called for the caller before any message was available.
- -EPERM is returned if this service was called from a context which cannot sleep (e.g. interrupt, non-realtime or scheduler locked).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: Always.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.
When called from a user-space task, this service may need to allocate some temporary buffer space from the system heap to hold the received data if the size of the latter exceeds a certain amount; the threshold before allocation is currently set to 64 bytes.

References rt_task_mcb::data, rt_task_mcb::opcode, and rt_task_mcb::size.

4.13.2.10 int rt_task_remove_hook (int *type*, void(*)(void *cookie) *routine*)

Remove a task hook. This service allows to remove a task hook previously registered using rt_task_add_hook().

Parameters:

type Defines the kind of hook to uninstall. Possible values are:

- T_HOOK_START
- T_HOOK_DELETE
- T_HOOK_SWITCH

Parameters:

routine The address of the user-supplied routine to remove from the hook list.

Returns:

0 is returned upon success. Otherwise, one of the following error codes indicates the cause of the failure:

- -EINVAL is returned if *type* is incorrect.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task

Rescheduling: never.

4.13.2.11 int rt_task_reply (int *flowid*, RT_TASK_MCB * *mcb_s*)

Reply to a task. This service is part of the synchronous message passing support available to Xenomai tasks. It allows the caller to send back a variable-sized message to the client task, once the initial message from this task has been pulled using rt_task_receive() and processed. As a consequence of this call, the remote task will be unblocked from the rt_task_send() service.

A basic message control block is used to store the location and size of the data area to send back, in addition to a user-defined status code.

Parameters:

flowid The flow identifier returned by a previous call to rt_task_receive() which uniquely identifies the current transaction.

mcb_s The address of an optional message control block referring to the message to be sent back. If *mcb_s* is NULL, the client will be unblocked without getting any reply data. When *mcb_s* is valid, the fields from this control block should be set as follows:

- mcb_s->data should contain the address of the payload data to send to the remote task.
- mcb_s->size should contain the size in bytes of the payload data pointed at by mcb_s->data. 0 is a legitimate value, and indicates that no payload data will be transferred. In the latter case, mcb_s->data will be ignored. See note.
- mcb_s->opcode is an opaque status code carried during the message transfer the caller can fill with any appropriate value. It will be made available "as is" to the remote task into the status code field by the rt_task_send() service. If *mcb_s* is NULL, 0 will be returned to the client into the status code field.

Returns:

O is returned upon success. Otherwise:

- -ENXIO is returned if *flowid* does not match the expected identifier returned from the latest call of the current task to rt_task_receive(), or if the remote task stopped waiting for the reply in the meantime (e.g. the client could have been deleted or forcibly unblocked).
- -EPERM is returned if this service was called from an invalid context (e.g. interrupt, or non-primary).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: Always.

Note:

When called from a user-space task, this service may need to allocate some temporary buffer space from the system heap to hold the reply data if the size of the latter exceeds a certain amount; the threshold before allocation is currently set to 64 bytes.

References rt_task_mcb::data, rt_task_mcb::opcode, and rt_task_mcb::size.

4.13.2.12 int rt_task_resume (RT_TASK * *task*)

Resume a real-time task. Forcibly resume the execution of a task which has been previously suspended by a call to rt_task_suspend().

The suspension nesting count is decremented so that rt_task_resume() will only resume the task if this count falls down to zero as a result of the current invocation.

Parameters:

task The descriptor address of the affected task.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor.
- -EIDRM is returned if *task* is a deleted task descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible if the suspension nesting level falls down to zero as a result of the current invocation.

4.13.2.13 RT_TASK* rt_task_self (void)

Retrieve the current task. Return the current task descriptor address.

Returns:

The address of the caller's task descriptor is returned upon success, or NULL if the calling context is asynchronous (i.e. not a Xenomai task).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine Those will cause a NULL return.
- Kernel-based task
- User-space task

Rescheduling: never.

Referenced by rt_task_delete().

4.13.2.14 ssize_t rt_task_send (RT_TASK * *task*, RT_TASK_MCB * *mcb_s*, RT_TASK_MCB * *mcb_r*, RTIME *timeout*)

Send a message to a task. This service is part of the synchronous message passing support available to Xenomai tasks. It allows the caller to send a variable-sized message to another task, waiting for the remote to receive the initial message by a call to rt_task_receive(), then reply to it using rt_task_reply().

A basic message control block is used to store the location and size of the data area to send or retrieve upon reply, in addition to a user-defined operation code.

Parameters:

task The descriptor address of the recipient task.

mcb_s The address of the message control block referring to the message to be sent. The fields from this control block should be set as follows:

- mcb_s->data should contain the address of the payload data to send to the remote task.
- mcb_s->size should contain the size in bytes of the payload data pointed at by mcb_s->data. 0 is a legitimate value, and indicates that no payload data will be transferred. In the latter case, mcb_s->data will be ignored. See note.
- mcb_s->opcode is an opaque operation code carried during the message transfer the caller can fill with any appropriate value. It will be made available "as is" to the remote task into the operation code field by the rt_task_receive() service.

Parameters:

mcb_r The address of an optional message control block referring to the reply message area. If *mcb_r* is NULL and a reply is sent back by the remote task, the reply message will be discarded, and -ENOBUFS will be returned to the caller. When *mcb_r* is valid, the fields from this control block should be set as follows:

- mcb_r->data should contain the address of a buffer large enough to collect the reply data from the remote task.

- mcb_r->size should contain the size in bytes of the buffer space pointed at by mcb_r->data. If mcb_r->size is lower than the actual size of the reply message, no data copy takes place and -ENOBUFS is returned to the caller. See note.

Upon return, mcb_r->opcode will contain the status code sent back from the remote task using rt_task_reply(), or 0 if unspecified.

Parameters:

timeout The number of clock ticks to wait for the remote task to reply to the initial message (see note). Passing TM_INFINITE causes the caller to block indefinitely until the remote task eventually replies. Passing TM_NONBLOCK causes the service to return immediately without waiting if the remote task is not waiting for messages (i.e. if *task* is not currently blocked on the rt_task_receive() service); however, the caller will wait indefinitely for a reply from that remote task if present.

Returns:

A positive value is returned upon success, representing the length (in bytes) of the reply message returned by the remote task. 0 is a success status, meaning either that *mcb_r* was NULL on entry, or that no actual message was passed to the remote call to rt_task_reply(). Otherwise:

- -ENOBUFS is returned if *mcb_r* does not point at a message area large enough to collect the remote task's reply. This includes the case where *mcb_r* is NULL on entry albeit the remote task attempts to send a reply message.

- -EWOULDBLOCK is returned if *timeout* is equal to TM_NONBLOCK and *task* is not currently blocked on the rt_task_receive() service.

- -EIDRM is returned if *task* has been deleted while waiting for a reply.

- -EINTR is returned if rt_task_unblock() has been called for the caller before any reply was available.

- -EPERM is returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime context).

- -ESRCH is returned if *task* cannot be found (when called from user-space only).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: Always.

Note:

The *timeout* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.
When called from a user-space task, this service may need to allocate some temporary buffer space from the system heap to hold both the sent and the reply data if this cumulated size exceeds a certain amount; the threshold before allocation is currently set to 64 bytes.

References rt_task_mcb::data, rt_task_mcb::flowid, rt_task_mcb::opcode, and rt_task_mcb::size.

4.13.2.15 int rt_task_set_mode (int *clrmask*, int *setmask*, int * *mode_r*)

Change task mode bits. Each Xenomai task has a set of internal bits determining various operating conditions; the rt_task_set_mode() service allows to alter three of them, respectively controlling:

- whether the task locks the rescheduling procedure,
- whether the task undergoes a round-robin scheduling,
- whether the task blocks the delivery of signals.

To this end, rt_task_set_mode() takes a bitmask of mode bits to clear for disabling the corresponding modes, and another one to set for enabling them. The mode bits which were previously in effect can be returned upon request.

The following bits can be part of the bitmask:

- T_LOCK causes the current task to lock the scheduler. Clearing this bit unlocks the scheduler.
- T_NOSIG disables the asynchronous signal delivery for the current task.
- When set, T_WARNSW causes the SIGXCPU signal to be sent to the current user-space task whenever it switches to the secondary mode. This feature is useful to detect unwanted migrations to the Linux domain.
- T_RPIOFF disables thread priority coupling between Xenomai and Linux schedulers. This bit prevents the root Linux thread from inheriting the priority of the running shadow Xenomai thread. Use CONFIG_XENO_OPT_RPIOFF to globally disable priority coupling.
- T_PRIMARY can be passed to switch the current user-space task to primary mode (setmask |= T_PRIMARY), or secondary mode (clrmask |= T_PRIMARY). Upon return from rt_task_set_mode(), the user-space task will run into the specified domain.

Normally, this service can only be called on behalf of a regular real-time task, either running in kernel or user-space. However, as a special exception, requests for setting/clearing the T_LOCK bit from asynchronous contexts are silently dropped, and the call returns successfully if no other mode bits have been specified. This is consistent with the fact that Xenomai enforces a scheduler lock until the outer interrupt handler has returned.

Parameters:

clrmask A bitmask of mode bits to clear for the current task, before *setmask* is applied. 0 is an acceptable value which leads to a no-op.

setmask A bitmask of mode bits to set for the current task. 0 is an acceptable value which leads to a no-op.

mode_r If non-NULL, *mode_r* must be a pointer to a memory location which will be written upon success with the previous set of active mode bits. If NULL, the previous set of active mode bits will not be returned.

Returns:

0 is returned upon success, or:

- -EINVAL if either *setmask* or *clrmask* specifies invalid bits. T_PRIMARY is invalid for kernel-based tasks.
- -EPERM is returned if this service was not called from a real-time task context.

Environments:

This service can be called from:

- Kernel-based task
- User-space task

Rescheduling: possible, if T_LOCK has been passed into *clrmask* and the calling context is a task.

References T_LOCK, T_NOSIG, T_RPIOFF, and T_WARNSW.

4.13.2.16 int rt_task_set_periodic (RT_TASK * *task*, RTIME *idate*, RTIME *period*)

Make a real-time task periodic. Make a task periodic by programing its first release point and its period in the processor time line. Subsequent calls to rt_task_wait_period() will delay the task until the next periodic release point in the processor timeline is reached.

Parameters:

task The descriptor address of the affected task. This task is immediately delayed until the first periodic release point is reached. If *task* is NULL, the current task is set periodic.

idate The initial (absolute) date of the first release point, expressed in clock ticks (see note). The affected task will be delayed until this point is reached. If *idate* is equal to TM_NOW, the current system date is used, and no initial delay takes place.

period The period of the task, expressed in clock ticks (see note). Passing TM_INFINITE attempts to stop the task's periodic timer; in the latter case, the routine always exits succesfully, regardless of the previous state of this timer.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor, or *period* is different from TM_INFINITE but shorter than the scheduling latency value for the target system, as available from /proc/xenomai/latency.
- -EIDRM is returned if *task* is a deleted task descriptor.

- -ETIMEDOUT is returned if *idate* is different from TM_INFINITE and represents a date in the past.
- -EWOULDBLOCK is returned if the system timer is not active.
- -EPERM is returned if *task* is NULL but not called from a task context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code or interrupt only if *task* is non-NULL.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always if the operation affects the current task and *idate* has not elapsed yet.

Note:

The *idate* and *period* values will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.13.2.17 int rt_task_set_priority (RT_TASK * *task*, int *prio*)

Change the base priority of a real-time task. Changing the base priority of a task does not affect the priority boost the target task might have obtained as a consequence of a previous priority inheritance.

Parameters:

task The descriptor address of the affected task.

prio The new task priority. This value must range from [0 .. 99] (inclusive) where 0 is the lowest effective priority.

Returns:

Upon success, the previously set priority is returned. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor, or if *prio* is invalid.
- -EPERM is returned if *task* is NULL but not called from a task context.
- -EIDRM is returned if *task* is a deleted task descriptor.

Side-effects:

- This service calls the rescheduling procedure.
- Assigning the same priority to a running or ready task moves it to the end of its priority group, thus causing a manual round-robin.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *task* is non-NULL.
- Kernel-based task
- User-space task

Rescheduling: possible if *task* is the current one.

4.13.2.18 int rt_task_shadow (RT_TASK * *task*, const char * *name*, int *prio*, int *mode*)

Turns the current Linux task into a native Xenomai task. Creates a real-time task running in the context of the calling regular Linux task in user-space.

Parameters:

task In non-NULL, the address of a task descriptor Xenomai will use to store the task-related data; this descriptor must always be valid while the task is active therefore it must be allocated in permanent memory. If NULL is passed, then the descriptor will not be returned; main() threads which do not need to be referred to by other threads may use this syntax to promote themselves to the real-time domain for instance.

Note:

Allowing for a NULL descriptor pointer to be passed is a recent feature which is not available with any earlier Xenomai release.

The current context is switched to primary execution mode and returns immediately, unless T_SUSP has been passed in the *mode* parameter.

Parameters:

name An ASCII string standing for the symbolic name of the task. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created task.

prio The base priority which will be set for the current task. This value must range from [0 .. 99] (inclusive) where 0 is the lowest effective priority.

mode The task creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new task:

- T_FPU allows the task to use the FPU whenever available on the platform. This flag is forced for this call, therefore it can be omitted.
- T_SUSP causes the task to enter the suspended mode after it has been put under Xenomai's control. In such a case, a call to rt_task_resume() will be needed to wake up the current task.
- T_CPU(cpuid) makes the current task affine to CPU # **cpuid**. CPU identifiers range from 0 to RTHAL_NR_CPUS - 1 (inclusive). The calling task will migrate to another processor before this service returns if the current one is not part of the CPU affinity mask.

Passing T_CPU(0)|T_CPU(1) in the *mode* parameter thus defines a task affine to CPUs #0 and #1.

Returns:

0 is returned upon success. Otherwise:

- -EBUSY is returned if the current Linux task is already mapped to a Xenomai context.
- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to create or register the task.
- -EEXIST is returned if the *name* is already in use by some registered object.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- User-space task (enters primary mode)

Rescheduling: possible.

Note:

When creating or shadowing a Xenomai thread for the first time in user-space, Xenomai installs a handler for the SIGWINCH signal. If you had installed a handler before that, it will be automatically called by Xenomai for SIGWINCH signals that it has not sent.

If, however, you install a signal handler for SIGWINCH after creating or shadowing the first Xenomai thread, you have to explicitly call the function xeno_sigwinch_handler at the beginning of your signal handler, using its return to know if the signal was in fact an internal signal of Xenomai (in which case it returns 1), or if you should handle the signal (in which case it returns 0). xeno_sigwinch_handler prototype is:

int xeno_sigwinch_handler(int sig, siginfo_t *si, void *ctxt);

Which means that you should register your handler with sigaction, using the SA_SIGINFO flag, and pass all the arguments you received to xeno_sigwinch_handler.

References T_WARNSW.

4.13.2.19 int rt_task_sleep (RTIME *delay*)

Delay the calling task (relative). Delay the execution of the calling task for a number of internal clock ticks.

Parameters:

delay The number of clock ticks to wait before resuming the task (see note). Passing zero causes the task to return immediately with no delay.

Returns:

0 is returned upon success, otherwise:

- -EINTR is returned if rt_task_unblock() has been called for the sleeping task before the sleep time has elapsed.
- -EWOULDBLOCK is returned if the system timer is inactive.
- -EPERM is returned if this service was called from a context which cannot sleep (e.g. interrupt, non-realtime or scheduler locked).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless a null delay is given.

Note:

The *delay* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.13.2.20 int rt_task_sleep_until (RTIME *date*)

Delay the calling task (absolute). Delay the execution of the calling task until a given date is reached.

Parameters:

date The absolute date in clock ticks to wait before resuming the task (see note). As a special case, TM_INFINITE is an acceptable value that makes the caller block indefinitely, until rt_task_unblock() is called against it. Otherwise, any wake up date in the past causes the task to return immediately with no delay.

Returns:

0 is returned upon success. Otherwise:

- -EINTR is returned if rt_task_unblock() has been called for the sleeping task before the sleep time has elapsed.
- -ETIMEDOUT is returned if *date* has already elapsed.
- -EWOULDBLOCK is returned if the system timer is inactive, and

  Date:

  is valid but different from TM_INFINITE.

- -EPERM is returned if this service was called from a context which cannot sleep (e.g. interrupt, non-realtime or scheduler locked).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always unless a date in the past is given.

Note:

The *date* value will be interpreted as jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

4.13.2.21 int rt_task_slice (RT_TASK * *task*, RTIME *quantum*)

Set a task's round-robin quantum. Set the time credit allotted to a task undergoing the round-robin scheduling. If *quantum* is non-zero, rt_task_slice() also refills the current quantum for the target task, otherwise, time-slicing is stopped for that task.

Parameters:

task The descriptor address of the affected task. If *task* is NULL, the current task is considered.

quantum The round-robin quantum for the task expressed in ticks (see note).

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor.
- -EPERM is returned if *task* is NULL but not called from a task context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *task* is non-NULL.
- Kernel-based task
- User-space task

Rescheduling: never.

Note:

The *quantum* value is always interpreted as a count of ticks. If the task undergoes aperiodic timing, the tick duration is defined by CONFIG_XENO_OPT_TIMING_VIRTICK.

4.13.2.22 int rt_task_spawn (RT_TASK * *task*, const char * *name*, int *stksize*, int *prio*, int *mode*, void(*)(void *cookie) *entry*, void * *cookie*) [inline, static]

Spawn a new real-time task. Creates and immediately starts a real-time task, either running in a kernel module or in user-space depending on the caller's context. This service is a simple shorthand for rt_task_create() followed by a call to rt_task_start().

Parameters:

task The address of a task descriptor Xenomai will use to store the task-related data. This descriptor must always be valid while the task is active therefore it must be allocated in permanent memory.

name An ASCII string standing for the symbolic name of the task. When non-NULL and non-empty, this string is copied to a safe place into the descriptor, and passed to the registry package if enabled for indexing the created task.

stksize The size of the stack (in bytes) for the new task. If zero is passed, a reasonable pre-defined size will be substituted.

prio The base priority of the new task. This value must range from [0 .. 99] (inclusive) where 0 is the lowest effective priority.

mode The task creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new task:

- T_FPU allows the task to use the FPU whenever available on the platform. This flag is forced for user-space tasks.

- T_SUSP causes the task to start in suspended mode. In such a case, the thread will have to be explicitly resumed using the rt_task_resume() service for its execution to actually begin.

- T_CPU(cpuid) makes the new task affine to CPU # **cpuid**. CPU identifiers range from 0 to RTHAL_NR_CPUS - 1 (inclusive).

- T_JOINABLE (user-space only) allows another task to wait on the termination of the new task. This implies that rt_task_join() is actually called for this task to clean up any user-space located resources after its termination.

Passing T_FPU|T_CPU(1) in the *mode* parameter thus creates a task with FPU support enabled and which will be affine to CPU #1.

Parameters:

entry The address of the task's body routine. In other words, it is the task entry point.

cookie A user-defined opaque cookie the real-time kernel will pass to the emerging task as the sole argument of its entry point.

Returns:

0 is returned upon success. Otherwise:

- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to create the new task's stack space or register the task.

- -EEXIST is returned if the *name* is already in use by some registered object.

- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Note:

When creating or shadowing a Xenomai thread for the first time in user-space, Xenomai installs a handler for the SIGWINCH signal. If you had installed a handler before that, it will be automatically called by Xenomai for SIGWINCH signals that it has not sent.

If, however, you install a signal handler for SIGWINCH after creating or shadowing the first Xenomai thread, you have to explicitly call the function xeno_sigwinch_handler at the beginning of your signal handler, using its return to know if the signal was in fact an internal signal of Xenomai (in which case it returns 1), or if you should handle the signal (in which case it returns 0). xeno_sigwinch_handler prototype is:

int xeno_sigwinch_handler(int sig, siginfo_t *si, void *ctxt);

Which means that you should register your handler with sigaction, using the SA_SIGINFO flag, and pass all the arguments you received to xeno_sigwinch_handler.

References rt_task_create(), and rt_task_start().

4.13.2.23 int rt_task_start (RT_TASK * *task*, void(*)(void *cookie) *entry*, void * *cookie*)

Start a real-time task. Start a (newly) created task, scheduling it for the first time. This call releases the target task from the dormant state.

The TSTART hooks are called on behalf of the calling context (if any, see rt_task_add_hook()).

Parameters:

task The descriptor address of the affected task which must have been previously created by the rt_task_create() service.

entry The address of the task's body routine. In other words, it is the task entry point.

cookie A user-defined opaque cookie the real-time kernel will pass to the emerging task as the sole argument of its entry point.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor.

- -EIDRM is returned if *task* is a deleted task descriptor.
- -EBUSY is returned if *task* is already started.
- -EPERM is returned if this service was called from an asynchronous context.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Kernel-based task
- User-space task

Rescheduling: possible.

Referenced by rt_task_spawn().

4.13.2.24 int rt_task_suspend (RT_TASK * *task*)

Suspend a real-time task. Forcibly suspend the execution of a task. This task will not be eligible for scheduling until it is explicitly resumed by a call to rt_task_resume(). In other words, the suspended state caused by a call to rt_task_suspend() is cumulative with respect to the delayed and blocked states caused by other services, and is managed separately from them.

A nesting count is maintained so that rt_task_suspend() and rt_task_resume() must be used in pairs.

Receiving a Linux signal causes the suspended task to resume immediately.

Parameters:

task The descriptor address of the affected task. If *task* is NULL, the current task is suspended.

Returns:

0 is returned upon success. Otherwise:

- -EINTR is returned if a Linux signal has been received by the suspended task.
- -EINVAL is returned if *task* is not a task descriptor.
- -EPERM is returned if this service was called from an invalid context (e.g. interrupt, non-realtime context).
- -EIDRM is returned if *task* is a deleted task descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Interrupt service routine only if *task* is non-NULL.
- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always if *task* is NULL.

4.13.2.25 int rt_task_unbind (RT_TASK * *task*) [inline, static]

Unbind from a real-time task. This user-space only service unbinds the calling task from the task object previously retrieved by a call to rt_task_bind().

Parameters:

task The address of a task descriptor to unbind from.

Returns:

0 is always returned.

This service can be called from:

- User-space task.

Rescheduling: never.

4.13.2.26 int rt_task_unblock (RT_TASK * *task*)

Unblock a real-time task. Break the task out of any wait it is currently in. This call clears all delay and/or resource wait condition for the target task. However, rt_task_unblock() does not resume a task which has been forcibly suspended by a previous call to rt_task_suspend(). If all suspensive conditions are gone, the task becomes eligible anew for scheduling.

Parameters:

task The descriptor address of the affected task.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *task* is not a task descriptor.
- -EIDRM is returned if *task* is a deleted task descriptor.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: possible.

4.13.2.27 int rt_task_wait_period (unsigned long * *overruns_r*)

Wait for the next periodic release point. Make the current task wait for the next periodic release point in the processor time line.

Parameters:

overruns_r If non-NULL, *overruns_r* must be a pointer to a memory location which will be written with the count of pending overruns. This value is copied only when rt_task_wait_period() returns -ETIMEDOUT or success; the memory location remains unmodified otherwise. If NULL, this count will never be copied back.

Returns:

0 is returned upon success; if *overruns_r* is valid, zero is copied to the pointed memory location. Otherwise:

- -EWOULDBLOCK is returned if rt_task_set_periodic() has not previously been called for the calling task.
- -EINTR is returned if rt_task_unblock() has been called for the waiting task before the next periodic release point has been reached. In this case, the overrun counter is reset too.
- -ETIMEDOUT is returned if a timer overrun occurred, which indicates that a previous release point has been missed by the calling task. If *overruns_r* is valid, the count of pending overruns is copied to the pointed memory location.
- -EPERM is returned if this service was called from a context which cannot sleep (e.g. interrupt, non-realtime or scheduler locked).

Environments:

This service can be called from:

- Kernel-based task
- User-space task (switches to primary mode)

Rescheduling: always, unless the current release point has already been reached. In the latter case, the current task immediately returns from this service without being delayed.

4.13.2.28 int rt_task_yield (void)

Manual round-robin. Move the current task to the end of its priority group, so that the next equal-priority task in ready state is switched in.

Returns:

0 is returned upon success. Otherwise:

- -EPERM is returned if this service was called from a context which cannot sleep (e.g. interrupt, non-realtime or scheduler locked).

Environments:

This service can be called from:

- Kernel-based task
- User-space task

Rescheduling: always if a next equal-priority task is ready to run, otherwise, this service leads to a no-op.

4.14 Timer management services.

Collaboration diagram for Timer management services.:

Data Structures

- struct rt_timer_info

  Structure containing timer-information useful to users.

Files

- file timer.h

  This file is part of the Xenomai project.

- file timer.c

  This file is part of the Xenomai project.

Typedefs

- typedef struct rt_timer_info RT_TIMER_INFO

  Structure containing timer-information useful to users.

Functions

- SRTIME rt_timer_ns2tsc (SRTIME ns)

  Convert nanoseconds to local CPU clock ticks.

- SRTIME rt_timer_tsc2ns (SRTIME ticks)

  Convert local CPU clock ticks to nanoseconds.

- RTIME rt_timer_tsc (void)

  Return the current TSC value.

- RTIME rt_timer_read (void)

  Return the current system time.

- SRTIME rt_timer_ns2ticks (SRTIME ns)

  Convert nanoseconds to internal clock ticks.

- SRTIME rt_timer_ticks2ns (SRTIME ticks)

  Convert internal clock ticks to nanoseconds.

- int rt_timer_inquire (RT_TIMER_INFO *info)
  Inquire about the timer.
- void rt_timer_spin (RTIME ns)
  Busy wait burning CPU cycles.
- int rt_timer_set_mode (RTIME nstick)
  Set the system clock rate.

4.14.1 Detailed Description

Timer-related services allow to control the Xenomai system timer which is used in all timed operations.

4.14.2 Typedef Documentation

4.14.2.1 typedef struct rt_timer_info RT_TIMER_INFO

Structure containing timer-information useful to users.

See also:

rt_timer_inquire()

4.14.3 Function Documentation

4.14.3.1 int rt_timer_inquire (RT_TIMER_INFO * *info*)

Inquire about the timer. Return various information about the status of the system timer.

Parameters:

info The address of a structure the timer information will be written to.

Returns:

This service always returns 0.

The information block returns the period and the current system date. The period can have the following values:

- TM_UNSET is a special value indicating that the system timer is inactive. A call to rt_timer_set_mode() re-activates it.
- TM_ONESHOT is a special value indicating that the timer has been set up in oneshot mode.
- Any other period value indicates that the system timer is currently running in periodic mode; it is a count of nanoseconds representing the period of the timer, i.e. the duration of a periodic tick or "jiffy".

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.14.3.2 SRTIME rt_timer_ns2ticks (SRTIME *ns*)

Convert nanoseconds to internal clock ticks. Convert a count of nanoseconds to internal clock ticks. This routine operates on signed nanosecond values.

Parameters:

ns The count of nanoseconds to convert.

Returns:

The corresponding value expressed in internal clock ticks.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.14.3.3 SRTIME rt_timer_ns2tsc (SRTIME *ns*)

Convert nanoseconds to local CPU clock ticks. Convert a count of nanoseconds to local CPU clock ticks. This routine operates on signed nanosecond values.

Parameters:

ns The count of nanoseconds to convert.

Returns:

The corresponding value expressed in CPU clock ticks.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.14.3.4 RTIME rt_timer_read (void)

Return the current system time. Return the current time maintained by the master time base.

Returns:

The current time expressed in clock ticks (see note).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

Note:

The value returned will represent a count of jiffies if the native skin is bound to a periodic time base (see CONFIG_XENO_OPT_NATIVE_PERIOD), or nanoseconds otherwise.

Examples:

trivial-periodic.c.

4.14.3.5 int rt_timer_set_mode (RTIME *nstick*)

Set the system clock rate. This routine switches to periodic timing mode and sets the clock tick rate, or resets the current timing mode to aperiodic/oneshot mode depending on the value of the *nstick* parameter. Since the native skin automatically sets its time base according to the configured policy and period at load time (see CONFIG_XENO_OPT_NATIVE_PERIOD), calling rt_timer_set_mode() is not required from applications unless the pre-defined mode and period need to be changed dynamically.

This service sets the time unit which will be relevant when specifying time intervals to the services taking timeout or delays as input parameters. In periodic mode, clock ticks will represent periodic jiffies. In oneshot mode, clock ticks will represent nanoseconds.

Parameters:

nstick The time base period in nanoseconds. If this parameter is equal to the special TM_-ONESHOT value, the time base is set to operate in a tick-less fashion (i.e. oneshot mode). Other values are interpreted as the time between two consecutive clock ticks in periodic timing mode (i.e. clock HZ = 1e9 / nstick).

Returns:

0 is returned on success. Otherwise:

- -ENODEV is returned if the underlying architecture does not support the requested periodic timing. Aperiodic/oneshot timing is always supported.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task

Rescheduling: never.

4.14.3.6 void rt_timer_spin (RTIME *ns*)

Busy wait burning CPU cycles. Enter a busy waiting loop for a count of nanoseconds. The precision of this service largely depends on the availability of a time stamp counter on the current CPU.

Since this service is usually called with interrupts enabled, the caller might be preempted by other real-time activities, therefore the actual delay might be longer than specified.

Parameters:

ns The time to wait expressed in nanoseconds.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.14.3.7 SRTIME rt_timer_ticks2ns (SRTIME *ticks*)

Convert internal clock ticks to nanoseconds. Convert a count of internal clock ticks to nanoseconds. This routine operates on signed tick values.

Parameters:

ticks The count of internal clock ticks to convert.

Returns:

The corresponding value expressed in nanoseconds.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.14.3.8 RTIME rt_timer_tsc (void)

Return the current TSC value. Return the value of the time stamp counter (TSC) maintained by the CPU of the underlying architecture.

Returns:

The current value of the TSC.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

4.14.3.9 SRTIME rt_timer_tsc2ns (SRTIME *ticks*)

Convert local CPU clock ticks to nanoseconds. Convert a local CPU clock ticks to nanoseconds. This routine operates on signed tick values.

Parameters:

ticks The count of local CPU clock ticks to convert.

Returns:

The corresponding value expressed in nanoseconds.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine
- Kernel-based task
- User-space task

Rescheduling: never.

Chapter 5

Data Structure Documentation

5.1 rt_heap_info Struct Reference

Structure containing heap-information useful to users.

5.1.1 Detailed Description

Structure containing heap-information useful to users.

See also:

rt_heap_inquire()

The documentation for this struct was generated from the following file:

- include/native/heap.h

5.2 rt_mutex_info Struct Reference

Structure containing mutex information useful to users.

Data Fields

- int locked

  > 0 if mutex is locked.

- int nwaiters

  Number of pending tasks.

- char name [XNOBJECT_NAME_LEN]

  Symbolic name.

- char owner [XNOBJECT_NAME_LEN]

  Symbolic name of the current owner, empty if unlocked.

5.2.1 Detailed Description

Structure containing mutex information useful to users.

See also:

rt_mutex_inquire()

5.2.2 Field Documentation

5.2.2.1 int rt_mutex_info::locked

> 0 if mutex is locked.

Referenced by rt_mutex_inquire().

5.2.2.2 char rt_mutex_info::name[XNOBJECT_NAME_LEN]

Symbolic name.

Referenced by rt_mutex_inquire().

5.2.2.3 int rt_mutex_info::nwaiters

Number of pending tasks.

Referenced by rt_mutex_inquire().

5.2.2.4 char rt_mutex_info::owner[XNOBJECT_NAME_LEN]

Symbolic name of the current owner, empty if unlocked.

Referenced by rt_mutex_inquire().

The documentation for this struct was generated from the following file:

- include/native/mutex.h

5.3 rt_task_info Struct Reference

Structure containing task-information useful to users.

Data Fields

- int bprio
  Base priority.
- int cprio
  Current priority.
- unsigned status
  Task's status.
- RTIME relpoint
  Time of next release.
- char name [XNOBJECT_NAME_LEN]
  Symbolic name assigned at creation.
- RTIME exectime
  Execution time in primary mode in nanoseconds.
- int modeswitches
  Number of primary->secondary mode switches.
- int ctxswitches
  Number of context switches.
- int pagefaults
  Number of triggered page faults.

5.3.1 Detailed Description

Structure containing task-information useful to users.

See also:

rt_task_inquire()

5.3.2 Field Documentation

5.3.2.1 int rt_task_info::bprio

Base priority.

Referenced by rt_task_inquire().

5.3.2.2 int rt_task_info::cprio

Current priority. May change through Priority Inheritance.

Referenced by rt_task_inquire().

5.3.2.3 int rt_task_info::ctxswitches

Number of context switches.

Referenced by rt_task_inquire().

5.3.2.4 RTIME rt_task_info::exectime

Execution time in primary mode in nanoseconds.

Referenced by rt_task_inquire().

5.3.2.5 int rt_task_info::modeswitches

Number of primary->secondary mode switches.

Referenced by rt_task_inquire().

5.3.2.6 char rt_task_info::name[XNOBJECT_NAME_LEN]

Symbolic name assigned at creation.

Referenced by rt_task_inquire().

5.3.2.7 int rt_task_info::pagefaults

Number of triggered page faults.

Referenced by rt_task_inquire().

5.3.2.8 RTIME rt_task_info::relpoint

Time of next release.

Referenced by rt_task_inquire().

5.3.2.9 unsigned rt_task_info::status

Task's status.

See also:

Task Status

Referenced by rt_task_inquire().

The documentation for this struct was generated from the following file:

- include/native/task.h

5.4 rt_task_mcb Struct Reference

Structure used in passing messages between tasks.

Data Fields

- int flowid

  Flow identifier.

- int opcode

  Operation code.

- caddr_t data

  Message address.

- size_t size

  Message size (bytes).

5.4.1 Detailed Description

Structure used in passing messages between tasks.

See also:

rt_task_send(), rt_task_reply(), rt_task_receive()

5.4.2 Field Documentation

5.4.2.1 caddr_t rt_task_mcb::data

Message address.

Referenced by rt_task_receive(), rt_task_reply(), and rt_task_send().

5.4.2.2 int rt_task_mcb::flowid

Flow identifier.

Referenced by rt_task_send().

5.4.2.3 int rt_task_mcb::opcode

Operation code.

Referenced by rt_task_receive(), rt_task_reply(), and rt_task_send().

5.4.2.4 size_t rt_task_mcb::size

Message size (bytes).

Referenced by rt_task_receive(), rt_task_reply(), and rt_task_send().

The documentation for this struct was generated from the following file:

- include/native/task.h

5.5 rt_timer_info Struct Reference

Structure containing timer-information useful to users.

5.5.1 Detailed Description

Structure containing timer-information useful to users.

See also:

rt_timer_inquire()

The documentation for this struct was generated from the following file:

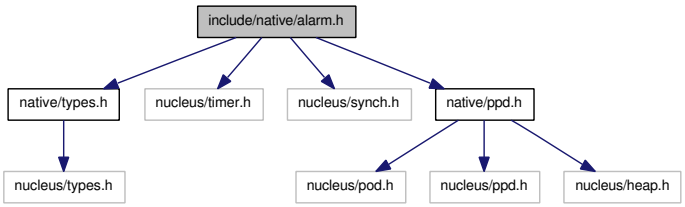
- include/native/timer.h

Chapter 6

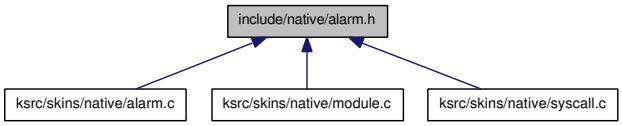
File Documentation

6.1 include/native/alarm.h File Reference

This file is part of the Xenomai project. Include dependency graph for alarm.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_alarm_create (RT_ALARM *alarm, const char *name, rt_alarm_t handler, void *cookie)

  Create an alarm object from kernel space.

- int rt_alarm_delete (RT_ALARM *alarm)

  Delete an alarm.

- int rt_alarm_start (RT_ALARM *alarm, RTIME value, RTIME interval)

  Start an alarm.

- int rt_alarm_stop (RT_ALARM *alarm)

  Stop an alarm.

- int rt_alarm_inquire (RT_ALARM *alarm, RT_ALARM_INFO *info)

  Inquire about an alarm.

6.1.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

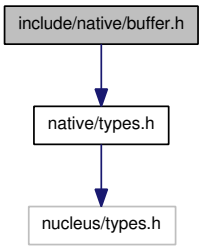
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

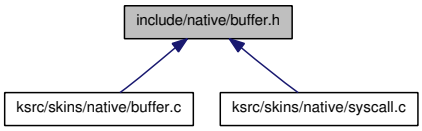
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

6.2 include/native/buffer.h File Reference

This file is part of the Xenomai project. Include dependency graph for buffer.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_buffer_bind (RT_BUFFER *bf, const char *name, RTIME timeout)
  Bind to a buffer.
- static int rt_buffer_unbind (RT_BUFFER *bf)
  Unbind from a buffer.
- int rt_buffer_create (RT_BUFFER *bf, const char *name, size_t bufsz, int mode)
  Create a buffer.
- int rt_buffer_delete (RT_BUFFER *bf)
  Delete a buffer.
- ssize_t rt_buffer_write (RT_BUFFER *bf, const void *ptr, size_t size, RTIME timeout)
  Write to a buffer.
- ssize_t rt_buffer_write_until (RT_BUFFER *bf, const void *ptr, size_t size, RTIME timeout)
  Write to a buffer (with absolute timeout date).
- ssize_t rt_buffer_read (RT_BUFFER *bf, void *ptr, size_t size, RTIME timeout)
  Read from a buffer.
- int rt_buffer_clear (RT_BUFFER *bf)
  Clear a buffer.
- int rt_buffer_inquire (RT_BUFFER *bf, RT_BUFFER_INFO *info)
  Inquire about a buffer.

6.2.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2008 Philippe Gerum <rpm@xenomai.org>

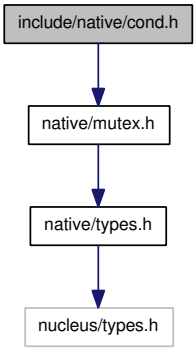
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

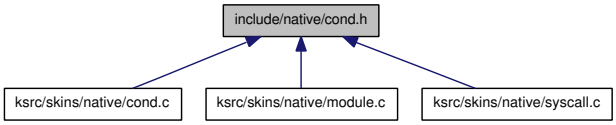
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

6.3 include/native/cond.h File Reference

This file is part of the Xenomai project. Include dependency graph for cond.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_cond_bind (RT_COND *cond, const char *name, RTIME timeout)
  Bind to a condition variable.
- static int rt_cond_unbind (RT_COND *cond)
  Unbind from a condition variable.
- int rt_cond_create (RT_COND *cond, const char *name)
  Create a condition variable.
- int rt_cond_delete (RT_COND *cond)
  Delete a condition variable.
- int rt_cond_signal (RT_COND *cond)
  Signal a condition variable.
- int rt_cond_broadcast (RT_COND *cond)
  Broadcast a condition variable.
- int rt_cond_wait (RT_COND *cond, RT_MUTEX *mutex, RTIME timeout)
  Wait on a condition.
- int rt_cond_wait_until (RT_COND *cond, RT_MUTEX *mutex, RTIME timeout)
  Wait on a condition (with absolute timeout date).

- int rt_cond_inquire (RT_COND *cond, RT_COND_INFO *info)

  Inquire about a condition variable.

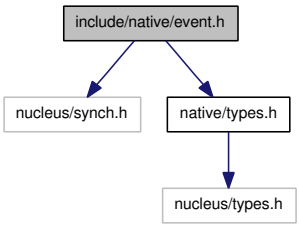
6.3.1 Detailed Description

This file is part of the Xenomai project.

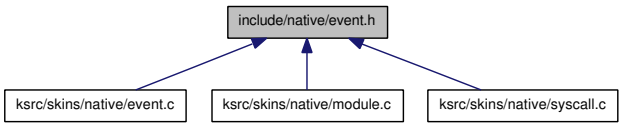
Note:

6.4 include/native/event.h File Reference

This file is part of the Xenomai project. Include dependency graph for event.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_event_bind (RT_EVENT *event, const char *name, RTIME timeout)
  Bind to an event flag group.
- static int rt_event_unbind (RT_EVENT *event)
  Unbind from an event flag group.
- int rt_event_create (RT_EVENT *event, const char *name, unsigned long ivalue, int mode)
  Create an event group.
- int rt_event_delete (RT_EVENT *event)
  Delete an event group.
- int rt_event_signal (RT_EVENT *event, unsigned long mask)
  Post an event group.
- int rt_event_wait (RT_EVENT *event, unsigned long mask, unsigned long *mask_r, int mode, RTIME timeout)
  Pend on an event group.
- int rt_event_wait_until (RT_EVENT *event, unsigned long mask, unsigned long *mask_r, int mode, RTIME timeout)
  Pend on an event group (with absolute timeout date).
- int rt_event_clear (RT_EVENT *event, unsigned long mask, unsigned long *mask_r)
  Clear an event group.
- int rt_event_inquire (RT_EVENT *event, RT_EVENT_INFO *info)

Inquire about an event group.

6.4.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

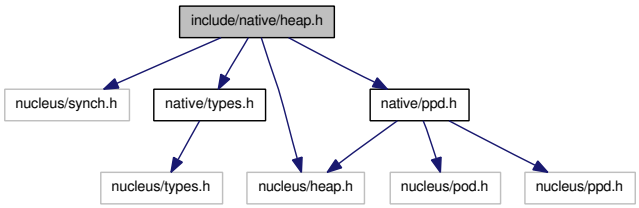
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

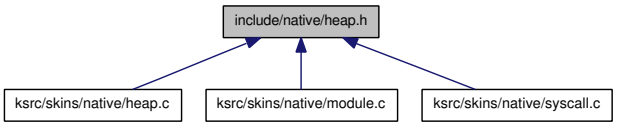
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

6.5 include/native/heap.h File Reference

This file is part of the Xenomai project. Include dependency graph for heap.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct rt_heap_info
  Structure containing heap-information useful to users.

Typedefs

- typedef struct rt_heap_info RT_HEAP_INFO
  Structure containing heap-information useful to users.

Functions

- int rt_heap_create (RT_HEAP *heap, const char *name, size_t heapsize, int mode)
  Create a memory heap or a shared memory segment.
- int rt_heap_delete (RT_HEAP *heap)
  Delete a real-time heap.
- int rt_heap_alloc (RT_HEAP *heap, size_t size, RTIME timeout, void **blockp)
  Allocate a block or return the single segment base.
- int rt_heap_free (RT_HEAP *heap, void *block)
  Free a block.
- int rt_heap_inquire (RT_HEAP *heap, RT_HEAP_INFO *info)
  Inquire about a heap.

6.5.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

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.

6.5.2 Typedef Documentation

6.5.2.1 typedef struct rt_heap_info RT_HEAP_INFO

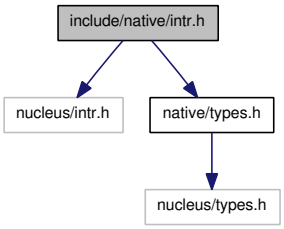
Structure containing heap-information useful to users.

See also:

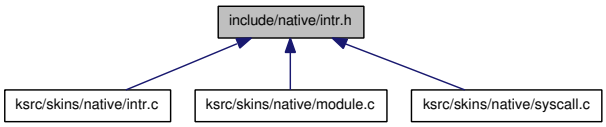
rt_heap_inquire()

6.6 include/native/intr.h File Reference

This file is part of the Xenomai project. Include dependency graph for intr.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_intr_bind (RT_INTR *intr, const char *name, RTIME timeout)

  Bind to an interrupt object.

- static int rt_intr_unbind (RT_INTR *intr)

  Unbind from an interrupt object.

- int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq, int mode)

  Create an interrupt object from user-space.

- int rt_intr_wait (RT_INTR *intr, RTIME timeout)

  Wait for the next interrupt.

- int rt_intr_delete (RT_INTR *intr)

  Delete an interrupt object.

- int rt_intr_enable (RT_INTR *intr)

  Enable an interrupt object.

- int rt_intr_disable (RT_INTR *intr)

  Disable an interrupt object.

- int rt_intr_inquire (RT_INTR *intr, RT_INTR_INFO *info)

  Inquire about an interrupt object.

6.6.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2005 Philippe Gerum <rpm@xenomai.org>

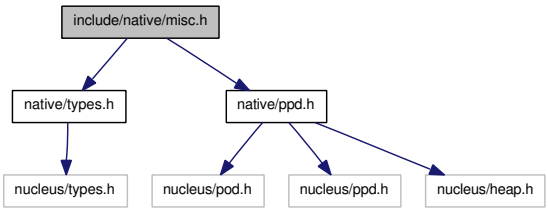
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

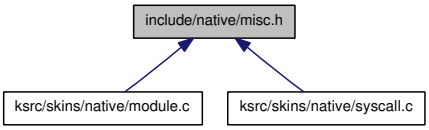
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

6.7 include/native/misc.h File Reference

This file is part of the Xenomai project. Include dependency graph for misc.h:



This graph shows which files directly or indirectly include this file:



6.7.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2005 Philippe Gerum <rpm@xenomai.org>

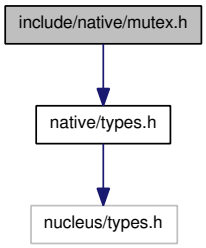
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

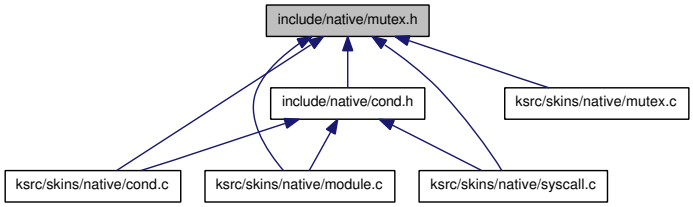
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

6.8 include/native/mutex.h File Reference

This file is part of the Xenomai project. Include dependency graph for mutex.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct rt_mutex_info
  Structure containing mutex information useful to users.

Typedefs

- typedef struct rt_mutex_info RT_MUTEX_INFO
  Structure containing mutex information useful to users.

Functions

- int rt_mutex_bind (RT_MUTEX *mutex, const char *name, RTIME timeout)
  Bind to a mutex.
- static int rt_mutex_unbind (RT_MUTEX *mutex)
  Unbind from a mutex.
- int rt_mutex_create (RT_MUTEX *mutex, const char *name)
  Create a mutex.
- int rt_mutex_delete (RT_MUTEX *mutex)
  Delete a mutex.

- int rt_mutex_acquire (RT_MUTEX *mutex, RTIME timeout)
  Acquire a mutex.
- int rt_mutex_acquire_until (RT_MUTEX *mutex, RTIME timeout)
  Acquire a mutex (with absolute timeout date).
- int rt_mutex_release (RT_MUTEX *mutex)
  Unlock mutex.
- int rt_mutex_inquire (RT_MUTEX *mutex, RT_MUTEX_INFO *info)
  Inquire about a mutex.

6.8.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

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.

6.8.2 Typedef Documentation

6.8.2.1 typedef struct rt_mutex_info RT_MUTEX_INFO

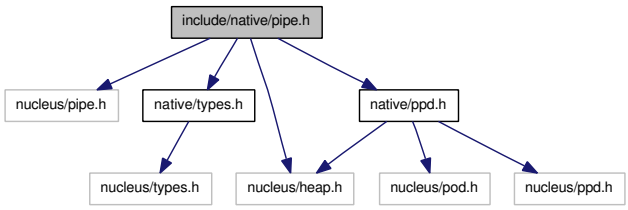
Structure containing mutex information useful to users.

See also:

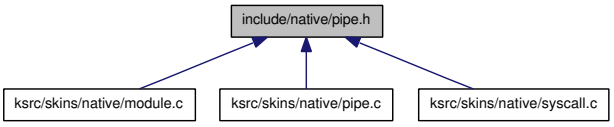
rt_mutex_inquire()

6.9 include/native/pipe.h File Reference

This file is part of the Xenomai project. Include dependency graph for pipe.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_pipe_create (RT_PIPE *pipe, const char *name, int minor, size_t poolsize)
  Create a message pipe.
- int rt_pipe_delete (RT_PIPE *pipe)
  Delete a message pipe.
- ssize_t rt_pipe_read (RT_PIPE *pipe, void *buf, size_t size, RTIME timeout)
  Read a message from a pipe.
- ssize_t rt_pipe_write (RT_PIPE *pipe, const void *buf, size_t size, int mode)
  Write a message to a pipe.
- ssize_t rt_pipe_stream (RT_PIPE *pipe, const void *buf, size_t size)
  Stream bytes to a pipe.
- ssize_t rt_pipe_receive (RT_PIPE *pipe, RT_PIPE_MSG **msg, RTIME timeout)
  Receive a message from a pipe.
- ssize_t rt_pipe_send (RT_PIPE *pipe, RT_PIPE_MSG *msg, size_t size, int mode)
  Send a message through a pipe.
- RT_PIPE_MSG * rt_pipe_alloc (RT_PIPE *pipe, size_t size)
  Allocate a message pipe buffer.
- int rt_pipe_free (RT_PIPE *pipe, RT_PIPE_MSG *msg)
  Free a message pipe buffer.

- int rt_pipe_flush (RT_PIPE *pipe, int mode)
  Flush the i/o queues associated with the kernel endpoint of a message pipe.
- int rt_pipe_monitor (RT_PIPE *pipe, int(*fn)(RT_PIPE *pipe, int event, long arg))
  Monitor a message pipe asynchronously.

6.9.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

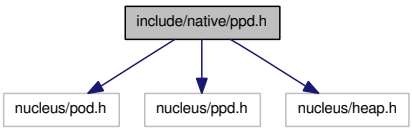
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

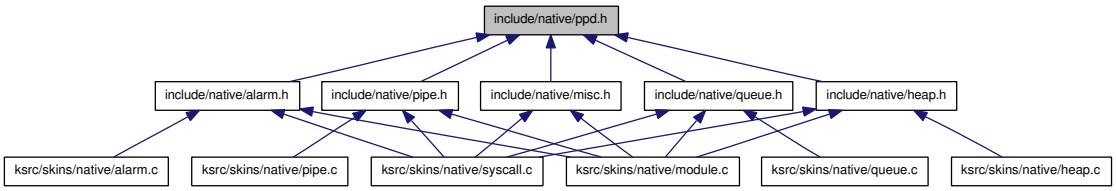
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

6.10 include/native/ppd.h File Reference

This file is part of the Xenomai project. Include dependency graph for ppd.h:



This graph shows which files directly or indirectly include this file:



6.10.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2007 Philippe Gerum <rpm@xenomai.org>

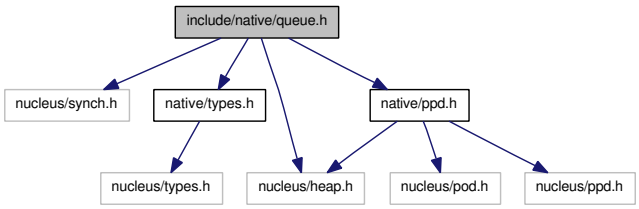
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

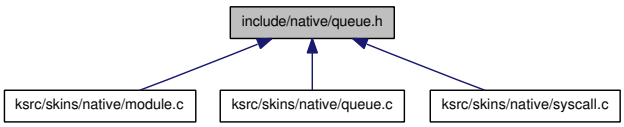
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

6.11 include/native/queue.h File Reference

This file is part of the Xenomai project. Include dependency graph for queue.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_queue_create (RT_QUEUE *q, const char *name, size_t poolsize, size_t qlimit, int mode)

  Create a message queue.

- int rt_queue_delete (RT_QUEUE *q)

  Delete a message queue.

- void * rt_queue_alloc (RT_QUEUE *q, size_t size)

  Allocate a message queue buffer.

- int rt_queue_free (RT_QUEUE *q, void *buf)

  Free a message queue buffer.

- int rt_queue_send (RT_QUEUE *q, void *buf, size_t size, int mode)

  Send a message to a queue.

- int rt_queue_write (RT_QUEUE *q, const void *buf, size_t size, int mode)

  Write a message to a queue.

- ssize_t rt_queue_receive (RT_QUEUE *q, void **bufp, RTIME timeout)

  Receive a message from a queue.

- ssize_t rt_queue_receive_until (RT_QUEUE *q, void **bufp, RTIME timeout)

  Receive a message from a queue (with absolute timeout date).

- ssize_t rt_queue_read (RT_QUEUE *q, void *bufp, size_t size, RTIME timeout)

  Read a message from a queue.

- ssize_t rt_queue_read_until (RT_QUEUE *q, void *bufp, size_t size, RTIME timeout)
  Read a message from a queue (with absolute timeout date).
- int rt_queue_flush (RT_QUEUE *q)
  Flush a message queue.
- int rt_queue_inquire (RT_QUEUE *q, RT_QUEUE_INFO *info)
  Inquire about a message queue.

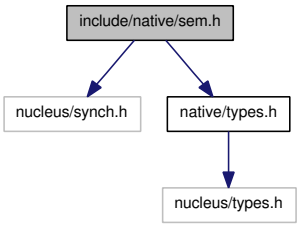
6.11.1 Detailed Description

This file is part of the Xenomai project.

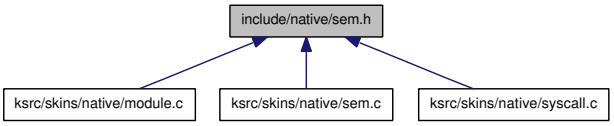
Note:

6.12 include/native/sem.h File Reference

This file is part of the Xenomai project. Include dependency graph for sem.h:



This graph shows which files directly or indirectly include this file:



Functions

- int rt_sem_bind (RT_SEM *sem, const char *name, RTIME timeout)
  Bind to a semaphore.
- static int rt_sem_unbind (RT_SEM *sem)
  Unbind from a semaphore.
- int rt_sem_create (RT_SEM *sem, const char *name, unsigned long icount, int mode)
  Create a counting semaphore.
- int rt_sem_delete (RT_SEM *sem)
  Delete a semaphore.
- int rt_sem_p (RT_SEM *sem, RTIME timeout)
  Pend on a semaphore.
- int rt_sem_p_until (RT_SEM *sem, RTIME timeout)
  Pend on a semaphore (with absolute timeout date).
- int rt_sem_v (RT_SEM *sem)
  Signal a semaphore.
- int rt_sem_broadcast (RT_SEM *sem)
  Broadcast a semaphore.
- int rt_sem_inquire (RT_SEM *sem, RT_SEM_INFO *info)
  Inquire about a semaphore.

6.12.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

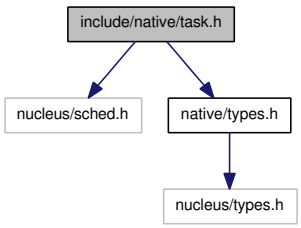
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.13 include/native/task.h File Reference

This file is part of the Xenomai project. Include dependency graph for task.h:



This graph shows which files directly or indirectly include this file:

Data Structures

- struct rt_task_info

  Structure containing task-information useful to users.

- struct rt_task_mcb

  Structure used in passing messages between tasks.

Defines

- #define T_BLOCKED XNPEND

  See XNPEND.

- #define T_DELAYED XNDELAY

  See XNDELAY.

- #define T_READY XNREADY

  See XNREADY.

- #define T_DORMANT XNDORMANT

  See XNDORMANT.

- #define T_STARTED XNSTARTED

  See XNSTARTED.

- #define T_BOOST XNBOOST

  See XNBOOST.

- #define T_LOCK XNLOCK

See XNLOCK.

- #define T_NOSIG XNASDI
  See XNASDI.
- #define T_WARNSW XNTRAPSW
  See XNTRAPSW.
- #define T_RPIOFF XNRPIOFF
  See XNRPIOFF.

Typedefs

- typedef struct rt_task_info RT_TASK_INFO
  Structure containing task-information useful to users.
- typedef struct rt_task_mcb RT_TASK_MCB
  Structure used in passing messages between tasks.

Functions

- int rt_task_shadow (RT_TASK *task, const char *name, int prio, int mode)
  Turns the current Linux task into a native Xenomai task.
- int rt_task_bind (RT_TASK *task, const char *name, RTIME timeout)
  Bind to a real-time task.
- static int rt_task_unbind (RT_TASK *task)
  Unbind from a real-time task.
- int rt_task_join (RT_TASK *task)
  Wait on the termination of a real-time task.
- int rt_task_create (RT_TASK *task, const char *name, int stksize, int prio, int mode) __-deprecated_in_kernel__
  Create a new real-time task.
- int rt_task_start (RT_TASK *task, void(*fun)(void *cookie), void *cookie)
  Start a real-time task.
- int rt_task_suspend (RT_TASK *task)
  Suspend a real-time task.
- int rt_task_resume (RT_TASK *task)
  Resume a real-time task.
- int rt_task_delete (RT_TASK *task)

Delete a real-time task.

- int rt_task_yield (void)
  Manual round-robin.
- int rt_task_set_periodic (RT_TASK *task, RTIME idate, RTIME period)
  Make a real-time task periodic.
- int rt_task_wait_period (unsigned long *overruns_r)
  Wait for the next periodic release point.
- int rt_task_set_priority (RT_TASK *task, int prio)
  Change the base priority of a real-time task.
- int rt_task_sleep (RTIME delay)
  Delay the calling task (relative).
- int rt_task_sleep_until (RTIME date)
  Delay the calling task (absolute).
- int rt_task_unblock (RT_TASK *task)
  Unblock a real-time task.
- int rt_task_inquire (RT_TASK *task, RT_TASK_INFO *info)
  Inquire about a real-time task.
- int rt_task_notify (RT_TASK *task, rt_sigset_t signals)
  Send signals to a task.
- int rt_task_set_mode (int clrmask, int setmask, int *mode_r)
  Change task mode bits.
- RT_TASK * rt_task_self (void)
  Retrieve the current task.
- int rt_task_slice (RT_TASK *task, RTIME quantum)
  Set a task's round-robin quantum.
- ssize_t rt_task_send (RT_TASK *task, RT_TASK_MCB *mcb_s, RT_TASK_MCB *mcb_r, RTIME timeout)
  Send a message to a task.
- int rt_task_receive (RT_TASK_MCB *mcb_r, RTIME timeout)
  Receive a message from a task.
- int rt_task_reply (int flowid, RT_TASK_MCB *mcb_s)
  Reply to a task.
- static int rt_task_spawn (RT_TASK *task, const char *name, int stksize, int prio, int mode, void(*entry)(void *cookie), void *cookie)
  Spawn a new real-time task.

6.13.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

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.

6.13.2 Typedef Documentation

6.13.2.1 typedef struct rt_task_info RT_TASK_INFO

Structure containing task-information useful to users.

See also:

rt_task_inquire()

6.13.2.2 typedef struct rt_task_mcb RT_TASK_MCB

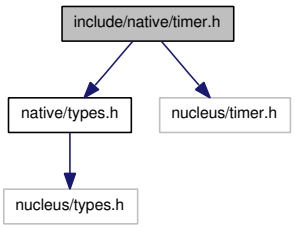
Structure used in passing messages between tasks.

See also:

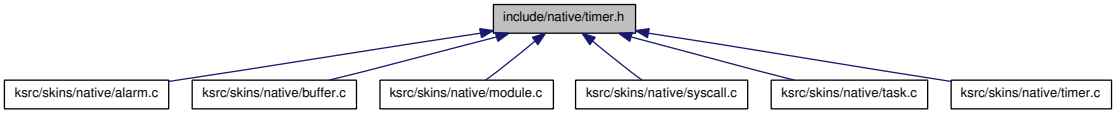
rt_task_send(), rt_task_reply(), rt_task_receive()

6.14 include/native/timer.h File Reference

This file is part of the Xenomai project. Include dependency graph for timer.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct rt_timer_info
  Structure containing timer-information useful to users.

Typedefs

- typedef struct rt_timer_info RT_TIMER_INFO
  Structure containing timer-information useful to users.

Functions

- SRTIME rt_timer_ns2tsc (SRTIME ns)
  Convert nanoseconds to local CPU clock ticks.
- SRTIME rt_timer_tsc2ns (SRTIME ticks)
  Convert local CPU clock ticks to nanoseconds.
- RTIME rt_timer_tsc (void)
  Return the current TSC value.
- RTIME rt_timer_read (void)
  Return the current system time.
- SRTIME rt_timer_ns2ticks (SRTIME ns)
  Convert nanoseconds to internal clock ticks.

- SRTIME rt_timer_ticks2ns (SRTIME ticks)

  Convert internal clock ticks to nanoseconds.

- int rt_timer_inquire (RT_TIMER_INFO *info)

  Inquire about the timer.

- void rt_timer_spin (RTIME ns)

  Busy wait burning CPU cycles.

- int rt_timer_set_mode (RTIME nstick)

  Set the system clock rate.

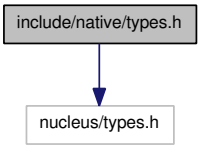
6.14.1 Detailed Description

This file is part of the Xenomai project.

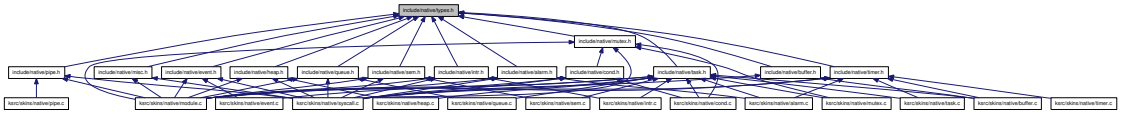
Note:

6.15 include/native/types.h File Reference

This file is part of the Xenomai project. Include dependency graph for types.h:



This graph shows which files directly or indirectly include this file:



6.15.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

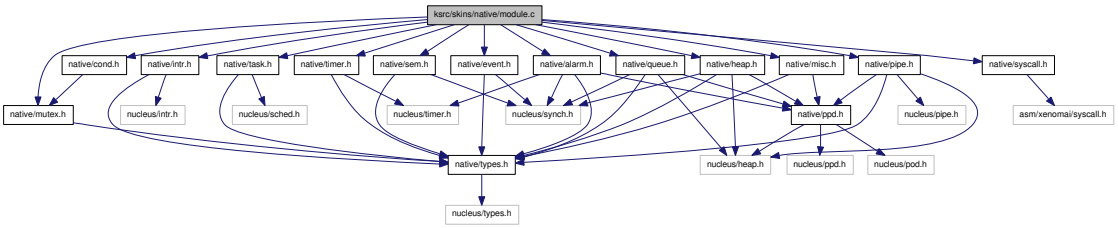
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.16 ksrc/skins/native/module.c File Reference

This file is part of the Xenomai project. Include dependency graph for module.c:



6.16.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

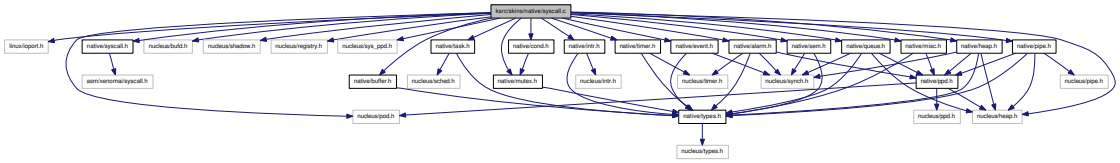
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.17 ksrc/skins/native/syscall.c File Reference

This file is part of the Xenomai project. Include dependency graph for syscall.c:



6.17.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

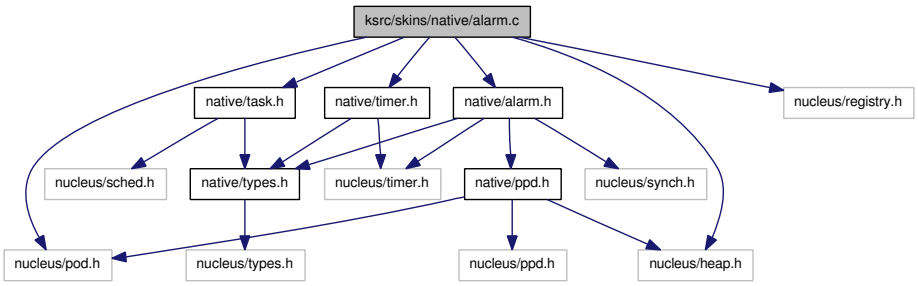
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.18 ksrc/skins/native/alarm.c File Reference

This file is part of the Xenomai project. Include dependency graph for alarm.c:



Functions

- int rt_alarm_create (RT_ALARM *alarm, const char *name, rt_alarm_t handler, void *cookie)

  Create an alarm object from kernel space.

- int rt_alarm_delete (RT_ALARM *alarm)

  Delete an alarm.

- int rt_alarm_start (RT_ALARM *alarm, RTIME value, RTIME interval)

  Start an alarm.

- int rt_alarm_stop (RT_ALARM *alarm)

  Stop an alarm.

- int rt_alarm_inquire (RT_ALARM *alarm, RT_ALARM_INFO *info)

  Inquire about an alarm.

6.18.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

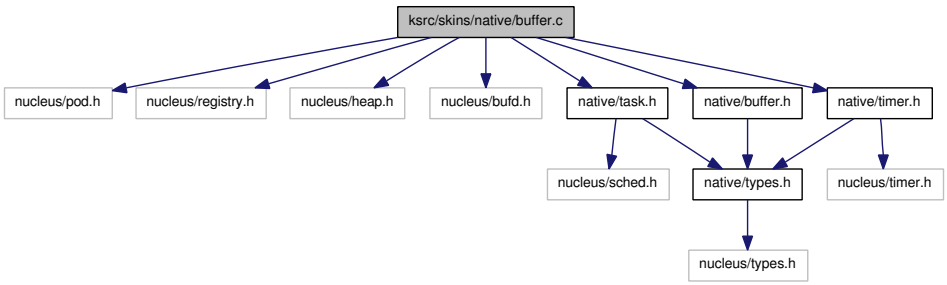
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.19 ksrc/skins/native/buffer.c File Reference

This file is part of the Xenomai project. Include dependency graph for buffer.c:



Functions

- int rt_buffer_create (RT_BUFFER *bf, const char *name, size_t bufsz, int mode)
  Create a buffer.
- int rt_buffer_delete (RT_BUFFER *bf)
  Delete a buffer.
- ssize_t rt_buffer_write (RT_BUFFER *bf, const void *ptr, size_t len, RTIME timeout)
  Write to a buffer.
- ssize_t rt_buffer_write_until (RT_BUFFER *bf, const void *ptr, size_t len, RTIME timeout)
  Write to a buffer (with absolute timeout date).
- ssize_t rt_buffer_read (RT_BUFFER *bf, void *ptr, size_t len, RTIME timeout)
  Read from a buffer.
- int rt_buffer_clear (RT_BUFFER *bf)
  Clear a buffer.
- int rt_buffer_inquire (RT_BUFFER *bf, RT_BUFFER_INFO *info)
  Inquire about a buffer.

6.19.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2008 Philippe Gerum <rpm@xenomai.org>

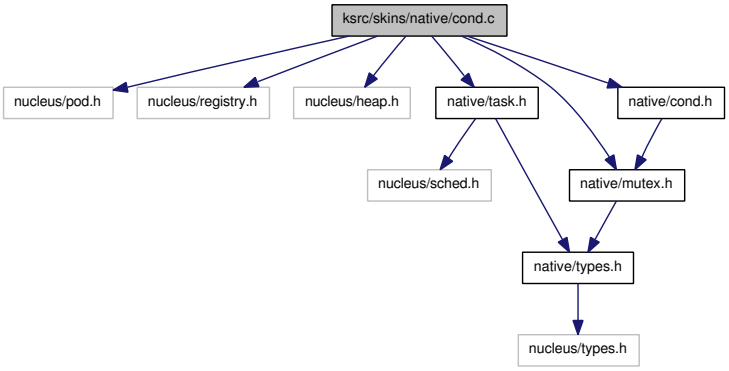
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.20 ksrc/skins/native/cond.c File Reference

This file is part of the Xenomai project. Include dependency graph for cond.c:



Functions

- int rt_cond_create (RT_COND *cond, const char *name)
  Create a condition variable.
- int rt_cond_delete (RT_COND *cond)
  Delete a condition variable.
- int rt_cond_signal (RT_COND *cond)
  Signal a condition variable.
- int rt_cond_broadcast (RT_COND *cond)
  Broadcast a condition variable.
- int rt_cond_wait (RT_COND *cond, RT_MUTEX *mutex, RTIME timeout)
  Wait on a condition.
- int rt_cond_wait_until (RT_COND *cond, RT_MUTEX *mutex, RTIME timeout)
  Wait on a condition (with absolute timeout date).
- int rt_cond_inquire (RT_COND *cond, RT_COND_INFO *info)
  Inquire about a condition variable.

6.20.1 Detailed Description

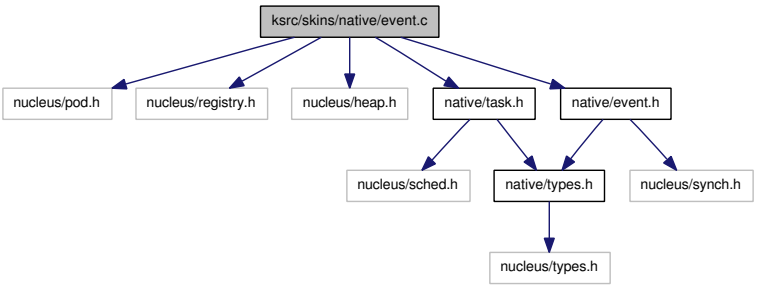
This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

6.21 ksrc/skins/native/event.c File Reference

This file is part of the Xenomai project. Include dependency graph for event.c:



Functions

- int rt_event_create (RT_EVENT *event, const char *name, unsigned long ivalue, int mode)

  Create an event group.

- int rt_event_delete (RT_EVENT *event)

  Delete an event group.

- int rt_event_signal (RT_EVENT *event, unsigned long mask)

  Post an event group.

- int rt_event_wait (RT_EVENT *event, unsigned long mask, unsigned long *mask_r, int mode, RTIME timeout)

  Pend on an event group.

- int rt_event_wait_until (RT_EVENT *event, unsigned long mask, unsigned long *mask_r, int mode, RTIME timeout)

  Pend on an event group (with absolute timeout date).

- int rt_event_clear (RT_EVENT *event, unsigned long mask, unsigned long *mask_r)

  Clear an event group.

- int rt_event_inquire (RT_EVENT *event, RT_EVENT_INFO *info)

  Inquire about an event group.

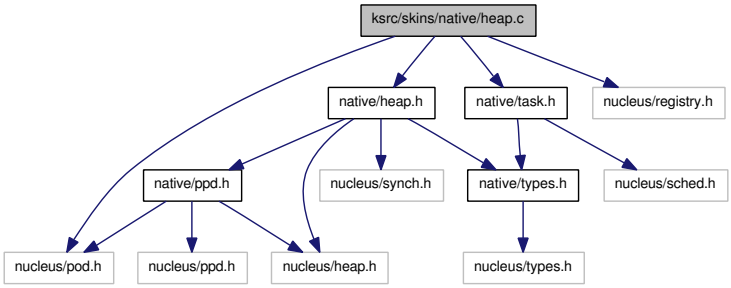
6.21.1 Detailed Description

This file is part of the Xenomai project.

Note:

6.22 ksrc/skins/native/heap.c File Reference

This file is part of the Xenomai project. Include dependency graph for heap.c:



Functions

- int rt_heap_create (RT_HEAP *heap, const char *name, size_t heapsize, int mode)
  Create a memory heap or a shared memory segment.
- int rt_heap_delete (RT_HEAP *heap)
  Delete a real-time heap.
- int rt_heap_alloc (RT_HEAP *heap, size_t size, RTIME timeout, void **blockp)
  Allocate a block or return the single segment base.
- int rt_heap_free (RT_HEAP *heap, void *block)
  Free a block.
- int rt_heap_inquire (RT_HEAP *heap, RT_HEAP_INFO *info)
  Inquire about a heap.

6.22.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

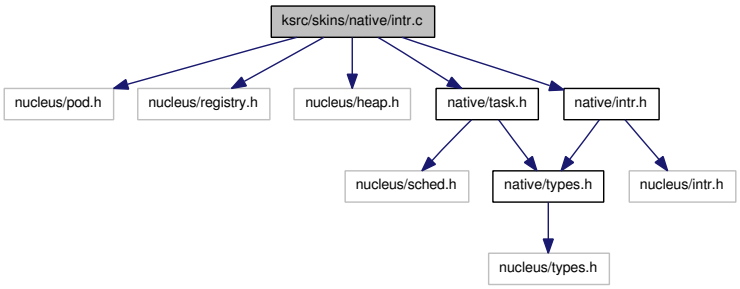
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.23 ksrc/skins/native/intr.c File Reference

This file is part of the Xenomai project. Include dependency graph for intr.c:



Functions

- int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq, rt_isr_t isr, rt_iack_t iack, int mode)
  Create an interrupt object from kernel space.
- int rt_intr_delete (RT_INTR *intr)
  Delete an interrupt object.
- int rt_intr_enable (RT_INTR *intr)
  Enable an interrupt object.
- int rt_intr_disable (RT_INTR *intr)
  Disable an interrupt object.
- int rt_intr_inquire (RT_INTR *intr, RT_INTR_INFO *info)
  Inquire about an interrupt object.

6.23.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2005 Philippe Gerum <rpm@xenomai.org>

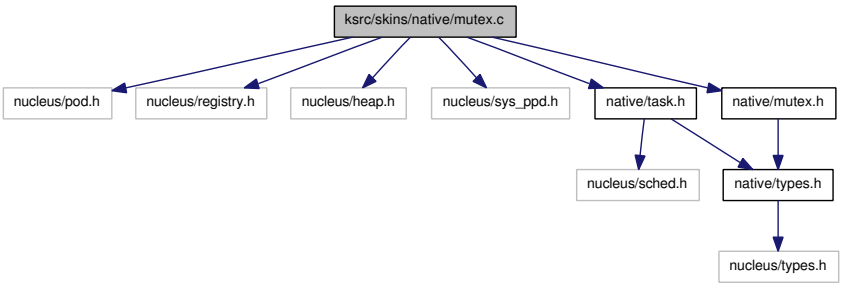
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.24 ksrc/skins/native/mutex.c File Reference

This file is part of the Xenomai project. Include dependency graph for mutex.c:



Functions

- int rt_mutex_create (RT_MUTEX *mutex, const char *name)
  Create a mutex.
- int rt_mutex_delete (RT_MUTEX *mutex)
  Delete a mutex.
- int rt_mutex_acquire (RT_MUTEX *mutex, RTIME timeout)
  Acquire a mutex.
- int rt_mutex_acquire_until (RT_MUTEX *mutex, RTIME timeout)
  Acquire a mutex (with absolute timeout date).
- int rt_mutex_release (RT_MUTEX *mutex)
  Unlock mutex.
- int rt_mutex_inquire (RT_MUTEX *mutex, RT_MUTEX_INFO *info)
  Inquire about a mutex.

6.24.1 Detailed Description

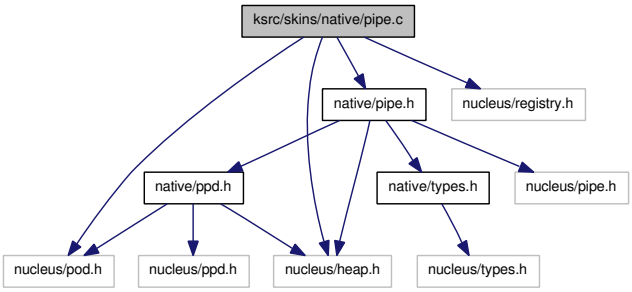
This file is part of the Xenomai project.

Note:

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.

6.25 ksrc/skins/native/pipe.c File Reference

This file is part of the Xenomai project. Include dependency graph for pipe.c:



Functions

- int rt_pipe_create (RT_PIPE *pipe, const char *name, int minor, size_t poolsize)
  Create a message pipe.
- int rt_pipe_delete (RT_PIPE *pipe)
  Delete a message pipe.
- ssize_t rt_pipe_receive (RT_PIPE *pipe, RT_PIPE_MSG **msgp, RTIME timeout)
  Receive a message from a pipe.
- ssize_t rt_pipe_read (RT_PIPE *pipe, void *buf, size_t size, RTIME timeout)
  Read a message from a pipe.
- ssize_t rt_pipe_send (RT_PIPE *pipe, RT_PIPE_MSG *msg, size_t size, int mode)
  Send a message through a pipe.
- ssize_t rt_pipe_write (RT_PIPE *pipe, const void *buf, size_t size, int mode)
  Write a message to a pipe.
- ssize_t rt_pipe_stream (RT_PIPE *pipe, const void *buf, size_t size)
  Stream bytes to a pipe.
- RT_PIPE_MSG * rt_pipe_alloc (RT_PIPE *pipe, size_t size)
  Allocate a message pipe buffer.
- int rt_pipe_free (RT_PIPE *pipe, RT_PIPE_MSG *msg)
  Free a message pipe buffer.
- int rt_pipe_flush (RT_PIPE *pipe, int mode)
  Flush the i/o queues associated with the kernel endpoint of a message pipe.
- int rt_pipe_monitor (RT_PIPE *pipe, int(*fn)(RT_PIPE *pipe, int event, long arg))
  Monitor a message pipe asynchronously.

6.25.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

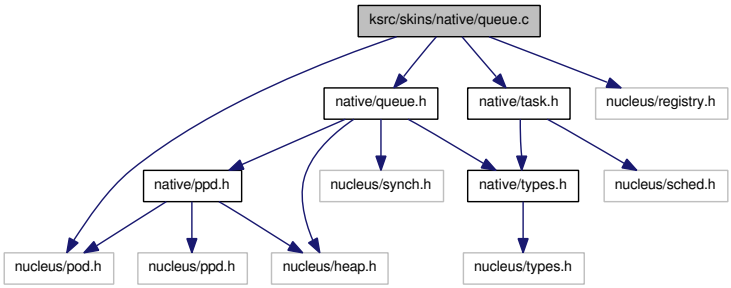
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.26 ksrc/skins/native/queue.c File Reference

This file is part of the Xenomai project. Include dependency graph for queue.c:



Functions

- int rt_queue_create (RT_QUEUE *q, const char *name, size_t poolsize, size_t qlimit, int mode)

  Create a message queue.

- int rt_queue_delete (RT_QUEUE *q)

  Delete a message queue.

- void * rt_queue_alloc (RT_QUEUE *q, size_t size)

  Allocate a message queue buffer.

- int rt_queue_free (RT_QUEUE *q, void *buf)

  Free a message queue buffer.

- int rt_queue_send (RT_QUEUE *q, void *mbuf, size_t size, int mode)

  Send a message to a queue.

- int rt_queue_write (RT_QUEUE *q, const void *buf, size_t size, int mode)

  Write a message to a queue.

- ssize_t rt_queue_receive (RT_QUEUE *q, void **bufp, RTIME timeout)

  Receive a message from a queue.

- ssize_t rt_queue_receive_until (RT_QUEUE *q, void **bufp, RTIME timeout)

  Receive a message from a queue (with absolute timeout date).

- ssize_t rt_queue_read (RT_QUEUE *q, void *buf, size_t size, RTIME timeout)

  Read a message from a queue.

- ssize_t rt_queue_read_until (RT_QUEUE *q, void *buf, size_t size, RTIME timeout)

  Read a message from a queue (with absolute timeout date).

- int rt_queue_flush (RT_QUEUE *q)

  Flush a message queue.

- int rt_queue_inquire (RT_QUEUE *q, RT_QUEUE_INFO *info)

  Inquire about a message queue.

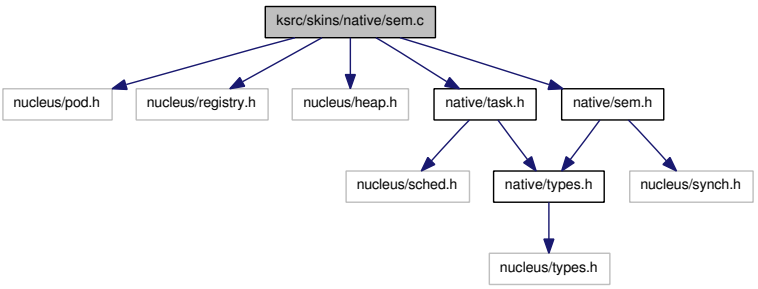
6.26.1 Detailed Description

This file is part of the Xenomai project.

Note:

6.27 ksrc/skins/native/sem.c File Reference

This file is part of the Xenomai project. Include dependency graph for sem.c:



Functions

- int rt_sem_create (RT_SEM *sem, const char *name, unsigned long icount, int mode)
  Create a counting semaphore.
- int rt_sem_delete (RT_SEM *sem)
  Delete a semaphore.
- int rt_sem_p (RT_SEM *sem, RTIME timeout)
  Pend on a semaphore.
- int rt_sem_p_until (RT_SEM *sem, RTIME timeout)
  Pend on a semaphore (with absolute timeout date).
- int rt_sem_v (RT_SEM *sem)
  Signal a semaphore.
- int rt_sem_broadcast (RT_SEM *sem)
  Broadcast a semaphore.
- int rt_sem_inquire (RT_SEM *sem, RT_SEM_INFO *info)
  Inquire about a semaphore.

6.27.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

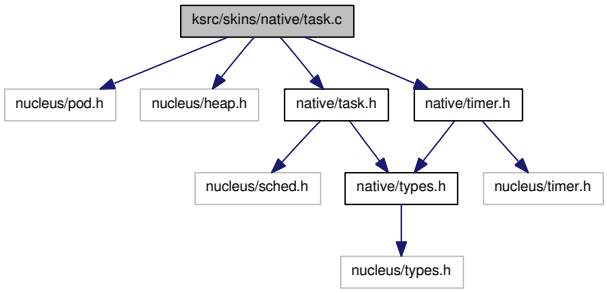
This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

6.28 ksrc/skins/native/task.c File Reference

This file is part of the Xenomai project. Include dependency graph for task.c:



Functions

- int rt_task_create (RT_TASK *task, const char *name, int stksize, int prio, int mode)
  Create a new real-time task.
- int rt_task_start (RT_TASK *task, void(*entry)(void *cookie), void *cookie)
  Start a real-time task.
- int rt_task_suspend (RT_TASK *task)
  Suspend a real-time task.
- int rt_task_resume (RT_TASK *task)
  Resume a real-time task.
- int rt_task_delete (RT_TASK *task)
  Delete a real-time task.
- int rt_task_yield (void)
  Manual round-robin.
- int rt_task_set_periodic (RT_TASK *task, RTIME idate, RTIME period)
  Make a real-time task periodic.
- int rt_task_wait_period (unsigned long *overruns_r)
  Wait for the next periodic release point.
- int rt_task_set_priority (RT_TASK *task, int prio)
  Change the base priority of a real-time task.
- int rt_task_sleep (RTIME delay)
  Delay the calling task (relative).
- int rt_task_sleep_until (RTIME date)
  Delay the calling task (absolute).

- int rt_task_unblock (RT_TASK *task)
  Unblock a real-time task.
- int rt_task_inquire (RT_TASK *task, RT_TASK_INFO *info)
  Inquire about a real-time task.
- int rt_task_add_hook (int type, void(*routine)(void *cookie))
  Install a task hook.
- int rt_task_remove_hook (int type, void(*routine)(void *cookie))
  Remove a task hook.
- int rt_task_catch (void(*handler)(rt_sigset_t))
  Install a signal handler.
- int rt_task_notify (RT_TASK *task, rt_sigset_t signals)
  Send signals to a task.
- int rt_task_set_mode (int clrmask, int setmask, int *mode_r)
  Change task mode bits.
- RT_TASK * rt_task_self (void)
  Retrieve the current task.
- int rt_task_slice (RT_TASK *task, RTIME quantum)
  Set a task's round-robin quantum.
- ssize_t rt_task_send (RT_TASK *task, RT_TASK_MCB *mcb_s, RT_TASK_MCB *mcb_r, RTIME timeout)
  Send a message to a task.
- int rt_task_receive (RT_TASK_MCB *mcb_r, RTIME timeout)
  Receive a message from a task.
- int rt_task_reply (int flowid, RT_TASK_MCB *mcb_s)
  Reply to a task.

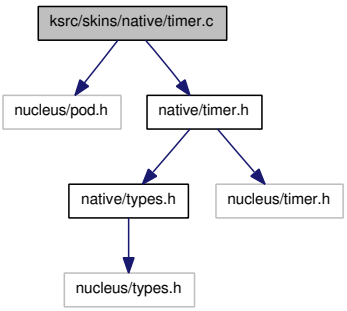
6.28.1 Detailed Description

This file is part of the Xenomai project.

Note:

6.29 ksrc/skins/native/timer.c File Reference

This file is part of the Xenomai project. Include dependency graph for timer.c:



Functions

- int rt_timer_inquire (RT_TIMER_INFO *info)
  Inquire about the timer.
- void rt_timer_spin (RTIME ns)
  Busy wait burning CPU cycles.
- int rt_timer_set_mode (RTIME nstick)
  Set the system clock rate.

6.29.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>

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.

Chapter 7

Example Documentation

7.1 bound_task.c

```
#include <sys/mman.h>
#include <native/task.h>

#define SIGNALS (0x1|0x4) /* Signals to send */

RT_TASK task_desc;

int main (int argc, char *argv[])

{
    int err;

    mlockall(MCL_CURRENT|MCL_FUTURE);

    /* Bind to a task which has been created elsewhere, either in
       kernel or user-space. The call will block us until such task is
       created with the expected name.  */

    err = rt_task_bind(&task_desc,"SomeTaskName",TM_NONBLOCK);

    if (!err)
        /* Send signals to the bound task */
        rt_task_notify(&task_desc,SIGNALS);

    /* ... */
}
```

7.2 cond_var.c

```
#include <native/mutex.h>
#include <native/cond.h>

RT_COND cond_desc;

RT_MUTEX mutex_desc;

int shared_event = 0;

void foo (void)

{
    int err;

    /* Create a condition variable and a mutex guarding it; we could
       also have attempted to bind to some pre-existing objects, using
       rt_cond_bind() and rt_mutex_bind() instead of creating them. */

    err = rt_mutex_create(&mutex_desc,"MyCondMutex");
    err = rt_cond_create(&cond_desc,"MyCondVar");

    /* Now, wait for some task to post the shared event... */

    rt_mutex_acquire(&mutex_desc,TM_INFINITE);

    while (!shared_event && !err)
        err = rt_cond_wait(&cond_desc,&mutex_desc,TM_INFINITE);

    rt_mutex_release(&mutex_desc);

    /* ... */
}

void bar (void)

{
    /* ... */

    /* Post the shared event. */

    rt_mutex_acquire(&mutex_desc,TM_INFINITE);

    shared_event = 1;
    rt_cond_signal(&cond_desc);

    rt_mutex_release(&mutex_desc);

    /* ... */
}

void cleanup (void)

{
    rt_cond_delete(&cond_desc);
    rt_mutex_delete(&mutex_desc);
}
```

7.3 event_flags.c

```
#include <native/event.h>

#define EVENT_INIT        0x0           /* No flags present at init */
#define EVENT_MODE        EV_PRIO       /* Tasks will wait by priority order */
#define EVENT_WAIT_MASK   (0x1|0x2|0x4) /* List of monitored events */
#define EVENT_SIGNAL_MASK (0x2)         /* List of events to send */

RT_EVENT ev_desc;

void foo (void)

{
    unsigned long mask_ret;
    int err;

    /* Create an event flag; we could also have attempted to bind to
       some pre-existing object, using rt_event_bind() instead of
       creating it. */

    err = rt_event_create(&ev_desc,
                          "MyEventFlagGroup",
                          EVENT_INIT,
                          EVENT_MODE);

    /* Now, wait for some task to post some event flags... */

    err = rt_event_wait(&ev_desc,
                        EVENT_WAIT_MASK,
                        &mask_ret,
                        EV_ANY, /* Disjunctive wait */
                        TM_INFINITE);
    /* ... */
}

void bar (void)

{
    /* ... */

    /* Post some events. */

    rt_event_signal(&ev_desc,EVENT_SIGNAL_MASK);

    /* ... */
}

void cleanup (void)

{
    rt_event_delete(&ev_desc);
}
```

7.4 kernel_task.c

```
#include <native/task.h>

#define TASK_PRIO  99              /* Highest RT priority */
#define TASK_MODE  T_FPU|T_CPU(0)  /* Uses FPU, bound to CPU #0 */
#define TASK_STKSZ 4096            /* Stack size (in bytes) */

RT_TASK task_desc;

void task_body (void *cookie)

{
    for (;;) {
    /* ... "cookie" should be NULL ... */
    }
}

int init_module (void)

{
    int err;

    /* ... */

    err = rt_task_create(&task_desc,
                         "MyTaskName",
                         TASK_STKSZ,
                         TASK_PRIO,
                         TASK_MODE);
    if (!err)
        rt_task_start(&task_desc,&task_body,NULL);

    /* ... */
}

void cleanup_module (void)

{
    rt_task_delete(&task_desc);
}
```

7.5 local_heap.c

```
#include <native/heap.h>

#define HEAP_SIZE (256*1024)
#define HEAP_MODE 0             /* Local heap. */

RT_HEAP heap_desc;

int init_module (void)

{
    void *block;
    int err;

    /* Create a 256Kb heap usable for dynamic memory allocation of
       variable-size blocks in kernel space. */

    err = rt_heap_create(&heap_desc,"MyHeapName",HEAP_SIZE,HEAP_MODE);

    if (err)
        fail();

    /* Request a 16-bytes block, asking for a non-blocking call since
       only Xenomai tasks may block. */
    err = rt_heap_alloc(&heap_desc,16,TM_NONBLOCK,&block);

    if (err)
        goto no_memory;

    /* Free the block: */
    rt_heap_free(&heap_desc,block);

    /* ... */
}

void cleanup_module (void)

{
    rt_heap_delete(&heap_desc);
}
```

7.6 msg_queue.c

```
#include <sys/mman.h>
#include <stdio.h>
#include <string.h>
#include <native/task.h>
#include <native/queue.h>

#define TASK_PRIO  99 /* Highest RT priority */
#define TASK_MODE  0  /* No flags */
#define TASK_STKSZ 0  /* Stack size (use default one) */

RT_QUEUE q_desc;

RT_TASK task_desc;

void consumer (void *cookie)

{
    ssize_t len;
    void *msg;
    int err;

    /* Bind to a queue which has been created elsewhere, either in
       kernel or user-space. The call will block us until such queue
       is created with the expected name. The queue should have been
       created with the Q_SHARED mode set, which is implicit when
       creation takes place in user-space. */

    err = rt_queue_bind(&q_desc,"SomeQueueName",TM_INFINITE);

    if (err)
        fail();

    /* Collect each message sent to the queue by the queuer() routine,
       until the queue is eventually removed from the system by a call
       to rt_queue_delete(). */

    while ((len = rt_queue_receive(&q_desc,&msg,TM_INFINITE)) > 0)
        {
        printf("received message> len=%d bytes, ptr=%p, s=%s\n",
               len,msg,(const char *)msg);
        rt_queue_free(&q_desc,msg);
        }

    /* We need to unbind explicitly from the queue in order to
       properly release the underlying memory mapping. Exiting the
       process unbinds all mappings automatically. */

    rt_queue_unbind(&q_desc);

    if (len != -EIDRM)
        /* We received some unexpected error notification. */
        fail();

    /* ... */
}

int main (int argc, char *argv[])

{
    static char *messages[] = { "hello", "world", NULL };
    int n, len;
    void *msg;

    mlockall(MCL_CURRENT|MCL_FUTURE);
```

```
    err = rt_task_create(&task_desc,
                         "MyTaskName",
                         TASK_STKSZ,
                         TASK_PRIO,
                         TASK_MODE);
    if (!err)
        rt_task_start(&task_desc,&task_body,NULL);

    /* ... */

    for (n = 0; messages[n] != NULL; n++)
        {
        len = strlen(messages[n]) + 1;
        /* Get a message block of the right size. */
        msg = rt_queue_alloc(&q_desc,len);

        if (!msg)
            /* No memory available. */
            fail();

        strcpy(msg,messages[n]);
        rt_queue_send(&q_desc,msg,len,Q_NORMAL);
        }

    rt_task_delete(&task_desc);
}
```

7.7 mutex.c

```
#include <native/mutex.h>

RT_MUTEX mutex_desc;

int main (int argc, char *argv[])

{
    int err;

    /* Create a mutex; we could also have attempted to bind to some
       pre-existing object, using rt_mutex_bind() and rt_mutex_bind()
       instead of creating it. In any case, priority inheritance is
       automatically enforced for mutual exclusion locks. */

    err = rt_mutex_create(&mutex_desc,"MyMutex");

    /* Now, grab the mutex lock, run the critical section, then
       release the lock: */

    rt_mutex_acquire(&mutex_desc,TM_INFINITE);

    /* ... Critical section ... */

    rt_mutex_release(&mutex_desc);

    /* ... */
}

void cleanup (void)

{
    rt_mutex_delete(&mutex_desc);
}
```

7.8 pipe.c

```
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <native/pipe.h>

#define PIPE_MINOR 0

/* User-space side */

int pipe_fd;

int main(int argc, char *argv[])
{
        char devname[32], buf[16];

        /* ... */

        sprintf(devname, "/dev/rtp%d", PIPE_MINOR);
        pipe_fd = open(devname, O_RDWR);

        if (pipe_fd < 0)
                fail();

        /* Wait for the prompt string "Hello"... */
        read(pipe_fd, buf, sizeof(buf));

        /* Then send the reply string "World": */
        write(pipe_fd, "World", sizeof("World"));

        /* ... */
}

void cleanup(void)
{
        close(pipe_fd);
}

/* Kernel-side */

#define TASK_PRIO  0            /* Highest RT priority */
#define TASK_MODE  T_FPU|T_CPU(0)       /* Uses FPU, bound to CPU #0 */
#define TASK_STKSZ 4096         /* Stack size (in bytes) */

RT_TASK task_desc;

RT_PIPE pipe_desc;

void task_body(void)
{
        RT_PIPE_MSG *msgout, *msgin;
        int err, len, n;

        for (;;) {
                /* ... */

                len = sizeof("Hello");
                /* Get a message block of the right size in order to
                   initiate the message-oriented dialog with the
                   user-space process. Sending a continuous stream of
                   bytes is also possible using rt_pipe_stream(), in
                   which case no message buffer needs to be
                   preallocated. */
                msgout = rt_pipe_alloc(len);
```

```
                if (!msgout)
                        fail();

                /* Send prompt message "Hello" (the output buffer will be freed
                   automatically)... */
                strcpy(RT_PIPE_MSGPTR(msgout), "Hello");
                rt_pipe_send(&pipe_desc, msgout, len, P_NORMAL);

                /* Then wait for the reply string "World": */
                n = rt_pipe_receive(&pipe_desc, &msgin, TM_INFINITE);

                if (n < 0) {
                        printf("receive error> errno=%d\n", n);
                        continue;
                }

                if (n == 0) {
                        if (msg == NULL) {
                                printf("pipe closed by peer while reading\n");
                                continue;
                        }

                        printf("empty message received\n");
                } else
                        printf("received msg> %s, size=%d\n", P_MSGPTR(msg),
                               P_MSGSIZE(msg));

                /* Free the received message buffer. */
                rt_pipe_free(&pipe_desc, msgin);

                /* ... */
        }
}

init init_module(void)
{
        int err;

        err = rt_pipe_create(&pipe_desc, NULL, PIPE_MINOR);

        if (err)
                fail();

        /* ... */

        err = rt_task_create(&task_desc,
                             "MyTaskName", TASK_STKSZ, TASK_PRIO, TASK_MODE);
        if (!err)
                rt_task_start(&task_desc, &task_body, NULL);

        /* ... */
}

void cleanup_module(void)
{
        rt_pipe_delete(&pipe_desc);
        rt_task_delete(&task_desc);
}
```

7.9 semaphore.c

```
#include <native/sem.h>

#define SEM_INIT 1       /* Initial semaphore count */
#define SEM_MODE S_FIFO  /* Wait by FIFO order */

RT_SEM sem_desc;

void foo (void)

{
    int err;

    /* Create a semaphore; we could also have attempted to bind to
       some pre-existing object, using rt_sem_bind() instead of
       creating it. */

    err = rt_sem_create(&sem_desc,"MySemaphore",SEM_INIT,SEM_MODE);

    for (;;) {

        /* Now, wait for a semaphore unit... */
        rt_sem_p(&sem_desc,TM_INFINITE);

        /* ... */

        /* then release it. */
        rt_sem_v(&sem_desc);

        /* ... */
    }
}

void cleanup (void)

{
    rt_sem_delete(&sem_desc);
}
```

7.10 shared_mem.c

```
#include <native/heap.h>

RT_HEAP heap_desc;

void *shared_mem; /* Start address of the shared memory segment */

/* A shared memory segment with Xenomai is implemented as a mappable
   real-time heap object managed as a single memory block. In this
   mode, the allocation routine always returns the start address of
   the heap memory to all callers, and the free routine always leads
   to a no-op. */

int main (int argc, char *argv[])

{
    int err;

    /* Bind to a shared heap which has been created elsewhere, either
       in kernel or user-space. Here we cannot wait and the heap must
       be available at once, since the caller is not a Xenomai-enabled
       thread. The heap should have been created with the H_SHARED
       mode set. */

    err = rt_heap_bind(&heap_desc,"SomeShmName",TM_NONBLOCK);

    if (err)
        fail();

    /* Get the address of the shared memory segment. The "size" and
       "timeout" arguments are unused here. */
    rt_heap_alloc(&heap_desc,0,TM_NONBLOCK,&shared_mem);

    /* ... */
}

void cleanup (void)

{
    /* We need to unbind explicitly from the heap in order to
       properly release the underlying memory mapping. Exiting the
       process unbinds all mappings automatically. */
    rt_heap_unbind(&heap_desc);
}
```

7.11 sigxcpu.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <getopt.h>
#include <execinfo.h>
#include <sys/mman.h>
#include <native/task.h>

RT_TASK task;

void task_body (void *cookie)

{
    /* Ask Xenomai to warn us upon switches to secondary mode. */
    rt_task_set_mode(0, T_WARNSW, NULL);

    /* A real-time task always starts in primary mode. */

    for (;;) {
        rt_task_sleep(1000000000);
        /* Running in primary mode... */
        printf("Switched to secondary mode\n");
        /* ...printf() => write(2): we have just switched to secondary
           mode: SIGXCPU should have been sent to us by now. */
    }
}

void warn_upon_switch(int sig __attribute__((unused)))

{
    void *bt[32];
    int nentries;

    /* Dump a backtrace of the frame which caused the switch to
       secondary mode: */
    nentries = backtrace(bt,sizeof(bt) / sizeof(bt[0]));
    backtrace_symbols_fd(bt,nentries,fileno(stdout));
}

int main (int argc, char **argv)

{
    int err;

    mlockall(MCL_CURRENT | MCL_FUTURE);

    signal(SIGXCPU, warn_upon_switch);

    err = rt_task_create(&task,"mytask",0,1,T_FPU);

    if (err)
        {
        fprintf(stderr,"failed to create task, code %d\n",err);
        return 0;
        }

    err = rt_task_start(&task,&task_body,NULL);

    if (err)
        {
        fprintf(stderr,"failed to start task, code %d\n",err);
        return 0;
        }
```

```
    pause();

    return 0;
}
```

7.12 trivial-periodic.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/timer.h>

RT_TASK demo_task;

/* NOTE: error handling omitted. */

void demo(void *arg)
{
        RTIME now, previous;

        /*
         * Arguments: &task (NULL=self),
         *            start time,
         *            period (here: 1 s)
         */
        rt_task_set_periodic(NULL, TM_NOW, 1000000000);
        previous = rt_timer_read();

        while (1) {
                rt_task_wait_period(NULL);
                now = rt_timer_read();

                /*
                 * NOTE: printf may have unexpected impact on the timing of
                 *       your program. It is used here in the critical loop
                 *       only for demonstration purposes.
                 */
                printf("Time since last turn: %ld.%06ld ms\n",
                       (long)(now - previous) / 1000000,
                       (long)(now - previous) % 1000000);
                       previous = now;
        }
}

void catch_signal(int sig)
{
}

int main(int argc, char* argv[])
{
        signal(SIGTERM, catch_signal);
        signal(SIGINT, catch_signal);

        /* Avoids memory swapping for this program */
        mlockall(MCL_CURRENT|MCL_FUTURE);

        /*
         * Arguments: &task,
         *            name,
         *            stack size (0=default),
         *            priority,
         *            mode (FPU, start suspended, ...)
         */
        rt_task_create(&demo_task, "trivial", 0, 99, 0);

        /*
         * Arguments: &task,
         *            task function,
         *            function argument
```

```
         */
        rt_task_start(&demo_task, &demo, NULL);

        pause();

        rt_task_delete(&demo_task);
}
```

7.13 user_alarm.c

```
#include <sys/mman.h>
#include <native/task.h>
#include <native/alarm.h>

#define TASK_PRIO  99 /* Highest RT priority */
#define TASK_MODE  0  /* No flags */
#define TASK_STKSZ 0  /* Stack size (use default one) */

#define ALARM_VALUE    500000   /* First shot at now + 500 us */
#define ALARM_INTERVAL 250000   /* Period is 250 us */

RT_ALARM alarm_desc;

RT_TASK server_desc;

void alarm_server (void *cookie)

{
    for (;;) {

       /* Wait for the next alarm to trigger. */
       err = rt_alarm_wait(&alarm_desc);

       if (!err) {
           /* Process the alarm shot. */
       }
    }
}

int main (int argc, char *argv[])

{
    int err;

    mlockall(MCL_CURRENT|MCL_FUTURE);

    /* ... */

    err = rt_alarm_create(&alarm_desc,"MyAlarm");

    err = rt_alarm_start(&alarm_desc,
                         ALARM_VALUE,
                         ALARM_INTERVAL);
    /* ... */

    err = rt_task_create(&server_desc,
                         "MyAlarmServer",
                         TASK_STKSZ,
                         TASK_PRIO,
                         TASK_MODE);
    if (!err)
        rt_task_start(&server_desc,&alarm_server,NULL);

    /* ... */
}

void cleanup (void)

{
    rt_alarm_delete(&alarm_desc);
    rt_task_delete(&server_desc);
}
```

7.14 user_irq.c

```
#include <sys/mman.h>
#include <native/task.h>
#include <native/intr.h>

#define IRQ_NUMBER 7  /* Intercept interrupt #7 */
#define TASK_PRIO  99 /* Highest RT priority */
#define TASK_MODE  0  /* No flags */
#define TASK_STKSZ 0  /* Stack size (use default one) */

RT_INTR intr_desc;

RT_TASK server_desc;

void irq_server (void *cookie)

{
    for (;;) {

       /* Wait for the next interrupt on channel #7. */
       err = rt_intr_wait(&intr_desc,TM_INFINITE);

       if (!err) {
           /* Process interrupt. */
       }
    }
}

int main (int argc, char *argv[])

{
    int err;

    mlockall(MCL_CURRENT|MCL_FUTURE);

    /* ... */

    err = rt_intr_create(&intr_desc,"MyIrq",IRQ_NUMBER,0);

    /* ... */

    err = rt_task_create(&server_desc,
                         "MyIrqServer",
                         TASK_STKSZ,
                         TASK_PRIO,
                         TASK_MODE);
    if (!err)
        rt_task_start(&server_desc,&irq_server,NULL);

    /* ... */
}

void cleanup (void)

{
    rt_intr_delete(&intr_desc);
    rt_task_delete(&server_desc);
}
```

7.15 user_task.c

```
#include <sys/mman.h>
#include <native/task.h>

#define TASK_PRIO  99 /* Highest RT priority */
#define TASK_MODE  0  /* No flags */
#define TASK_STKSZ 0  /* Stack size (use default one) */

RT_TASK task_desc;

void task_body (void *cookie)

{
    for (;;) {
    /* ... "cookie" should be NULL ... */
    }
}

int main (int argc, char *argv[])

{
    int err;

    mlockall(MCL_CURRENT|MCL_FUTURE);

    /* ... */

    err = rt_task_create(&task_desc,
                         "MyTaskName",
                         TASK_STKSZ,
                         TASK_PRIO,
                         TASK_MODE);
    if (!err)
        rt_task_start(&task_desc,&task_body,NULL);

    /* ... */
}

void cleanup (void)

{
    rt_task_delete(&task_desc);
}
```

Index