Xenomai nanokernel API Reference Manual

2.2.90

Generated by Doxygen 1.4.6

Sun Nov 5 18:53:29 2006

Contents

Chapter 1

Xenomai nanokernel API Module Index

1.1 Xenomai nanokernel API Modules

Here is a list of all modules:

Chapter 2

Xenomai nanokernel API Hierarchical Index

2.1 Xenomai nanokernel API Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Xenomai nanokernel API Data Structure Index

3.1 Xenomai nanokernel API Data Structures

Here are the data structures with brief descriptions:

Chapter 4

Xenomai nanokernel API File Index

4.1 Xenomai nanokernel API File List

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

Chapter 5

Xenomai nanokernel API Module Documentation

5.1 Status and Mode Defines.

5.1.1 Detailed Description

These defines are the used by the skins to report status of tasks and other objects.

Defines

- #define XNSUSP 0x00000001

  Suspended.

- #define XNPEND 0x00000002

  Sleep-wait for a resource.

- #define XNDELAY 0x00000004

  Delayed.

- #define XNREADY 0x00000008

  Linked to the ready queue.

- #define XNDORMANT 0x00000010

  Not started yet or killed.

- #define XNZOMBIE 0x00000020

  Zombie thread in deletion process.

- #define XNRESTART 0x00000040

  Restarting thread.

- #define XNSTARTED 0x00000080

  Could be restarted.

- #define XNRELAX 0x00000100
  Relaxed shadow thread (blocking bit).
- #define XNHELD 0x00000200
  Held thread from suspended partition.
- #define XNTIMEO 0x00000400
  Woken up due to a timeout condition.
- #define XNRMID 0x00000800
  Pending on a removed resource.
- #define XNBREAK 0x00001000
  Forcibly awaken from a wait state.
- #define XNKICKED 0x00002000
  Kicked upon Linux signal (shadow only).
- #define XNBOOST 0x00004000
  Undergoes regular PIP boost.
- #define XNDEBUG 0x00008000
  Hit debugger breakpoint (shadow only).
- #define XNLOCK 0x00010000
  Not preemptible.
- #define XNRRB 0x00020000
  Undergoes a round-robin scheduling.
- #define XNASDI 0x00040000
  ASR are disabled.
- #define XNSHIELD 0x00080000
  IRQ shield is enabled (shadow only).
- #define XNTRAPSW 0x00100000
  Trap execution mode switches.
- #define XNRPIOFF 0x00200000
  Stop priority coupling (shadow only).
- #define XNFPU 0x00400000
  Thread uses FPU.
- #define XNSHADOW 0x00800000
  Shadow thread.
- #define XNROOT 0x01000000

Root thread (that is, Linux/IDLE).

- #define XNINVPS 0x02000000

  Using inverted priority scale.

5.2 Dynamic memory allocation services.

5.2.1 Detailed Description

Dynamic memory allocation services.

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. You can find it at various locations on the net, including http://docs.FreeBSD.org/44doc/papers/kernmalloc.pdf. A minor variation allows this implementation to have 'extendable' heaps when needed, with multiple memory extents providing autonomous page address spaces.

The data structures hierarchy is as follows:

```
HEAP {
     block_buckets[]
     extent_queue -------+
}                        |
                         V
                      EXTENT #1 {
                             {static header}
                             page_map[npages]
                             page_array[npages][pagesize]
                      } -+
                         |
                         |
                         V
                      EXTENT #n {
                             {static header}
                             page_map[npages]
                             page_array[npages][pagesize]
                      }
```

Files

- file heap.c

  Dynamic memory allocation services.

Functions

- int xnheap_init (xnheap_t *heap, void *heapaddr, u_long heapsize, u_long pagesize)

  Initialize a memory heap.

- int xnheap_destroy (xnheap_t *heap, void(*flushfn)(xnheap_t *heap, void *extaddr, u_long extsize, void *cookie), void *cookie)

  Destroys a memory heap.

- void * xnheap_alloc (xnheap_t *heap, u_long size)

  Allocate a memory block from a memory heap.

- int xnheap_test_and_free (xnheap_t *heap, void *block, int(*ckfn)(void *block))

  Test and release a memory block to a memory heap.

- int xnheap_free (xnheap_t *heap, void *block)

  Release a memory block to a memory heap.

- int xnheap_extend (xnheap_t *heap, void *extaddr, u_long extsize)

  Extend a memory heap.

- void xnheap_schedule_free (xnheap_t *heap, void *block, xnholder_t *link)

  Schedule a memory block for release.

5.2.2 Function Documentation

5.2.2.1 void * xnheap_alloc (xnheap_t * *heap*, u_long *size*)

Allocate a memory block from a memory heap.

Allocates a contiguous region of memory from an active memory heap. Such allocation is guaranteed to be time-bounded.

Parameters:

heap The descriptor address of the heap to get memory from.

size The size in bytes of the requested block. Sizes lower or equal to the page size are rounded either to the minimum allocation size if lower than this value, or to the minimum alignment size if greater or equal to this value. In the current implementation, with MINALLOC = 8 and MINALIGN = 16, a 7 bytes request will be rounded to 8 bytes, and a 17 bytes request will be rounded to 32.

Returns:

The address of the allocated region upon success, or NULL if no memory is available from the specified heap.

Environments:

This service can be called from:

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

Rescheduling: never.

5.2.2.2 void xnheap_destroy (xnheap_t * *heap*, void(*)(xnheap_t *heap, void *extaddr, u_long extsize, void *cookie) *flushfn*, void * *cookie*)

Destroys a memory heap.

Destroys a memory heap.

Parameters:

heap The descriptor address of the destroyed heap.

flushfn If non-NULL, the address of a flush routine which will be called for each extent attached to the heap. This routine can be used by the calling code to further release the heap memory.

cookie If *flushfn* is non-NULL, *cookie* is an opaque pointer which will be passed unmodified to *flushfn*.

Returns:

0 is returned on success, or -EBUSY if external mappings are still pending on the heap memory.

Environments:

This service can be called from:

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

Rescheduling: never.

5.2.2.3 int xnheap_extend (xnheap_t * *heap*, void * *extaddr*, u_long *extsize*)

Extend a memory heap.

Add a new extent to an existing memory heap.

Parameters:

heap The descriptor address of the heap to add an extent to.

extaddr The address of the extent memory.

extsize The size of the extent memory (in bytes). In the current implementation, this size must match the one of the initial extent passed to xnheap_init().

Returns:

0 is returned upon success, or -EINVAL is returned if *extsize* differs from the initial extent's size.

Environments:

This service can be called from:

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

Rescheduling: never.

5.2.2.4 int xnheap_free (xnheap_t * *heap*, void * *block*)

Release a memory block to a memory heap.

Releases a memory region to the memory heap it was previously allocated from.

Parameters:

heap The descriptor address of the heap to release memory to.

block The address of the region to be returned to the heap.

Returns:

0 is returned upon success, or -EINVAL is returned whenever the block is not a valid region of the specified heap.

Environments:

This service can be called from:

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

Rescheduling: never.

5.2.2.5 xnheap_init (xnheap_t * *heap*, void * *heapaddr*, u_long *heapsize*, u_long *pagesize*)

Initialize a memory heap.

Initializes a memory heap suitable for time-bounded allocation requests of dynamic memory.

Parameters:

heap The address of a heap descriptor which will be used to store the allocation data. This descriptor must always be valid while the heap is active therefore it must be allocated in permanent memory.

heapaddr The address of the heap storage area. All allocations will be made from the given area in time-bounded mode. Since additional extents can be added to a heap, this parameter is also known as the "initial extent".

heapsize The size in bytes of the initial extent pointed at by *heapaddr*. *heapsize* must be a multiple of pagesize and lower than 16 Mbytes. *heapsize* must be large enough to contain an internal header. The following formula gives the size of this header:

hdrsize = (sizeof(xnextent_t) + ((heapsize - sizeof(xnextent_t))) / (pagesize + 1) + 15) & ~15.

pagesize The size in bytes of the fundamental memory page which will be used to subdivide the heap internally. Choosing the right page size is important regarding performance and memory fragmentation issues, so it might be a good idea to take a look at `http://docs.FreeBSD.org/44doc/papers/kernmalloc.pdf` to pick the best one for your needs. In the current implementation, pagesize must be a power of two in the range [8 .. 32768] inclusive.

Returns:

0 is returned upon success, or one of the following error codes:

- -EINVAL is returned whenever a parameter is invalid.

Environments:

This service can be called from:

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

Rescheduling: never.

5.2.2.6 int xnheap_schedule_free (xnheap_t * *heap*, void * *block*, xnholder_t * *link*)

Schedule a memory block for release.

This routine records a block for later release by xnheap_finalize_free(). This service is useful to lazily free blocks of heap memory when immediate release is not an option, e.g. when active references are still pending on the object for a short time after the call. xnheap_finalize_free() is expected to be eventually called by the client code at some point in the future when actually freeing the idle objects is deemed safe.

Parameters:

heap The descriptor address of the heap to release memory to.

block The address of the region to be returned to the heap.

link The address of a link member, likely but not necessarily within the released object, which will be used by the heap manager to hold the block in the queue of idle objects.

Environments:

This service can be called from:

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

Rescheduling: never.

5.2.2.7 int xnheap_test_and_free (xnheap_t * *heap*, void * *block*, int(*)(void *block) *ckfn*)

Test and release a memory block to a memory heap.

Releases a memory region to the memory heap it was previously allocated from. Before the actual release is performed, an optional user-defined can be invoked to check for additional criteria with respect to the request consistency.

Parameters:

heap The descriptor address of the heap to release memory to.

block The address of the region to be returned to the heap.

ckfn The address of a user-supplied verification routine which is to be called after the memory address specified by *block* has been checked for validity. The routine is expected to proceed to further consistency checks, and either return zero upon success, or non-zero upon error. In the latter case, the release process is aborted, and *ckfn's* return value is passed back to the caller of this service as its error return code. *ckfn* must not trigger the rescheduling procedure either directly or indirectly.

Returns:

0 is returned upon success, or -EINVAL is returned whenever the block is not a valid region of the specified heap. Additional return codes can also be defined locally by the *ckfn* routine.

Environments:

This service can be called from:

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

Rescheduling: never.

5.3 Interrupt management.

5.3.1 Detailed Description

Interrupt management.

Files

- file intr.c

  Interrupt management.

Functions

- int xnintr_init (xnintr_t *intr, const char *name, unsigned irq, xnisr_t isr, xniack_t iack, xnflags_t flags)

  Initialize an interrupt object.

- int xnintr_destroy (xnintr_t *intr)

  Destroy an interrupt object.

- int xnintr_attach (xnintr_t *intr, void *cookie)

  Attach an interrupt object.

- int xnintr_detach (xnintr_t *intr)

  Detach an interrupt object.

- int xnintr_enable (xnintr_t *intr)

  Enable an interrupt object.

- int xnintr_disable (xnintr_t *intr)

  Disable an interrupt object.

- xnarch_cpumask_t xnintr_affinity (xnintr_t *intr, xnarch_cpumask_t cpumask)

  Set interrupt's processor affinity.

5.3.2 Function Documentation

5.3.2.1 xnarch_cpumask_t xnintr_affinity (xnintr_t * *intr*, xnarch_cpumask_t *cpumask*)

Set interrupt's processor affinity.

Causes the IRQ associated with the interrupt object *intr* to be received only on processors which bits are set in *cpumask*.

Parameters:

intr The descriptor address of the interrupt object which affinity is to be changed.

cpumask The new processor affinity of the interrupt object.

Returns:
the previous cpumask on success, or an empty mask on failure.

Note:
Depending on architectures, setting more than one bit in *cpumask* could be meaningless.

5.3.2.2 int xnintr_attach (xnintr_t * *intr*, void * *cookie*)

Attach an interrupt object.

Attach an interrupt object previously initialized by xnintr_init(). After this operation is completed, all IRQs received from the corresponding interrupt channel are directed to the object's ISR.

Parameters:
- *intr* The descriptor address of the interrupt object to attach.
- *cookie* A user-defined opaque value which is stored into the interrupt object descriptor for further retrieval by the ISR/ISR handlers.

Returns:
0 is returned on success. Otherwise, -EINVAL is returned if a low-level error occurred while attaching the interrupt. -EBUSY is specifically returned if the interrupt object was already attached.

Environments:

This service can be called from:

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

Rescheduling: never.

Note:
Attaching an interrupt resets the tracked number of receipts to zero.

5.3.2.3 int xnintr_destroy (xnintr_t * *intr*)

Destroy an interrupt object.

Destroys an interrupt object previously initialized by xnintr_init(). The interrupt object is automatically detached by a call to xnintr_detach(). No more IRQs will be dispatched by this object after this service has returned.

Parameters:
- *intr* The descriptor address of the interrupt object to destroy.

Returns:
0 is returned on success. Otherwise, -EBUSY is returned if an error occurred while detaching the interrupt (see xnintr_detach()).

Environments:

This service can be called from:

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

Rescheduling: never.

5.3.2.4 int xnintr_detach (xnintr_t * *intr*)

Detach an interrupt object.

Detach an interrupt object previously attached by xnintr_attach(). After this operation is completed, no more IRQs are directed to the object's ISR, but the interrupt object itself remains valid. A detached interrupt object can be attached again by a subsequent call to xnintr_attach().

Parameters:

intr The descriptor address of the interrupt object to detach.

Returns:

0 is returned on success. Otherwise, -EINVAL is returned if a low-level error occurred while detaching the interrupt. Detaching a non-attached interrupt object leads to a null-effect and returns 0.

Environments:

This service can be called from:

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

Rescheduling: never.

5.3.2.5 int xnintr_disable (xnintr_t * *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.

Parameters:

intr The descriptor address of the interrupt object to disable.

Returns:

0 is returned on success. Otherwise, -EINVAL is returned if a low-level error occurred while disabling the interrupt.

Environments:

This service can be called from:

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

Rescheduling: never.

5.3.2.6 int xnintr_enable (xnintr_t * *intr*)

Enable an interrupt object.

Enables the hardware interrupt line associated with an interrupt object. Over real-time control layers which mask and acknowledge IRQs, this operation is necessary to revalidate the interrupt channel so that more interrupts can be notified.

Parameters:
intr The descriptor address of the interrupt object to enable.

Returns:
0 is returned on success. Otherwise, -EINVAL is returned if a low-level error occurred while enabling the interrupt.

Environments:

This service can be called from:

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

Rescheduling: never.

5.3.2.7 int xnintr_init (xnintr_t * *intr*, const char * *name*, unsigned *irq*, xnisr_t *isr*, xniack_t *iack*, xnflags_t *flags*)

Initialize an interrupt object.

Associates an interrupt object with an IRQ line.

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 service from the nucleus.

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:

- XN_ISR_HANDLED indicates that the interrupt request has been fulfilled by the ISR.

- XN_ISR_NONE indicates the opposite to XN_ISR_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.

- XN_ISR_PROPAGATE tells the nucleus 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 the nucleus and the host system.

- XN_ISR_NOENABLE causes the nucleus to ask the real-time control layer _not_ to re-enable the IRQ line (read the following section). xnarch_end_irq() must be called to re-enable the IRQ line later.

The nucleus re-enables the IRQ line by default. Over some real-time control layers which mask and acknowledge IRQs, this operation is necessary to revalidate the interrupt channel so that more interrupts can be notified.

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 the nucleus 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.

irq The hardware interrupt channel associated with the interrupt object. This value is architecture-dependent. An interrupt object must then be attached to the hardware interrupt vector using the xnintr_attach() service for the associated IRQs to be directed to this object.

isr The address of a valid low-level interrupt service routine if this parameter is non-zero. This handler will be called each time the corresponding IRQ is delivered on behalf of an interrupt context. When called, the ISR is passed the descriptor address of the interrupt object.

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.

flags A set of creation flags affecting the operation. The valid flags are:

- XN_ISR_SHARED enables IRQ-sharing with other interrupt objects.

- XN_ISR_EDGE is an additional flag need to be set together with XN_ISR_SHARED to enable IRQ-sharing of edge-triggered interrupts.

Returns:

No error condition being defined, 0 is always returned.

Environments:

This service can be called from:

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

Rescheduling: never.

5.4 Xenomai nucleus.

5.4.1 Detailed Description

An abstract RTOS core.

Modules

- Status and Mode Defines.

  These defines are the used by the skins to report status of tasks and other objects.

- Dynamic memory allocation services.
- Interrupt management.
- Real-time pod services.
- Registry services.
- Real-time shadow services.
- Thread synchronization services.
- Timer services.

5.5 Real-time pod services.

5.5.1 Detailed Description

Real-time pod services.

Files

- file pod.h

  Real-time pod interface header.

- file pod.c

  Real-time pod services.

Data Structures

- struct xnsched

  Scheduling information structure.

- struct xnpod

  Real-time pod descriptor.

Typedefs

- typedef xnsched xnsched_t

  Scheduling information structure.

Functions

- void xnpod_schedule_runnable (xnthread_t *thread, int flags)

  Hidden rescheduling procedure.

- int xnpod_init (xnpod_t *pod, int minpri, int maxpri, xnflags_t flags)

  Initialize a new pod.

- int xnpod_start_timer (u_long nstick, xnisr_t tickhandler)

  Start the system timer.

- void xnpod_stop_timer (void)

  Stop the system timer.

- int xnpod_reset_timer (void)

  Reset the system timer.

- void xnpod_shutdown (int xtype)

Shutdown the current pod.

- int xnpod_init_thread (xnthread_t *thread, const char *name, int prio, xnflags_t flags, unsigned stacksize)

  Initialize a new thread.

- int xnpod_start_thread (xnthread_t *thread, xnflags_t mode, int imask, xnarch_cpumask_t affinity, void(*entry)(void *cookie), void *cookie)

  Initial start of a newly created thread.

- void xnpod_restart_thread (xnthread_t *thread)

  Restart a thread.

- void xnpod_delete_thread (xnthread_t *thread)

  Delete a thread.

- xnflags_t xnpod_set_thread_mode (xnthread_t *thread, xnflags_t clrmask, xnflags_t setmask)

  Change a thread's control mode.

- void xnpod_resume_thread (xnthread_t *thread, xnflags_t mask)

  Resume a thread.

- int xnpod_unblock_thread (xnthread_t *thread)

  Unblock a thread.

- void xnpod_renice_thread (xnthread_t *thread, int prio)

  Change the base priority of a thread.

- int xnpod_migrate_thread (int cpu)

  Migrate the current thread.

- void xnpod_rotate_readyq (int prio)

  Rotate a priority level in the ready queue.

- void xnpod_schedule (void)

  Rescheduling procedure entry point.

- void xnpod_dispatch_signals (void)

  Deliver pending asynchronous signals to the running thread.

- void xnpod_activate_rr (xnticks_t quantum)

  Globally activate the round-robin scheduling.

- void xnpod_deactivate_rr (void)

  Globally deactivate the round-robin scheduling.

- void xnpod_set_time (xnticks_t newtime)

  Set the nucleus idea of time.

- int xnpod_set_thread_periodic (xnthread_t *thread, xnticks_t idate, xnticks_t period)

  Make a thread periodic.

- int xnpod_wait_thread_period (unsigned long *overruns_r)

  Wait for the next periodic release point.

- xnticks_t xnpod_get_time (void)

  Get the nucleus idea of time.

- int xnpod_add_hook (int type, void(*routine)(xnthread_t *))

  Install a nucleus hook.

- int xnpod_remove_hook (int type, void(*routine)(xnthread_t *))

  Remove a nucleus hook.

- void xnpod_suspend_thread (xnthread_t *thread, xnflags_t mask, xnticks_t timeout, xnsynch_t *wchan)

  Suspend a thread.

- void xnpod_welcome_thread (xnthread_t *thread, int imask)

  Thread prologue.

- static void xnpod_preempt_current_thread (xnsched_t *sched)

  Preempts the current thread.

- int xnpod_trap_fault (void *fltinfo)

  Default fault handler.

- int xnpod_announce_tick (xnintr_t *intr)

  Announce a new clock tick.

5.5.2 Function Documentation

5.5.2.1 void xnpod_activate_rr (xnticks_t *quantum*)

Globally activate the round-robin scheduling.

This service activates the round-robin scheduling for all threads which have the XNRRB flag set in their status mask (see xnpod_set_thread_mode()). Each of them will run for the given time quantum, then preempted and moved to the end of its priority group in the ready queue. This process is repeated until the round-robin scheduling is disabled for those threads.

Parameters:

quantum The time credit which will be given to each rr-enabled thread (in ticks).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

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

Rescheduling: never.

5.5.2.2 int xnpod_add_hook (int *type*, void(*)(xnthread_t *) *routine*)

Install a nucleus hook.

The nucleus 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:

- XNHOOK_THREAD_START: The user-defined routine will be called on behalf of the starter thread whenever a new thread starts. The descriptor address of the started thread is passed to the routine.

- XNHOOK_THREAD_DELETE: The user-defined routine will be called on behalf of the deletor thread whenever a thread is deleted. The descriptor address of the deleted thread is passed to the routine.

- XNHOOK_THREAD_SWITCH: The user-defined routine will be called on behalf of the resuming thread whenever a context switch takes place. The descriptor address of the thread which has been switched out is passed to the routine.

Parameters:

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

Returns:

0 is returned on 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
- Kernel-based task
- User-space task

Rescheduling: never.

5.5.2.3 int xnpod_announce_tick (xnintr_t * *intr*)

Announce a new clock tick.

This is the default service routine for clock ticks which performs the necessary housekeeping chores for time-related services managed by the nucleus. In a way or another, this routine must be called to announce each incoming clock tick to the nucleus.

Parameters:

intr The descriptor address of the interrupt object associated to the timer interrupt.

Side-effect: Since this routine manages the round-robin scheduling, the running thread (which has been preempted by the timer interrupt) can be switched out as a result of its time credit being exhausted. The nucleus always calls the rescheduling procedure after the outer interrupt has been processed.

Returns:

XN_ISR_HANDLED|XN_ISR_NOENABLE is always returned.

Environments:

This service can be called from:

- Interrupt service routine, must be called with interrupts off.

Rescheduling: possible.

5.5.2.4 void xnpod_deactivate_rr (void)

Globally deactivate the round-robin scheduling.

This service deactivates the round-robin scheduling for all threads which have the XNRRB flag set in their status mask (see xnpod_set_thread_mode()).

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.5 void xnpod_delete_thread (xnthread_t * *thread*)

Delete a thread.

Terminates a thread and releases all the nucleus resources it currently holds. A thread exists in the system since xnpod_init_thread() has been called to create it, so this service must be called in order to destroy it afterwards.

Parameters:

thread The descriptor address of the terminated thread.

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

Self-terminating a thread is allowed. In such a case, this service does not return to the caller.

Environments:

This service can be called from:

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

Rescheduling: possible if the current thread self-deletes.

5.5.2.6 void xnpod_dispatch_signals (void)

Deliver pending asynchronous signals to the running thread.

For internal use only.

This internal routine checks for the presence of asynchronous signals directed to the running thread, and attempts to start the asynchronous service routine (ASR) if any. Called with nklock locked, interrupts off.

5.5.2.7 xnticks_t xnpod_get_time (void)

Get the nucleus idea of time.

This service gets the nucleus (external) clock time.

Returns:

The current nucleus time (in ticks) if the underlying time source runs in periodic mode, or the system time (converted to nanoseconds) as maintained by the CPU if aperiodic mode is in effect, or no timer is running.

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.8 int xnpod_init (xnpod_t * *pod*, int *minpri*, int *maxpri*, xnflags_t *flags*)

Initialize a new pod.

Initializes a new pod which can subsequently be used to start real-time activities. Once a pod is active, real-time APIs can be stacked over. There can only be a single pod active in the host environment. Such environment can be confined to a process (e.g. simulator or UVM), or expand machine-wide (e.g. Adeos).

Parameters:

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

minpri The value of the lowest priority level which is valid for threads created on behalf of this pod.

maxpri The value of the highest priority level which is valid for threads created on behalf of this pod.

flags A set of creation flags affecting the operation. The only defined flag is XNREUSE, which tells the nucleus that a pre-existing pod exhibiting the same properties as the one which is being registered may be reused. In such a case, the call returns successfully, keeping the active pod unmodified.

minpri might be numerically higher than maxpri if the upper real-time interface exhibits a reverse priority scheme. For instance, some APIs may define a range like minpri=255, maxpri=0 specifying that thread priorities increase as the priority level decreases numerically.

Returns:

0 is returned on success. Otherwise:

- -EBUSY is returned if a pod already exists. As a special exception, if the Xenomai pod is currently loaded with no active attachment onto it, it is forcibly unloaded and replaced by the new pod.

- -ENOMEM is returned if the memory manager fails to initialize.

Environments:

This service can be called from:

- Kernel module initialization code

Note:

No initialization code called by this routine may refer to the global "nkpod" pointer.

5.5.2.9 int xnpod_init_thread (xnthread_t * *thread*, const char * *name*, int *prio*, xnflags_t *flags*, unsigned *stacksize*)

Initialize a new thread.

Initializes a new thread attached to the active pod. The thread is left in an innocuous state until it is actually started by xnpod_start_thread().

Parameters:

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

Warning:

Some architectures may require the descriptor to be properly aligned in memory; this is an additional reason for descriptors not to be laid in the program stack where alignement constraints might not always be satisfied.

Parameters:

name An ASCII string standing for the symbolic name of the thread. This name is copied to a safe place into the thread descriptor. This name might be used in various situations by the nucleus for issuing human-readable diagnostic messages, so it is usually a good idea to provide a sensible value here. The simulator even uses this name intensively to identify threads in the debugging GUI it provides. However, passing NULL here is always legal and means "anonymous".

prio The base priority of the new thread. This value must range from [minpri .. maxpri] (inclusive) as specified when calling the xnpod_init() service.

flags A set of creation flags affecting the operation. The following flags can be part of this bitmask, each of them affecting the nucleus behaviour regarding the created thread:

- XNSUSP creates the thread in a suspended state. In such a case, the thread will have to be explicitely resumed using the xnpod_resume_thread() service for its execution to actually begin, additionally to issuing xnpod_start_thread() for it. This flag can also be specified when invoking xnpod_start_thread() as a starting mode.

- XNFPU (enable FPU) tells the nucleus that the new thread will use the floating-point unit. In such a case, the nucleus will handle the FPU context save/restore ops upon thread switches at the expense of a few additional cycles per context switch. By default, a thread is not expected to use the FPU. This flag is simply ignored when the nucleus runs on behalf of a userspace-based real-time control layer since the FPU management is always active if present.

- XNINVPS tells the nucleus that the new thread will use an inverted priority scale with respect to the one enforced by the current pod. This means that the calling skin will still have to normalize the priority levels passed to the nucleus routines so that they conform to the pod's priority scale, but the nucleus will automatically rescale those values when displaying the priority information (e.g. /proc/xenomai/sched output). This bit must not be confused with the XNRPRIO bit, which is internally set by the nucleus during pod initialization when the low priority level is found to be numerically higher than the high priority bound. Having the XNINVPS bit set for a thread running on a pod with XNRPRIO unset means that the skin emulates a decreasing priority scale using the pod's increasing priority scale. This is typically the case for skins running over the core pod (see include/nucleus/core.h).

Parameters:

stacksize The size of the stack (in bytes) for the new thread. If zero is passed, the nucleus will use a reasonable pre-defined size depending on the underlying real-time control layer.

After creation, the new thread can be set a magic cookie by skins using xnthread_set_magic() to unambiguously identify threads created in their realm. This value will be copied as-is to the *magic* field of the thread struct. 0 is a conventional value for "no magic".

Returns:

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

- -EINVAL is returned if *flags* has invalid bits set.
- -ENOMEM is returned if not enough memory is available from the system heap to create the new thread's stack.

Side-effect: This routine does not call the rescheduling procedure.

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.10 int xnpod_migrate_thread (int *cpu*)

Migrate the current thread.

This call makes the current thread migrate to another CPU if its affinity allows it.

Parameters:

cpu The destination CPU.

Return values:

0 if the thread could migrate ;

-EPERM if the calling context is asynchronous, or the current thread affinity forbids this migration ;

-EBUSY if the scheduler is locked.

5.5.2.11 void xnpod_preempt_current_thread (xnsched_t * *sched*) `[inline, static]`

Preempts the current thread.

For internal use only.

Preempts the running thread (because a higher priority thread has just been readied). The thread is re-inserted to the front of its priority group in the ready thread queue. Must be called with nklock locked, interrupts off.

5.5.2.12 int xnpod_remove_hook (int *type*, void(*)(xnthread_t *) *routine*)

Remove a nucleus hook.

This service removes a nucleus hook previously registered using xnpod_add_hook().

Parameters:

type Defines the kind of hook to remove among XNHOOK_THREAD_START, XNHOOK_-THREAD_DELETE and XNHOOK_THREAD_SWITCH.

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

Returns:

0 is returned on success. Otherwise, -EINVAL is returned if type is incorrect or if the routine has never been registered before.

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.13 void xnpod_renice_thread (xnthread_t * *thread*, int *prio*)

Change the base priority of a thread.

Changes the base priority of a thread. If the reniced thread is currently blocked, waiting in priority-pending mode (XNSYNCH_PRIO) for a synchronization object to be signaled, the nucleus will attempt to reorder the object's wait queue so that it reflects the new sleeper's priority, unless the XNSYNCH_DREORD flag has been set for the pended object.

Parameters:

thread The descriptor address of the affected thread.

prio The new thread priority.

It is absolutely required to use this service to change a thread priority, in order to have all the needed housekeeping chores correctly performed. i.e. Do *not* change the thread.cprio field by hand, unless the thread is known to be in an innocuous state (e.g. dormant).

Side-effects:

- This service does not call the rescheduling procedure but may affect the ready queue.
- Assigning the same priority to a running or ready thread moves it to the end of the ready queue, thus causing a manual round-robin.
- If the reniced thread is a user-space shadow, propagate the request to the mated Linux task.

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.14 int xnpod_reset_timer (void)

Reset the system timer.

Reset the system timer to its default setup. The default setup data are obtained, by order of priority, from:

- the "tick_arg" module parameter when passed to the nucleus. Zero means aperiodic timing, any other value is used as the constant period to use for undergoing the periodic timing mode.
- or, the value of the CONFIG_XENO_OPT_TIMING_PERIOD configuration parameter if CONFIG_XENO_OPT_TIMING_PERIODIC is also set. If the latter is unset, the aperiodic mode will be used.

Returns:
0 is returned on success. Otherwise:

- -EBUSY is returned if the timer has already been set with incompatible requirements (different mode, different period if periodic, or non-default tick handler). xnpod_stop_timer() must be issued before xnpod_reset_timer() is called again.
- -ENODEV is returned if the underlying architecture does not support the requested periodic timing.
- -ENOSYS is returned if no active pod exists.

Side-effect: A host timing service is started in order to relay the canonical periodical tick to the underlying architecture, regardless of the frequency used for Xenomai's system tick. This routine does not call the rescheduling procedure.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task in secondary mode

Rescheduling: never.

5.5.2.15 void xnpod_restart_thread (xnthread_t * *thread*)

Restart a thread.

Restarts a previously started thread. The thread is first terminated then respawned using the same information that prevailed when it was first started, including the mode bits and interrupt mask initially passed to the xnpod_start_thread() service. As a consequence of this call, the thread entry point is rerun.

Parameters:

thread The descriptor address of the affected thread which must have been previously started by the xnpod_start_thread() service.

Self-restarting a thread is allowed. However, restarting the root thread is not.

Environments:

This service can be called from:

- Kernel-based task
- User-space task

Rescheduling: possible.

5.5.2.16 void xnpod_resume_thread (xnthread_t * *thread*, xnflags_t *mask*)

Resume a thread.

Resumes the execution of a thread previously suspended by one or more calls to xnpod_suspend_-thread(). This call removes a suspensive condition affecting the target thread. When all suspensive conditions are gone, the thread is left in a READY state at which point it becomes eligible anew for scheduling.

Parameters:

thread The descriptor address of the resumed thread.

mask The suspension mask specifying the suspensive condition to remove from the thread's wait mask. Possible values usable by the caller are:

- XNSUSP. This flag removes the explicit suspension condition. This condition might be additive to the XNPEND condition.
- XNDELAY. This flag removes the counted delay wait condition.
- XNPEND. This flag removes the resource wait condition. If a watchdog is armed, it is automatically disarmed by this call. Unlike the two previous conditions, only the current thread can set this condition for itself, i.e. no thread can force another one to pend on a resource.

When the thread is eventually resumed by one or more calls to xnpod_resume_thread(), the caller of xnpod_suspend_thread() in the awakened thread that suspended itself should check for the following bits in its own status mask to determine what caused its wake up:

- XNRMID means that the caller must assume that the pended synchronization object has been destroyed (see xnsynch_flush()).
- XNTIMEO means that the delay elapsed, or the watchdog went off before the corresponding synchronization object was signaled.
- XNBREAK means that the wait has been forcibly broken by a call to xnpod_unblock_thread().

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.17 void xnpod_rotate_readyq (int *prio*)

Rotate a priority level in the ready queue.

The thread at the head of the ready queue of the given priority level is moved to the end of this queue. Therefore, the execution of threads having the same priority is switched. Round-robin scheduling policies may be implemented by periodically issuing this call in a given period of time. It should be noted that the nucleus already provides a built-in round-robin mode though (see xnpod_activate_rr()).

Parameters:

prio The priority level to rotate. if XNPOD_RUNPRIO is given, the running thread priority is used to rotate the queue.

The priority level which is considered is always the base priority of a thread, not the possibly PIP-boosted current priority value. Specifying a priority level with no thread on it is harmless, and will simply lead to a null-effect.

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.18 void xnpod_schedule (void)

Rescheduling procedure entry point.

This is the central rescheduling routine which should be called to validate and apply changes which have previously been made to the nucleus scheduling state, such as suspending, resuming or changing the priority of threads. This call first determines if a thread switch should take place, and performs it as needed. xnpod_schedule() actually switches threads if:

- the running thread has been blocked or deleted.
- or, the running thread has a lower priority than the first ready to run thread.
- or, the running thread does not lead no more the ready threads (round-robin).

The nucleus implements a lazy rescheduling scheme so that most of the services affecting the threads state MUST be followed by a call to the rescheduling procedure for the new scheduling state to be applied. In other words, multiple changes on the scheduler state can be done in a row, waking threads up, blocking others, without being immediately translated into the corresponding context switches, like it would be necessary would it appear that a higher priority thread than the current one became runnable for instance. When all changes have been applied, the rescheduling procedure is then called to consider those changes, and possibly replace the current thread by another one.

As a notable exception to the previous principle however, every action which ends up suspending or deleting the current thread begets an immediate call to the rescheduling procedure on behalf of the service causing the state transition. For instance, self-suspension, self-destruction, or sleeping on a synchronization object automatically leads to a call to the rescheduling procedure, therefore the caller does not need to explicitely issue xnpod_schedule() after such operations.

The rescheduling procedure always leads to a null-effect if the scheduler is locked (XNLOCK bit set in the status mask of the running thread), or if it is called on behalf of an ISR or callout.

Calling this procedure with no applicable context switch pending is harmless and simply leads to a null-effect.

Side-effects:

- If an asynchronous service routine exists, the pending asynchronous signals are delivered to a resuming thread or on behalf of the caller before it returns from the procedure if no context switch has taken place. This behaviour can be disabled by setting the XNASDI flag in the thread's status mask by calling xnpod_set_thread_mode().

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine, although this leads to a no-op.
- Kernel-based task
- User-space task

Note:

The switch hooks are called on behalf of the resuming thread.

5.5.2.19 void xnpod_schedule_runnable (xnthread_t * *thread*, int *flags*)

Hidden rescheduling procedure.

For internal use only.

This internal routine should NEVER be used directly by the upper interfaces. It reinserts the given thread into the ready queue then switches to the highest priority runnable thread. It must be called with nklock locked, interrupts off.

Parameters:

thread The descriptor address of the thread to reinsert into the ready queue.

flags A bitmask composed as follows:

- XNPOD_SCHEDLIFO causes the target thread to be inserted at front of its priority group in the ready queue. Otherwise, the FIFO ordering is applied.
- XNPOD_NOSWITCH reorders the ready queue without switching contexts. This feature is used to preserve the atomicity of some operations.

5.5.2.20 xnflags_t xnpod_set_thread_mode (xnthread_t * *thread*, xnflags_t *clrmask*, xnflags_t *setmask*)

Change a thread's control mode.

Change the control mode of a given thread. The control mode affects the behaviour of the nucleus regarding the specified thread.

Parameters:

thread The descriptor address of the affected thread.

clrmask Clears the corresponding bits from the control field before setmask is applied. The scheduler lock held by the current thread can be forcibly released by passing the XNLOCK bit in this mask. In this case, the lock nesting count is also reset to zero.

setmask The new thread mode. The following flags can be part of this bitmask, each of them affecting the nucleus behaviour regarding the thread:

- XNLOCK causes the thread to lock the scheduler. The target thread will have to call the xnpod_unlock_sched() service to unlock the scheduler or clear the XNLOCK bit forcibly using this service.
- XNRRB causes the thread to be marked as undergoing the round-robin scheduling policy. The contents of the thread.rrperiod field determines the time quantum (in ticks) allowed for its next slice. If the thread is already undergoing the round-robin scheduling policy at the time this service is called, the time quantum remains unchanged.
- XNASDI disables the asynchronous signal handling for this thread. See xnpod_schedule() for more on this.
- XNSHIELD enables the interrupt shield for the current user-space task. When engaged, the interrupt shield protects the shadow task running in secondary mode from any preemption by the regular Linux interrupt handlers, without delaying in any way Xenomai's interrupt handling. The shield is operated on a per-task basis at each context switch, depending on the setting of this flag. This feature is only available if the CONFIG_XENO_OPT_ISHIELD option has been enabled at configuration time; otherwise, this flag is simply ignored.

- XNRPIOFF 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.

Environments:

This service can be called from:

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

Rescheduling: never, therefore, the caller should reschedule if XNLOCK has been passed into *clrmask*.

5.5.2.21 int xnpod_set_thread_periodic (xnthread_t * *thread*, xnticks_t *idate*, xnticks_t *period*)

Make a thread periodic.

Make a thread periodic by programming its first release point and its period in the processor time line. Subsequent calls to xnpod_wait_thread_period() will delay the thread until the next periodic release point in the processor timeline is reached.

Parameters:

thread The descriptor address of the affected thread. This thread is immediately delayed until the first periodic release point is reached.

idate The initial (absolute) date of the first release point, expressed in clock ticks (see note). The affected thread will be delayed until this point is reached. If *idate* is equal to XN_INFINITE, the current system date is used, and no initial delay takes place.

period The period of the thread, expressed in clock ticks (see note). As a side-effect, passing XN_INFINITE attempts to stop the thread'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:

- -ETIMEDOUT is returned *idate* is different from XN_INFINITE and represents a date in the past.
- -EWOULDBLOCK is returned if the system timer has not been started using xnpod_start_-timer().
- -EINVAL is returned if *period* is different from XN_INFINITE but shorter than the scheduling latency value for the target system, as available from /proc/xenomai/latency.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Kernel-based task
- User-space task

Rescheduling: possible if the operation affects the current thread and *idate* has not elapsed yet.

Note:

This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are interpreted as periodic jiffies. In oneshot mode, clock ticks are interpreted as nanoseconds.

5.5.2.22 void xnpod_set_time (xnticks_t *newtime*)

Set the nucleus idea of time.

The nucleus tracks the current time as a monotonously increasing count of ticks announced by the timer source since the epoch. The epoch is initially the same as the underlying architecture system time. This service changes the epoch. Running timers use a different time base thus are not affected by this operation.

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.23 void xnpod_shutdown (int *xtype*)

Shutdown the current pod.

Forcibly shutdowns the active pod. All existing nucleus threads (but the root one) are terminated, and the system heap is freed.

Parameters:

xtype An exit code passed to the host environment who started the nucleus. Zero is always interpreted as a successful return.

The nucleus never calls this routine directly. Skins should provide their own shutdown handlers which end up calling xnpod_shutdown() after their own housekeeping chores have been carried out.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

Rescheduling: never.

5.5.2.24 int xnpod_start_thread (xnthread_t * *thread*, xnflags_t *mode*, int *imask*, xnarch_cpumask_t *affinity*, void(*)(void *cookie) *entry*, void * *cookie*)

Initial start of a newly created thread.

Starts a (newly) created thread, scheduling it for the first time. This call releases the target thread from the XNDORMANT state. This service also sets the initial mode and interrupt mask for the new thread.

Parameters:

thread The descriptor address of the affected thread which must have been previously initialized by the xnpod_init_thread() service.

mode The initial thread mode. The following flags can be part of this bitmask, each of them affecting the nucleus behaviour regarding the started thread:

- XNLOCK causes the thread to lock the scheduler when it starts. The target thread will have to call the xnpod_unlock_sched() service to unlock the scheduler.
- XNRRB causes the thread to be marked as undergoing the round-robin scheduling policy at startup. The contents of the thread.rrperiod field determines the time quantum (in ticks) allowed for its next slice.
- XNASDI disables the asynchronous signal handling for this thread. See xnpod_schedule() for more on this.
- XNSUSP makes the thread start in a suspended state. In such a case, the thread will have to be explicitely resumed using the xnpod_resume_thread() service for its execution to actually begin.

Parameters:

imask The interrupt mask that should be asserted when the thread starts. The processor interrupt state will be set to the given value when the thread starts running. The interpretation of this value might be different across real-time layers, but a non-zero value should always mark an interrupt masking in effect (e.g. cli()). Conversely, a zero value should always mark a fully preemptible state regarding interrupts (i.e. sti()).

affinity The processor affinity of this thread. Passing XNPOD_ALL_CPUS or an empty affinity set means "any cpu".

entry The address of the thread's body routine. In other words, it is the thread entry point.

cookie A user-defined opaque cookie the nucleus will pass to the emerging thread as the sole argument of its entry point.

The START hooks are called on behalf of the calling context (if any).

Return values:

0 if *thread* could be started ;

-EBUSY if *thread* was already started ;

-EINVAL if the value of *affinity* is invalid.

Environments:

This service can be called from:

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

Rescheduling: possible.

5.5.2.25 int xnpod_start_timer (u_long *nstick*, xnisr_t *tickhandler*)

Start the system timer.

The nucleus needs a time source to provide the time-related services to the upper interfaces. xnpod_start_timer() tunes the timer hardware so that a user-defined routine is called according to a given frequency. On architectures that provide a oneshot-programmable time source, the system timer can operate either in aperiodic or periodic mode. Using the aperiodic mode still allows to run periodic timings over it: the underlying hardware will simply be reprogrammed after each tick by the timer manager using the appropriate interval value (see xntimer_start()).

The time interval that elapses between two consecutive invocations of the handler is called a tick.

Parameters:

nstick The timer period in nanoseconds. XNPOD_DEFAULT_TICK can be used to set this value according to the arch-dependent settings. If this parameter is equal to XN_APERIODIC_TICK, the underlying hardware timer is set to operate in oneshot-programming mode. In this mode, timing accuracy is higher - since it is not rounded to a constant time slice. The aperiodic mode gives better results in configuration involving threads requesting timing services over different time scales that cannot be easily expressed as multiples of a single base tick, or would lead to a waste of high frequency periodical ticks.

tickhandler The address of the tick handler which will process each incoming tick. XNPOD_-DEFAULT_TICKHANDLER can be passed to use the system-defined entry point (i.e. xnpod_announce_tick()). In any case, a user-supplied handler should end up calling xnpod_announce_tick() to inform the nucleus of the incoming tick.

Returns:

0 is returned on success. Otherwise:

- -EBUSY is returned if the timer has already been set with incompatible requirements (different mode, different period if periodic, or different handler). xnpod_stop_timer() must be issued before xnpod_start_timer() is called again.
- -EINVAL is returned if an invalid null tick handler has been passed, or if the timer precision cannot represent the duration of a single host tick.
- -ENODEV is returned if the underlying architecture does not support the requested periodic timing.
- -ENOSYS is returned if no active pod exists.

Side-effect: A host timing service is started in order to relay the canonical periodical tick to the underlying architecture, regardless of the frequency used for Xenomai's system tick. This routine does not call the rescheduling procedure.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task in secondary mode

Rescheduling: never.

5.5.2.26 void xnpod_stop_timer (void)

Stop the system timer.

Stops the system timer previously started by a call to xnpod_start_timer().

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- User-space task in secondary mode

Rescheduling: never.

5.5.2.27 void xnpod_suspend_thread (xnthread_t * *thread*, xnflags_t *mask*, xnticks_t *timeout*, xnsynch_t * *wchan*)

Suspend a thread.

Suspends the execution of a thread according to a given suspensive condition. This thread will not be eligible for scheduling until it all the pending suspensive conditions set by this service are removed by one or more calls to xnpod_resume_thread().

Parameters:

thread The descriptor address of the suspended thread.

mask The suspension mask specifying the suspensive condition to add to the thread's wait mask. Possible values usable by the caller are:

- XNSUSP. This flag forcibly suspends a thread, regardless of any resource to wait for. A reverse call to xnpod_resume_thread() specifying the XNSUSP bit must be issued to remove this condition, which is cumulative with other suspension bits.*wchan* should be NULL when using this suspending mode.
- XNDELAY. This flags denotes a counted delay wait (in ticks) which duration is defined by the value of the timeout parameter.
- XNPEND. This flag denotes a wait for a synchronization object to be signaled. The wchan argument must points to this object. A timeout value can be passed to bound the wait. This suspending mode should not be used directly by the upper interface, but rather through the xnsynch_sleep_on() call.

Parameters:

timeout The timeout which may be used to limit the time the thread pends for a resource. This value is a wait time given in ticks (see note). Passing XN_INFINITE specifies an unbounded wait. All other values are used to initialize a watchdog timer. If the current operation mode is oneshot and *timeout* elapses before xnpod_suspend_thread() has completed, then the target thread will not be suspended, and this routine leads to a null effect.

wchan The address of a pended resource. This parameter is used internally by the synchronization object implementation code to specify on which object the suspended thread pends. NULL is a legitimate value when this parameter does not apply to the current suspending mode (e.g. XNSUSP).

Note:

If the target thread is a shadow which has received a Linux-originated signal, then this service immediately exits without suspending the thread, but raises the XNBREAK condition in its status.

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 current thread suspends itself.

Note:

This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are interpreted as periodic jiffies. In oneshot mode, clock ticks are interpreted as nanoseconds.

5.5.2.28 void xnpod_trap_fault (void * *fltinfo*)

Default fault handler.

This is the default handler which is called whenever an uncontrolled exception or fault is caught. If the fault is caught on behalf of a real-time thread, the fault handler stored into the service table (svctable.faulthandler) is invoked and the fault is not propagated to the host system. Otherwise, the fault is unhandled by the nucleus and simply propagated.

Parameters:

fltinfo An opaque pointer to the arch-specific buffer describing the fault. The actual layout is defined by the xnarch_fltinfo_t type in each arch-dependent layer file.

5.5.2.29 int xnpod_unblock_thread (xnthread_t * *thread*)

Unblock a thread.

Breaks the thread out of any wait it is currently in. This call removes the XNDELAY and XNPEND suspensive conditions previously put by xnpod_suspend_thread() on the target thread. If all suspensive conditions are gone, the thread is left in a READY state at which point it becomes eligible anew for scheduling.

Parameters:

thread The descriptor address of the unblocked thread.

This call neither releases the thread from the XNSUSP, XNRELAX nor the XNDORMANT suspensive conditions.

When the thread resumes execution, the XNBREAK bit is set in the unblocked thread's status mask. Unblocking a non-blocked thread is perfectly harmless.

Returns:

non-zero is returned if the thread was actually unblocked from a pending wait state, 0 otherwise.

Environments:

This service can be called from:

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

Rescheduling: never.

5.5.2.30 int xnpod_wait_thread_period (unsigned long * *overruns_r*)

Wait for the next periodic release point.

Make the current thread 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 xnpod_wait_thread_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 xnpod_set_thread_periodic() has not previously been called for the calling thread.

- -EINTR is returned if xnpod_unblock_thread() has been called for the waiting thread before the next periodic release point has been reached. In this case, the overrun counter is reset too.

- -ETIMEDOUT is returned if the timer has overrun, which indicates that one or more previous release points have been missed by the calling thread. If *overruns_r* is valid, the count of pending overruns is copied to the pointed memory location.

Environments:

This service can be called from:

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

Rescheduling: always, unless the current release point has already been reached. In the latter case, the current thread immediately returns from this service without being delayed.

5.5.2.31 void xnpod_welcome_thread (xnthread_t * *thread*, int *imask*)

Thread prologue.

For internal use only.

This internal routine is called on behalf of a (re)starting thread's prologue before the user entry point is invoked. This call is reserved for internal housekeeping chores and cannot be inlined.

Entered with nklock locked, irqs off.

5.6 Registry services.

5.6.1 Detailed Description

The registry provides a mean to index real-time object descriptors created by Xenomai skins on unique alphanumeric keys. When labeled this way, a real-time object is globally exported; it can be searched for, and its descriptor returned to the caller for further use; the latter operation is called a "binding". When no object has been registered under the given name yet, the registry can be asked to set up a rendez-vous, blocking the caller until the object is eventually registered.

Files

- file registry.c

  This file is part of the Xenomai project.

Functions

- int xnregistry_enter (const char *key, void *objaddr, xnhandle_t *phandle, xnpnode_t *pnode)

  Register a real-time object.

- int xnregistry_bind (const char *key, xnticks_t timeout, xnhandle_t *phandle)

  Bind to a real-time object.

- int xnregistry_remove (xnhandle_t handle)

  Forcibly unregister a real-time object.

- int xnregistry_remove_safe (xnhandle_t handle, xnticks_t timeout)

  Unregister an idle real-time object.

- void * xnregistry_get (xnhandle_t handle)

  Find and lock a real-time object into the registry.

- u_long xnregistry_put (xnhandle_t handle)

  Unlock a real-time object from the registry.

- void * xnregistry_fetch (xnhandle_t handle)

  Find a real-time object into the registry.

5.6.2 Function Documentation

5.6.2.1 int xnregistry_bind (const char * *key*, xnticks_t *timeout*, xnhandle_t * *phandle*)

Bind to a real-time object.

This service retrieves the registry handle of a given object identified by its key. Unless otherwise specified, this service will block the caller if the object is not registered yet, waiting for such registration to occur.

Parameters:

key A valid NULL-terminated string which identifies the object to bind to.

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

phandle A pointer to a memory location which will be written upon success with the generic handle defined by the registry for the retrieved object. Contents of this memory is undefined upon failure.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *key* is NULL.
- -EINTR is returned if xnpod_unblock_thread() has been called for the waiting thread before the retrieval has completed.
- -EWOULDBLOCK is returned if *timeout* is equal to XN_NONBLOCK and the searched object is not registered on entry. As a special exception, this error is also returned if this service should block, but was called from a context which cannot sleep (e.g. interrupt, non-realtime or scheduler locked).
- -ETIMEDOUT is returned if the object cannot be retrieved within the specified amount of time.

Environments:

This service can be called from:

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

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

Note:

This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are interpreted as periodic jiffies. In oneshot mode, clock ticks are interpreted as nanoseconds.

5.6.2.2 int xnregistry_enter (const char * *key*, void * *objaddr*, xnhandle_t * *phandle*, xnpnode_t * *pnode*)

Register a real-time object.

This service allocates a new registry slot for an associated object, and indexes it by an alphanumeric key for later retrieval.

Parameters:

key A valid NULL-terminated string by which the object will be indexed and later retrieved in the registry. Since it is assumed that such key is stored into the registered object, it will *not* be copied but only kept by reference in the registry.

objaddr An opaque pointer to the object to index by *key*.

phandle A pointer to a generic handle defined by the registry which will uniquely identify the indexed object, until the latter is unregistered using the xnregistry_remove() service.

pnode A pointer to an optional /proc node class descriptor. This structure provides the information needed to export all objects from the given class through the /proc filesystem, under the /proc/xenomai/registry entry. Passing NULL indicates that no /proc support is available for the newly registered object.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *key* or *objaddr* are NULL, or if *key* constains an invalid '/' character.
- -ENOMEM is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the object.
- -EEXIST is returned if the *key* is already in use.

Environments:

This service can be called from:

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

Rescheduling: possible.

5.6.2.3 u_long xnregistry_fetch (xnhandle_t *handle*)

Find a real-time object into the registry.

This service retrieves an object from its handle into the registry and returns the memory address of its descriptor.

Parameters:

handle The generic handle of the object to fetch. If XNOBJECT_SELF is passed, the object is the calling Xenomai thread.

Returns:

The memory address of the object's descriptor is returned on success. Otherwise, NULL is returned if *handle* does not reference a registered object, or if *handle* is equal to XNOBJECT_-SELF but the current context is not a real-time thread.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *handle* is different from XNOBJECT_SELF.
- Kernel-based thread

Rescheduling: never.

5.6.2.4 void * xnregistry_get (xnhandle_t *handle*)

Find and lock a real-time object into the registry.

This service retrieves an object from its handle into the registry and prevents it removal atomically. A locking count is tracked, so that xnregistry_get() and xnregistry_put() must be used in pair.

Parameters:

handle The generic handle of the object to find and lock. If XNOBJECT_SELF is passed, the object is the calling Xenomai thread.

Returns:

The memory address of the object's descriptor is returned on success. Otherwise, NULL is returned if *handle* does not reference a registered object, or if *handle* is equal to XNOBJECT_SELF but the current context is not a real-time thread.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *handle* is different from XNOBJECT_SELF.
- Kernel-based thread.

Rescheduling: never.

5.6.2.5 u_long xnregistry_put (xnhandle_t *handle*)

Unlock a real-time object from the registry.

This service decrements the lock count of a registered object previously locked by a call to xnregistry_get(). The object is actually unlocked from the registry when the locking count falls down to zero, thus waking up any thread currently blocked on xnregistry_remove() for unregistering it.

Parameters:

handle The generic handle of the object to unlock. If XNOBJECT_SELF is passed, the object is the calling Xenomai thread.

Returns:

The decremented lock count is returned upon success. Zero is also returned if *handle* does not reference a registered object, or if *handle* is equal to XNOBJECT_SELF but the current context is not a real-time thread.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code
- Interrupt service routine only if *handle* is different from XNOBJECT_SELF.
- Kernel-based thread

Rescheduling: possible if the lock count falls down to zero and some thread is currently waiting for the object to be unlocked.

5.6.2.6 int xnregistry_remove (xnhandle_t *handle*)

Forcibly unregister a real-time object.

This service forcibly removes an object from the registry. The removal is performed regardless of the current object's locking status.

Parameters:

handle The generic handle of the object to remove.

Returns:

0 is returned upon success. Otherwise:

- -ESRCH is returned if *handle* does not reference a registered object.

Environments:

This service can be called from:

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

Rescheduling: never.

5.6.2.7 int xnregistry_remove_safe (xnhandle_t *handle*, xnticks_t *timeout*)

Unregister an idle real-time object.

This service removes an object from the registry. The caller might sleep as a result of waiting for the target object to be unlocked prior to the removal (see xnregistry_put()).

Parameters:

handle The generic handle of the object to remove.

timeout If the object is locked on entry, *param* gives the number of clock ticks to wait for the unlocking to occur (see note). Passing XN_INFINITE causes the caller to block indefinitely until the object is unlocked. Passing XN_NONBLOCK causes the service to return immediately without waiting if the object is locked on entry.

Returns:

0 is returned upon success. Otherwise:

- -ESRCH is returned if *handle* does not reference a registered object.
- -EWOULDBLOCK is returned if *timeout* is equal to XN_NONBLOCK and the object is locked on entry.
- -EBUSY is returned if *handle* refers to a locked object and the caller could not sleep until it is unlocked.
- -ETIMEDOUT is returned if the object cannot be removed within the specified amount of time.
- -EINTR is returned if xnpod_unblock_thread() has been called for the calling thread waiting for the object to be unlocked.

Environments:

This service can be called from:

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

Rescheduling: possible if the object to remove is currently locked and the calling context can sleep.

Note:

This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are interpreted as periodic jiffies. In oneshot mode, clock ticks are interpreted as nanoseconds.

5.7 Real-time shadow services.

5.7.1 Detailed Description

Real-time shadow services.

Files

- file shadow.c

  Real-time shadow services.

Functions

- int xnshadow_harden (void)

  Migrate a Linux task to the Xenomai domain.

- void xnshadow_relax (int notify)

  Switch a shadow thread back to the Linux domain.

- int xnshadow_map (xnthread_t *thread, xncompletion_t __user *u_completion)

  Create a shadow thread context.

- xnshadow_ppd_t * xnshadow_ppd_get (unsigned muxid)

  Return the per-process data attached to the calling process.

5.7.2 Function Documentation

5.7.2.1 int xnshadow_harden (void)

Migrate a Linux task to the Xenomai domain.

For internal use only.

This service causes the transition of "current" from the Linux domain to Xenomai. This is obtained by asking the gatekeeper to resume the shadow mated with "current" then triggering the rescheduling procedure in the Xenomai domain. The shadow will resume in the Xenomai domain as returning from schedule().

Environments:

This service can be called from:

- User-space thread operating in secondary (i.e. relaxed) mode.

Rescheduling: always.

5.7.2.2 int xnshadow_map (xnthread_t * *thread*, xncompletion_t __user * *u_completion*)

Create a shadow thread context.

For internal use only.

This call maps a nucleus thread to the "current" Linux task. The priority of the Linux task is set to the priority of the shadow thread bounded to the [0..MAX_RT_PRIO-1] range, and its scheduling policy is set to either SCHED_FIFO for non-zero priority levels, or SCHED_NORMAL otherwise.

Parameters:

thread The descriptor address of the new shadow thread to be mapped to "current". This descriptor must have been previously initialized by a call to xnpod_init_thread().

Warning:

The thread must have been set the same magic number (i.e. xnthread_set_magic()) than the one used to register the skin it belongs to. Failing to do so might lead to unexpected results.

Parameters:

u_completion is the address of an optional completion descriptor aimed at synchronizing our parent thread with us. If non-NULL, the information xnshadow_map() will store into the completion block will be later used to wake up the parent thread when the current shadow has been initialized. In the latter case, the new shadow thread is left in a dormant state (XNDORMANT) after its creation, leading to the suspension of "current" in the Linux domain, only processing signals. Otherwise, the shadow thread is immediately started and "current" immediately resumes in the Xenomai domain from this service.

Returns:

0 is returned on success. Otherwise:

- -ERESTARTSYS is returned if the current Linux task has received a signal, thus preventing the final migration to the Xenomai domain (i.e. in order to process the signal in the Linux domain). This error should not be considered as fatal.
- -EPERM is returned if the shadow thread has been killed before the current task had a chance to return to the caller. In such a case, the real-time mapping operation has failed globally, and no Xenomai resource remains attached to it.

Environments:
This service can be called from:

- Regular user-space process.

Rescheduling: always.

5.7.2.3 xnshadow_ppd_t* xnshadow_ppd_get (unsigned *muxid*)

Return the per-process data attached to the calling process.

This service returns the per-process data attached to the calling process for the skin whose muxid is *muxid*. It must be called with nklock locked, irqs off.

See xnshadow_register_interface() documentation for information on the way to attach a per-process data to a process.

Parameters:

muxid the skin muxid.

Returns:

the per-process data if the current context is a user-space process;
NULL otherwise.

5.7.2.4 void xnshadow_relax (int *notify*)

Switch a shadow thread back to the Linux domain.

For internal use only.

This service yields the control of the running shadow back to Linux. This is obtained by suspending the shadow and scheduling a wake up call for the mated user task inside the Linux domain. The Linux task will resume on return from xnpod_suspend_thread() on behalf of the root thread.

Parameters:

notify A boolean flag indicating whether threads monitored from secondary mode switches should be sent a SIGXCPU signal. For instance, some internal operations like task exit should not trigger such signal.

Environments:
This service can be called from:

- User-space thread operating in primary (i.e. harden) mode.

Rescheduling: always.

Note:

"current" is valid here since the shadow runs with the properties of the Linux task.

5.8 Thread synchronization services.

5.8.1 Detailed Description

Thread synchronization services.

Files

- file synch.c

  Thread synchronization services.

Functions

- void xnsynch_init (xnsynch_t *synch, xnflags_t flags)

  Initialize a synchronization object.

- void xnsynch_sleep_on (xnsynch_t *synch, xnticks_t timeout)

  Sleep on a synchronization object.

- static void xnsynch_clear_boost (xnsynch_t *synch, xnthread_t *lastowner)

  Clear the priority boost.

- void xnsynch_renice_sleeper (xnthread_t *thread)

  Change a sleeper's priority.

- xnthread_t * xnsynch_wakeup_one_sleeper (xnsynch_t *synch)

  Give the resource ownership to the next waiting thread.

- xnpholder_t * xnsynch_wakeup_this_sleeper (xnsynch_t *synch, xnpholder_t *holder)

  Give the resource ownership to a given waiting thread.

- int xnsynch_flush (xnsynch_t *synch, xnflags_t reason)

  Unblock all waiters pending on a resource.

- void xnsynch_forget_sleeper (xnthread_t *thread)

  Abort a wait for a resource.

- void xnsynch_release_all_ownerships (xnthread_t *thread)

  Release all ownerships.

5.8.2 Function Documentation

5.8.2.1 void xnsynch_clear_boost (xnsynch_t * *synch*, xnthread_t * *owner*) `[static]`

Clear the priority boost.

For internal use only.

This service is called internally whenever a synchronization object is not claimed anymore by sleepers to reset the object owner's priority to its initial level.

Parameters:

synch The descriptor address of the synchronization object.

owner The descriptor address of the thread which currently owns the synchronization object.

Note:

This routine must be entered nklock locked, interrupts off.

5.8.2.2 void xnsynch_flush (xnsynch_t * *synch*, xnflags_t *reason*)

Unblock all waiters pending on a resource.

This service atomically releases all threads which currently sleep on a given resource.

This service should be called by upper interfaces under circumstances requiring that the pending queue of a given resource is cleared, such as before the resource is deleted.

Parameters:

synch The descriptor address of the synchronization object to be flushed.

reason Some flags to set in the status mask of every unblocked thread. Zero is an acceptable value. The following bits are pre-defined by the nucleus:

- XNRMID should be set to indicate that the synchronization object is about to be destroyed (see xnpod_resume_thread()).
- XNBREAK should be set to indicate that the wait has been forcibly interrupted (see xnpod_-unblock_thread()).

Returns:

XNSYNCH_RESCHED is returned if at least one thread is unblocked, which means the caller should invoke xnpod_schedule() for applying the new scheduling state. Otherwise, XNSYNCH_DONE is returned.

Side-effects:

- The effective priority of the previous resource owner might be lowered to its base priority value as a consequence of the priority inheritance boost being cleared.
- The synchronization object is no more owned by any thread.

Environments:

This service can be called from:

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

Rescheduling: never.

5.8.2.3 void xnsynch_forget_sleeper (xnthread_t * *thread*)

Abort a wait for a resource.

For internal use only.

Performs all the necessary housekeeping chores to stop a thread from waiting on a given synchronization object.

Parameters:

thread The descriptor address of the affected thread.

When the trace support is enabled (i.e. MVM), the idle state is posted to the synchronization object's state diagram (if any) whenever no thread remains blocked on it. The real-time interfaces must ensure that such condition (i.e. EMPTY/IDLE) is mapped to state #0.

Note:

This routine must be entered nklock locked, interrupts off.

5.8.2.4 void xnsynch_init (xnsynch_t * *synch*, xnflags_t *flags*)

Initialize a synchronization object.

Initializes a new specialized object which can subsequently be used to synchronize real-time activities. The Xenomai nucleus provides a basic synchronization object which can be used to build higher resource objects. Nucleus threads can wait for and signal such objects in order to synchronize their activities.

This object has built-in support for priority inheritance.

Parameters:

synch The address of a synchronization object descriptor the nucleus 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.

flags A set of creation flags affecting the operation. The valid flags are:

- XNSYNCH_PRIO causes the threads waiting for the resource to pend in priority order. Otherwise, FIFO ordering is used (XNSYNCH_FIFO).

- XNSYNCH_PIP causes the priority inheritance mechanism to be automatically activated when a priority inversion is detected among threads using this object. Otherwise, no priority inheritance takes place upon priority inversion (XNSYNCH_NOPIP).

- XNSYNCH_DREORD (Disable REORDering) tells the nucleus that the wait queue should not be reordered whenever the priority of a blocked thread it holds is changed. If this flag is not specified, changing the priority of a blocked thread using xnpod_renice_thread() will cause this object's wait queue to be reordered according to the new priority level, provided the synchronization object makes the waiters wait by priority order on the awaited resource (XNSYNCH_PRIO).

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Kernel-based task
- User-space task

Rescheduling: never.

5.8.2.5 void xnsynch_release_all_ownerships (xnthread_t * *thread*)

Release all ownerships.

For internal use only.
This call is used internally to release all the ownerships obtained by a thread on synchronization objects. This routine must be entered interrupts off.

Parameters:
thread The descriptor address of the affected thread.

Note:
This routine must be entered nklock locked, interrupts off.

5.8.2.6 void xnsynch_renice_sleeper (xnthread_t * *thread*)

Change a sleeper's priority.

For internal use only.
This service is used by the PIP code to update the pending priority of a sleeping thread.

Parameters:
thread The descriptor address of the affected thread.

Note:
This routine must be entered nklock locked, interrupts off.

5.8.2.7 void xnsynch_sleep_on (xnsynch_t * *synch*, xnticks_t *timeout*)

Sleep on a synchronization object.

Makes the calling thread sleep on the specified synchronization object, waiting for it to be signaled.

This service should be called by upper interfaces wanting the current thread to pend on the given resource.

Parameters:
synch The descriptor address of the synchronization object to sleep on.
timeout The timeout which may be used to limit the time the thread pends on the resource. This value is a count of ticks (see note). Passing XN_INFINITE specifies an unbounded wait. All other values are used to initialize a nucleus watchdog timer.

Environments:

This service can be called from:

- Kernel module initialization/cleanup code

- Kernel-based task
- User-space task

Rescheduling: always.

Note:
This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are interpreted as periodic jiffies. In oneshot mode, clock ticks are interpreted as nanoseconds.

5.8.2.8 xnthread_t * xnsynch_wakeup_one_sleeper (xnsynch_t * *synch*)

Give the resource ownership to the next waiting thread.

This service gives the ownership of a synchronization object to the thread which is currently leading the object's pending list. The sleeping thread is unblocked, but no action is taken regarding the previous owner of the resource.

This service should be called by upper interfaces wanting to signal the given resource so that a single waiter is resumed.

Parameters:
synch The descriptor address of the synchronization object whose ownership is changed.

Returns:
The descriptor address of the unblocked thread.

Side-effects:

- The effective priority of the previous resource owner might be lowered to its base priority value as a consequence of the priority inheritance boost being cleared.
- The synchronization object ownership is transfered to the unblocked thread.

Environments:

This service can be called from:

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

Rescheduling: never.

5.8.2.9 void xnsynch_wakeup_this_sleeper (xnsynch_t * *synch*, xnpholder_t * *holder*)

Give the resource ownership to a given waiting thread.

This service gives the ownership of a given synchronization object to a specific thread which is currently pending on it. The sleeping thread is unblocked from its pending state. No action is taken regarding the previous resource owner.

This service should be called by upper interfaces wanting to signal the given resource so that a specific waiter is resumed.

Parameters:

synch The descriptor address of the synchronization object whose ownership is changed.

holder The link holder address of the thread to unblock (&thread->plink) which MUST be currently linked to the synchronization object's pending queue (i.e. synch->pendq).

Returns:

The link address of the unblocked thread in the synchronization object's pending queue.

Side-effects:

- The effective priority of the previous resource owner might be lowered to its base priority value as a consequence of the priority inheritance boost being cleared.

- The synchronization object ownership is transfered to the unblocked thread.

Environments:

This service can be called from:

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

Rescheduling: never.

5.9 Timer services.

5.9.1 Detailed Description

The Xenomai timer facility behaves slightly differently depending on the underlying system timer mode, i.e. periodic or aperiodic.

In periodic mode, the hardware timer ticks periodically without any external programming (aside of the initial one which sets its period).

If the underlying timer source is aperiodic, we need to reprogram the next shot after each tick at hardware level, and we do not need any periodic source.

Depending on the above mode, the timer object stores time values either as count of periodic ticks, or as count of CPU ticks.

Files

- file timer.c

Functions

- static void xntimer_do_tick_aperiodic (void)

  Process a timer tick in aperiodic mode.

- void xntimer_init (xntimer_t *timer, void(*handler)(xntimer_t *timer))

  Initialize a timer object.

- void xntimer_destroy (xntimer_t *timer)

  Release a timer object.

- xnticks_t xntimer_get_date (xntimer_t *timer)

  Return the absolute expiration date.

- xnticks_t xntimer_get_timeout (xntimer_t *timer)

  Return the relative expiration date.

- xnticks_t xntimer_get_interval (xntimer_t *timer)

  Return the timer interval value.

- void xntimer_freeze (void)

  Freeze all timers.

5.9.2 Function Documentation

5.9.2.1 void xntimer_destroy (xntimer_t * *timer*)

Release a timer object.

Destroys a timer. After it has been destroyed, all resources associated with the timer have been released. The timer is automatically deactivated before deletion if active on entry.

Parameters:
timer The address of a valid timer descriptor.

Environments:

This service can be called from:

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

Rescheduling: never.

5.9.2.2 void xntimer_do_tick_aperiodic (void) [static]

Process a timer tick in aperiodic mode.

For internal use only.
This routine informs all active timers that the clock has been updated by processing the outstanding timer list. Elapsed timer actions will be fired.
Environments:
This service can be called from:

- Interrupt service routine, nklock locked, interrupts off

Rescheduling: never.

Note:
Only active timers are inserted into the timer wheel.

5.9.2.3 void xntimer_freeze (void)

Freeze all timers.

For internal use only.
This routine deactivates all active timers atomically.
Environments:
This service can be called from:

- Interrupt service routine, nklock unlocked

Rescheduling: never.

Note:
Always make sure the nklock is free when stopping the underlying timing source by calling xnarch_stop_timer(), otherwise, deadlock situations would arise on some architectures.

5.9.2.4 xnticks_t xntimer_get_date (xntimer_t * *timer*)

Return the absolute expiration date.

Return the next expiration date of a timer in absolute clock ticks (see note).

Parameters:

timer The address of a valid timer descriptor.

Returns:

The expiration date converted to the current time unit. The special value XN_INFINITE is returned if *timer* is currently inactive.

Environments:

This service can be called from:

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

Rescheduling: never.

Note:

This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are expressed as periodic jiffies. In oneshot mode, clock ticks are expressed as nanoseconds.

5.9.2.5 xnticks_t xntimer_get_interval (xntimer_t * *timer*)

Return the timer interval value.

Return the timer interval value in clock ticks (see note).

Parameters:

timer The address of a valid timer descriptor.

Returns:

The expiration date converted to the current time unit. The special value XN_INFINITE is returned if *timer* is currently inactive or aperiodic.

Environments:

This service can be called from:

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

- User-space task

Rescheduling: never.

Note:
This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are expressed as periodic jiffies. In oneshot mode, clock ticks are expressed as nanoseconds.

5.9.2.6 xnticks_t xntimer_get_timeout (xntimer_t * *timer*)

Return the relative expiration date.

Return the next expiration date of a timer in relative clock ticks (see note).

Parameters:
timer The address of a valid timer descriptor.

Returns:
The expiration date converted to the current time unit. The special value XN_INFINITE is returned if *timer* is currently inactive. In oneshot mode, it might happen that the timer 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.

Environments:

This service can be called from:

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

Rescheduling: never.

Note:
This service is sensitive to the current operation mode of the system timer, as defined by the xnpod_start_timer() service. In periodic mode, clock ticks are expressed as periodic jiffies. In oneshot mode, clock ticks are expressed as nanoseconds.

5.9.2.7 void xntimer_init (xntimer_t * *timer*, void(*)(xntimer_t *timer) *handler*)

Initialize a timer object.

Creates a timer. When created, a timer is left disarmed; it must be started using xntimer_start() in order to be activated.

Parameters:
timer The address of a timer descriptor the nucleus 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.

handler The routine to call upon expiration of the timer.

There is no limitation on the number of timers which can be created/active concurrently.

Environments:

This service can be called from:

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

Rescheduling: never.

5.10 HAL.

5.10.1 Detailed Description

Generic Adeos-based hardware abstraction layer.

Files

- file hal.c

  Adeos-based Real-Time Abstraction Layer for PowerPC.

- file hal.c

  Adeos-based Real-Time Abstraction Layer for the Blackfin architecture.

- file nmi.c

  NMI watchdog support.

- file hal.c

  Generic Real-Time HAL.

- file nmi.c

  Adeos-based Real-Time Abstraction Layer for x86.

- file hal.c

  Adeos-based Real-Time Abstraction Layer for x86.

- file nmi.c

  NMI watchdog for x86, from linux/arch/i386/kernel/nmi.c.

- file smi.c

  SMI workaround for x86.

- file hal.c

  Adeos-based Real-Time Abstraction Layer for ia64.

- file hal.c

  Adeos-based Real-Time Abstraction Layer for PowerPC.

Functions

- int rthal_timer_request (void(*handler)(void), unsigned long nstick)

  Grab the hardware timer.

- void rthal_timer_release (void)

  Release the hardware timer.

- int rthal_irq_host_release (unsigned irq, void *dev_id)

Uninstall a shared Linux interrupt handler.

- int rthal_irq_enable (unsigned irq)

  Enable an interrupt source.

- int rthal_irq_disable (unsigned irq)

  Disable an interrupt source.

- int rthal_irq_request (unsigned irq, rthal_irq_handler_t handler, rthal_irq_ackfn_t ackfn, void *cookie)

  Install a real-time interrupt handler.

- int rthal_irq_release (unsigned irq)

  Uninstall a real-time interrupt handler.

- int rthal_irq_host_pend (unsigned irq)

  Propagate an IRQ event to Linux.

- int rthal_irq_affinity (unsigned irq, cpumask_t cpumask, cpumask_t *oldmask)

  Set/Get processor affinity for external interrupt.

- rthal_trap_handler_t rthal_trap_catch (rthal_trap_handler_t handler)

  Installs a fault handler.

- int rthal_apc_alloc (const char *name, void(*handler)(void *cookie), void *cookie)

  Allocate an APC slot.

- int rthal_apc_free (int apc)

  Releases an APC slot.

- int rthal_apc_schedule (int apc)

  Schedule an APC invocation.

5.10.2 Function Documentation

5.10.2.1 int rthal_apc_alloc (const char * *name*, void(*)(void *cookie) *handler*, void * *cookie*)

Allocate an APC slot.

APC is the acronym for Asynchronous Procedure Call, a mean by which activities from the Xenomai domain can schedule deferred invocations of handlers to be run into the Linux domain, as soon as possible when the Linux kernel gets back in control. Up to BITS_PER_LONG APC slots can be active at any point in time. APC support is built upon Adeos's virtual interrupt support.

The HAL guarantees that any Linux kernel service which would be callable from a regular Linux interrupt handler is also available to APC handlers, including over PREEMPT_RT kernels exhibiting a threaded IRQ model.

Parameters:

name is a symbolic name identifying the APC which will get reported through the /proc/xenomai/apc interface. Passing NULL to create an anonymous APC is allowed.

handler The address of the fault handler to call upon exception condition. The handle will be passed the *cookie* value unmodified.

cookie A user-defined opaque cookie the HAL will pass to the APC handler as its sole argument.

Returns:
an valid APC id. is returned upon success, or a negative error code otherwise:

- -EINVAL is returned if *handler* is invalid.

- -EBUSY is returned if no more APC slots are available.

Environments:

This service can be called from:

- Linux domain context.

5.10.2.2 int rthal_apc_free (int *apc*)

Releases an APC slot.

This service deallocates an APC slot obtained by rthal_apc_alloc().

Parameters:
apc The APC id. to release, as returned by a successful call to the rthal_apc_alloc() service.

Returns:
0 is returned upon success. Otherwise:

- -EINVAL is returned if *apc* is invalid.

Environments:

This service can be called from:

- Any domain context.

5.10.2.3 int rthal_apc_schedule (int *apc*)

Schedule an APC invocation.

This service marks the APC as pending for the Linux domain, so that its handler will be called as soon as possible, when the Linux domain gets back in control.

When posted from the Linux domain, the APC handler is fired as soon as the interrupt mask is explicitely cleared by some kernel code. When posted from the Xenomai domain, the APC handler is fired as soon as the Linux domain is resumed, i.e. after Xenomai has completed all its pending duties.

Parameters:
apc The APC id. to schedule.

Returns:

0 or 1 are returned upon success, the former meaning that the APC was already pending. Otherwise:

- -EINVAL is returned if *apc* is invalid.

Environments:

This service can be called from:

- Any domain context, albeit the usual calling place is from the Xenomai domain.

5.10.2.4 int rthal_irq_affinity (unsigned *irq*, cpumask_t *cpumask*, cpumask_t * *oldmask*)

Set/Get processor affinity for external interrupt.

On SMP systems, this service ensures that the given interrupt is preferably dispatched to the specified set of processors. The previous affinity mask is returned by this service.

Parameters:

irq The interrupt source whose processor affinity is affected by the operation. Only external interrupts can have their affinity changed/queried, thus virtual interrupt numbers allocated by rthal_alloc_virq() are invalid values for this parameter.

cpumask A list of CPU identifiers passed as a bitmask representing the new affinity for this interrupt. A zero value cause this service to return the current affinity mask without changing it.

oldmask If non-NULL, a pointer to a memory area which will bve overwritten by the previous affinity mask used for this interrupt source, or a zeroed mask if an error occurred. This service always returns a zeroed mask on uniprocessor systems.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *irq* is invalid.

Environments:

This service can be called from:

- Linux domain context.

5.10.2.5 int rthal_irq_disable (unsigned *irq*)

Disable an interrupt source.

Disables an interrupt source at PIC level. After this call has returned, no more IRQs from the given source will be allowed, until the latter is enabled again using rthal_irq_enable().

Parameters:

irq The interrupt source to disable. This value is architecture-dependent.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *irq* is invalid.

- Other error codes might be returned in case some internal error happens at the Adeos level. Such error might caused by conflicting Adeos requests made by third-party code.

Environments:

This service can be called from:

- Any domain context.

5.10.2.6 int rthal_irq_enable (unsigned *irq*)

Enable an interrupt source.

Enables an interrupt source at PIC level. Since Adeos masks and acknowledges the associated interrupt source upon IRQ receipt, this action is usually needed whenever the HAL handler does not propagate the IRQ event to the Linux domain, thus preventing the regular Linux interrupt handling code from re-enabling said source. After this call has returned, IRQs from the given source will be enabled again.

Parameters:

irq The interrupt source to enable. This value is architecture-dependent.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *irq* is invalid.

- Other error codes might be returned in case some internal error happens at the Adeos level. Such error might caused by conflicting Adeos requests made by third-party code.

Environments:

This service can be called from:

- Any domain context.

5.10.2.7 int rthal_irq_host_pend (unsigned *irq*)

Propagate an IRQ event to Linux.

Causes the given IRQ to be propagated down to the Adeos pipeline to the Linux kernel. This operation is typically used after the given IRQ has been processed into the Xenomai domain by a real-time interrupt handler (see rthal_irq_request()), in case such interrupt must also be handled by the Linux kernel.

Parameters:

irq The interrupt source to detach the shared handler from. This value is architecture-dependent.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *irq* is invalid.

Environments:

This service can be called from:

- Xenomai domain context.

5.10.2.8 int rthal_irq_host_release (unsigned *irq*, void * *dev_id*)

Uninstall a shared Linux interrupt handler.

Uninstalls a shared interrupt handler from the Linux domain for the given interrupt source. The handler is removed from the existing list of Linux handlers for this interrupt source.

Parameters:

irq The interrupt source to detach the shared handler from. This value is architecture-dependent.

dev_id is a valid device id, identical in essence to the one requested by the free_irq() service provided by the Linux kernel. This value will be used to locate the handler to remove from the chain of existing Linux handlers for the given interrupt source. This parameter must match the device id. passed to rthal_irq_host_request() for the same handler instance.

Returns:

0 is returned upon success. Otherwise:

- -EINVAL is returned if *irq* is invalid.

Environments:

This service can be called from:

- Linux domain context.

5.10.2.9 int rthal_irq_release (unsigned *irq*)

Uninstall a real-time interrupt handler.

Uninstalls an interrupt handler previously attached using the rthal_irq_request() service.

Parameters:

irq The hardware interrupt channel to uninstall a handler from. This value is architecture-dependent.

Returns:
0 is returned upon success. Otherwise:

- -EINVAL is returned if *irq* is invalid.
- Other error codes might be returned in case some internal error happens at the Adeos level. Such error might caused by conflicting Adeos requests made by third-party code.

Environments:

This service can be called from:

- Any domain context.

5.10.2.10 int rthal_irq_request (unsigned *irq*, rthal_irq_handler_t *handler*, rthal_irq_ackfn_t *ackfn*, void * *cookie*)

Install a real-time interrupt handler.

Installs an interrupt handler for the specified IRQ line by requesting the appropriate Adeos virtualization service. The handler is invoked by Adeos on behalf of the Xenomai domain context. Once installed, the HAL interrupt handler will be called prior to the regular Linux handler for the same interrupt source.

Parameters:

irq The hardware interrupt channel to install a handler on. This value is architecture-dependent.

handler The address of a valid interrupt service routine. This handler will be called each time the corresponding IRQ is delivered, and will be passed the *cookie* value unmodified.

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

cookie A user-defined opaque cookie the HAL will pass to the interrupt handler as its sole argument.

Returns:
0 is returned upon success. Otherwise:

- -EBUSY is returned if an interrupt handler is already installed. rthal_irq_release() must be issued first before a handler is installed anew.
- -EINVAL is returned if *irq* is invalid or *handler* is NULL.
- Other error codes might be returned in case some internal error happens at the Adeos level. Such error might caused by conflicting Adeos requests made by third-party code.

Environments:

This service can be called from:

- Any domain context.

5.10.2.11 void rthal_timer_release (void)

Release the hardware timer.

Releases the hardware timer, thus reverting the effect of a previous call to rthal_timer_request(). In case the timer hardware is shared with Linux, a periodic setup suitable for the Linux kernel will be reset.

Environments:

This service can be called from:

- Linux domain context.

5.10.2.12 int rthal_timer_request (void(*)(void) *handler*, unsigned long *nstick*)

Grab the hardware timer.

rthal_timer_request() grabs and tunes the hardware timer so that a user-defined routine is called according to a given frequency. On architectures that provide a oneshot-programmable time source, the hardware timer can operate either in aperiodic or periodic mode. Using the aperiodic mode still allows to run periodic timings over it: the underlying hardware simply needs to be reprogrammed after each tick using the appropriate interval value

The time interval that elapses between two consecutive invocations of the handler is called a tick. The user-supplied handler will always be invoked on behalf of the Xenomai domain for each incoming tick.

Parameters:

handler The address of the tick handler which will process each incoming tick.

nstick The timer period in nanoseconds. If this parameter is zero, the underlying hardware timer is set to operate in oneshot-programming mode. In this mode, timing accuracy is higher - since it is not rounded to a constant time slice - at the expense of a lesser efficicency due to the timer chip programming duties. On the other hand, the shorter the period, the higher the overhead induced by the periodic mode, since the handler will end up consuming a lot of CPU power to process useless ticks.

Returns:

0 is returned on success. Otherwise:

- -EBUSY is returned if the hardware timer has already been grabbed. rthal_timer_request() must be issued before rthal_timer_request() is called again.

Environments:

This service can be called from:

- Linux domain context.

5.10.2.13 int rthal_trap_catch (rthal_trap_handler_t *handler*)

Installs a fault handler.

The HAL attempts to invoke a fault handler whenever an uncontrolled exception or fault is caught at machine level. This service allows to install a user-defined handler for such events.

Parameters:

handler The address of the fault handler to call upon exception condition. The handler is passed the address of the low-level information block describing the fault as passed by Adeos. Its layout is implementation-dependent.

Returns:

The address of the fault handler previously installed.

Environments:

This service can be called from:

- Any domain context.

Chapter 6

Xenomai nanokernel API Data Structure Documentation

6.1 xnpod Struct Reference

`#include <pod.h>`

6.1.1 Detailed Description

Real-time pod descriptor.

The source of all Xenomai magic.

Data Fields

- xnflags_t status
- xnticks_t jiffies
- xnticks_t wallclock_offset
- xntimer_t htimer
- xnsched_t sched [XNARCH_NR_CPUS]
- xnqueue_t threadq
- int threadq_rev
- volatile u_long schedlck
- xnqueue_t tstartq
- xnqueue_t tswitchq
- xnqueue_t tdeleteq
- int minpri
- int maxpri
- int root_prio_base
- u_long tickvalue
- u_long ticks2sec
- int refcnt

- struct {
  void(* settime)(xnticks_t newtime)
  int(* faulthandler)(xnarch_fltinfo_t *fltinfo)
  int(* unload)(void)
  } svctable

6.1.2 Field Documentation

6.1.2.1 int(* xnpod::faulthandler)(xnarch_fltinfo_t *fltinfo)

Trap/exception handler.

6.1.2.2 xntimer_t xnpod::htimer

Host timer.

6.1.2.3 xnticks_t xnpod::jiffies

Periodic ticks elapsed since boot.

6.1.2.4 int xnpod::maxpri

Maximum priority value.

6.1.2.5 int xnpod::minpri

Minimum priority value.

6.1.2.6 int xnpod::refcnt

Reference count.

6.1.2.7 int xnpod::root_prio_base

Base priority of ROOT thread.

6.1.2.8 xnsched_t xnpod::sched[XNARCH_NR_CPUS]

Per-cpu scheduler slots.

6.1.2.9 volatile u_long xnpod::schedlck

Scheduler lock count.

6.1.2.10 void(* xnpod::settime)(xnticks_t newtime)

Clock setting hook.

6.1.2.11 xnflags_t xnpod::status

Status bitmask.

6.1.2.12 struct { ... } xnpod::svctable

Table of overridable service entry points.

6.1.2.13 xnqueue_t xnpod::tdeleteq

Thread delete hook queue.

6.1.2.14 xnqueue_t xnpod::threadq

All existing threads.

6.1.2.15 int xnpod::threadq_rev

Modification counter of threadq.

6.1.2.16 u_long xnpod::ticks2sec

Number of ticks per second (1e9 if aperiodic).

6.1.2.17 u_long xnpod::tickvalue

Tick duration (ns, 1 if aperiodic).

6.1.2.18 xnqueue_t xnpod::tstartq

Thread start hook queue.

6.1.2.19 xnqueue_t xnpod::tswitchq

Thread switch hook queue.

6.1.2.20 int(* xnpod::unload)(void)

Unloading hook.

6.1.2.21 xnticks_t xnpod::wallclock_offset

Difference between wallclock time and epoch in ticks.

The documentation for this struct was generated from the following file:

- include/nucleus/pod.h

6.2 xnsched Struct Reference

`#include <pod.h>`

6.2.1 Detailed Description

Scheduling information structure.

Data Fields

- xnflags_t status
- xnthread_t * runthread
- xnarch_cpumask_t resched
- xnsched_queue_t readyq
- volatile unsigned inesting
- xnthread_t rootcb

6.2.2 Field Documentation

6.2.2.1 volatile unsigned xnsched::inesting

Interrupt nesting level.

6.2.2.2 xnsched_queue_t xnsched::readyq

Ready-to-run threads (prioritized).

6.2.2.3 xnarch_cpumask_t xnsched::resched

Mask of CPUs needing rescheduling.

6.2.2.4 xnthread_t xnsched::rootcb

Root thread control block.

6.2.2.5 xnthread_t* xnsched::runthread

Current thread (service or user).

6.2.2.6 xnflags_t xnsched::status

Scheduler specific status bitmask

The documentation for this struct was generated from the following file:

- include/nucleus/pod.h

Chapter 7

Xenomai nanokernel API File Documentation

7.1 include/nucleus/pod.h File Reference

7.1.1 Detailed Description

Real-time pod interface header.

Author:
Philippe Gerum

`#include <nucleus/thread.h>`

`#include <nucleus/intr.h>`

Data Structures

- struct xnsched

  Scheduling information structure.

- struct xnpod

  Real-time pod descriptor.

Typedefs

- typedef xnsched xnsched_t

  Scheduling information structure.

Functions

- void xnpod_schedule_runnable (xnthread_t *thread, int flags)

  Hidden rescheduling procedure.

- int xnpod_init (xnpod_t *pod, int minpri, int maxpri, xnflags_t flags)

  Initialize a new pod.

- int xnpod_start_timer (u_long nstick, xnisr_t tickhandler)

  Start the system timer.

- void xnpod_stop_timer (void)

  Stop the system timer.

- int xnpod_reset_timer (void)

  Reset the system timer.

- void xnpod_shutdown (int xtype)

  Shutdown the current pod.

- int xnpod_init_thread (xnthread_t *thread, const char *name, int prio, xnflags_t flags, unsigned stacksize)

  Initialize a new thread.

- int xnpod_start_thread (xnthread_t *thread, xnflags_t mode, int imask, xnarch_cpumask_t affinity, void(*entry)(void *cookie), void *cookie)

  Initial start of a newly created thread.

- void xnpod_restart_thread (xnthread_t *thread)

  Restart a thread.

- void xnpod_delete_thread (xnthread_t *thread)

  Delete a thread.

- xnflags_t xnpod_set_thread_mode (xnthread_t *thread, xnflags_t clrmask, xnflags_t setmask)

  Change a thread's control mode.

- void xnpod_resume_thread (xnthread_t *thread, xnflags_t mask)

Resume a thread.

- int xnpod_unblock_thread (xnthread_t *thread)
  Unblock a thread.
- void xnpod_renice_thread (xnthread_t *thread, int prio)
  Change the base priority of a thread.
- int xnpod_migrate_thread (int cpu)
  Migrate the current thread.
- void xnpod_rotate_readyq (int prio)
  Rotate a priority level in the ready queue.
- void xnpod_schedule (void)
  Rescheduling procedure entry point.
- void xnpod_dispatch_signals (void)
  Deliver pending asynchronous signals to the running thread.
- void xnpod_activate_rr (xnticks_t quantum)
  Globally activate the round-robin scheduling.
- void xnpod_deactivate_rr (void)
  Globally deactivate the round-robin scheduling.
- void xnpod_set_time (xnticks_t newtime)
  Set the nucleus idea of time.
- int xnpod_set_thread_periodic (xnthread_t *thread, xnticks_t idate, xnticks_t period)
  Make a thread periodic.
- int xnpod_wait_thread_period (unsigned long *overruns_r)
  Wait for the next periodic release point.
- xnticks_t xnpod_get_time (void)
  Get the nucleus idea of time.
- int xnpod_add_hook (int type, void(*routine)(xnthread_t *))
  Install a nucleus hook.
- int xnpod_remove_hook (int type, void(*routine)(xnthread_t *))
  Remove a nucleus hook.

7.2 include/nucleus/registry.h File Reference

7.2.1 Detailed Description

This file is part of the Xenomai project.

Note:

Copyright (C) 2004 Philippe Gerum <rpm@xenomai.org>


```
#include <nucleus/types.h>
```

7.3 ksrc/arch/arm/hal.c File Reference

7.3.1 Detailed Description

Adeos-based Real-Time Abstraction Layer for PowerPC.


```
#include <linux/version.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <linux/console.h>
#include <asm/system.h>
#include <asm/hardirq.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/xenomai/hal.h>
#include <stdarg.h>
```

Functions

- int rthal_timer_request (void(*handler)(void), unsigned long nstick)

  Grab the hardware timer.

- void rthal_timer_release (void)

  Release the hardware timer.

- int rthal_irq_host_release (unsigned irq, void *dev_id)

  Uninstall a shared Linux interrupt handler.

- int rthal_irq_enable (unsigned irq)

  Enable an interrupt source.

- int rthal_irq_disable (unsigned irq)
  Disable an interrupt source.

7.4 ksrc/arch/blackfin/hal.c File Reference

7.4.1 Detailed Description

Adeos-based Real-Time Abstraction Layer for the Blackfin architecture.


```
#include <linux/version.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <asm/system.h>
#include <asm/atomic.h>
#include <asm/irqchip.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/xenomai/hal.h>
```

Functions

- int rthal_timer_request (void(*handler)(void), unsigned long nstick)
  Grab the hardware timer.
- void rthal_timer_release (void)
  Release the hardware timer.
- int rthal_irq_enable (unsigned irq)
  Enable an interrupt source.
- int rthal_irq_disable (unsigned irq)
  Disable an interrupt source.
- int rthal_irq_host_release (unsigned irq, void *dev_id)
  Uninstall a shared Linux interrupt handler.

7.5 ksrc/arch/generic/hal.c File Reference

7.5.1 Detailed Description

Generic Real-Time HAL.


```
#include <linux/version.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <linux/console.h>
#include <linux/kallsyms.h>
#include <asm/system.h>
#include <asm/hardirq.h>
#include <asm/irq.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/xenomai/hal.h>
#include <stdarg.h>
```

Functions

- int rthal_irq_request (unsigned irq, rthal_irq_handler_t handler, rthal_irq_ackfn_t ackfn, void *cookie)

  Install a real-time interrupt handler.

- int rthal_irq_release (unsigned irq)

  Uninstall a real-time interrupt handler.

- int rthal_irq_host_pend (unsigned irq)

  Propagate an IRQ event to Linux.

- int rthal_irq_affinity (unsigned irq, cpumask_t cpumask, cpumask_t *oldmask)

  Set/Get processor affinity for external interrupt.

- rthal_trap_handler_t rthal_trap_catch (rthal_trap_handler_t handler)
  Installs a fault handler.
- int rthal_apc_alloc (const char *name, void(*handler)(void *cookie), void *cookie)
  Allocate an APC slot.
- int rthal_apc_free (int apc)
  Releases an APC slot.
- int rthal_apc_schedule (int apc)
  Schedule an APC invocation.

7.6 ksrc/arch/i386/hal.c File Reference

7.6.1 Detailed Description

Adeos-based Real-Time Abstraction Layer for x86.

Inspired from original RTAI/x86 HAL interface:

Copyright ©2000 Paolo Mantegazza,

Copyright ©2000 Steve Papacharalambous,

Copyright ©2000 Stuart Hughes,

RTAI/x86 rewrite over Adeos:

Copyright ©2002 Philippe Gerum. NMI watchdog, SMI workaround:

Copyright ©2004 Gilles Chanteperdrix.

Xenomai 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, Inc., 675 Mass Ave, Cambridge MA 02139, USA; either version 2 of the License, or (at your option) any later version.

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

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

```
#include <linux/version.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <linux/console.h>
#include <asm/system.h>
#include <asm/hardirq.h>
#include <asm/desc.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/xenomai/hal.h>
#include <stdarg.h>
```

Functions

- int rthal_timer_request (void(*handler)(void), unsigned long nstick)

  Grab the hardware timer.

- void rthal_timer_release (void)

Release the hardware timer.

- int rthal_irq_host_release (unsigned irq, void *dev_id)

  Uninstall a shared Linux interrupt handler.

- int rthal_irq_enable (unsigned irq)

  Enable an interrupt source.

- int rthal_irq_disable (unsigned irq)

  Disable an interrupt source.

7.7 ksrc/arch/ia64/hal.c File Reference

7.7.1 Detailed Description

Adeos-based Real-Time Abstraction Layer for ia64.


```
#include <linux/version.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <linux/console.h>
#include <asm/system.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/xenomai/hal.h>
#include <stdarg.h>
```

Functions

- int rthal_timer_request (void(*handler)(void), unsigned long nstick)
  Grab the hardware timer.
- void rthal_timer_release (void)
  Release the hardware timer.
- int rthal_irq_host_release (unsigned irq, void *dev_id)
  Uninstall a shared Linux interrupt handler.
- int rthal_irq_enable (unsigned irq)
  Enable an interrupt source.
- int rthal_irq_disable (unsigned irq)
  Disable an interrupt source.

7.8 ksrc/arch/powerpc/hal.c File Reference

7.8.1 Detailed Description

Adeos-based Real-Time Abstraction Layer for PowerPC.


```
#include <linux/version.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <linux/irq.h>
#include <linux/console.h>
#include <asm/system.h>
#include <asm/hardirq.h>
#include <asm/hw_irq.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/xenomai/hal.h>
#include <stdarg.h>
```

Functions

- int rthal_timer_request (void(*handler)(void), unsigned long nstick)

  Grab the hardware timer.

- void rthal_timer_release (void)

  Release the hardware timer.

- int rthal_irq_host_release (unsigned irq, void *dev_id)

  Uninstall a shared Linux interrupt handler.

- int rthal_irq_enable (unsigned irq)

Enable an interrupt source.

- int rthal_irq_disable (unsigned irq)

  Disable an interrupt source.

7.9 ksrc/arch/blackfin/nmi.c File Reference

7.9.1 Detailed Description

NMI watchdog support.


```
#include <linux/version.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <asm/system.h>
#include <asm/atomic.h>
#include <asm/irqchip.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/xenomai/hal.h>
```

7.10 ksrc/arch/generic/nmi.c File Reference

7.10.1 Detailed Description

Adeos-based Real-Time Abstraction Layer for x86.


```
#include <linux/version.h>
#include <linux/module.h>
#include <asm/system.h>
#include <asm/unistd.h>
#include <asm/uaccess.h>
#include <asm/xenomai/hal.h>
```

7.11 ksrc/arch/i386/nmi.c File Reference

7.11.1 Detailed Description

NMI watchdog for x86, from linux/arch/i386/kernel/nmi.c.

Original authors: Ingo Molnar, Mikael Pettersson, Pavel Machek.

Adaptation to Xenomai by Gilles Chanteperdrix


```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/nmi.h>
#include <asm/nmi.h>
#include <asm/msr.h>
#include <asm/xenomai/hal.h>
```

7.12 ksrc/arch/i386/smi.c File Reference

7.12.1 Detailed Description

SMI workaround for x86.

Cut/Pasted from Vitor Angelo "smi" module. Adapted by Gilles Chanteperdrix <gilles.chanteperdrix@laposte.net>.


```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/pci.h>
#include <linux/pci_ids.h>
#include <linux/reboot.h>
#include <asm/xenomai/hal.h>
```

7.13 ksrc/nucleus/heap.c File Reference

7.13.1 Detailed Description

Dynamic memory allocation services.

Author:
Philippe Gerum


```
#include <nucleus/pod.h>
#include <nucleus/thread.h>
#include <nucleus/heap.h>
#include <asm/xenomai/bits/heap.h>
```

Functions

- int xnheap_init (xnheap_t *heap, void *heapaddr, u_long heapsize, u_long pagesize)
  Initialize a memory heap.
- int xnheap_destroy (xnheap_t *heap, void(*flushfn)(xnheap_t *heap, void *extaddr, u_long extsize, void *cookie), void *cookie)
  Destroys a memory heap.
- void * xnheap_alloc (xnheap_t *heap, u_long size)
  Allocate a memory block from a memory heap.
- int xnheap_test_and_free (xnheap_t *heap, void *block, int(*ckfn)(void *block))
  Test and release a memory block to a memory heap.
- int xnheap_free (xnheap_t *heap, void *block)
  Release a memory block to a memory heap.
- int xnheap_extend (xnheap_t *heap, void *extaddr, u_long extsize)
  Extend a memory heap.
- void xnheap_schedule_free (xnheap_t *heap, void *block, xnholder_t *link)
  Schedule a memory block for release.

7.14 ksrc/nucleus/intr.c File Reference

7.14.1 Detailed Description

Interrupt management.

Author:

Philippe Gerum


```
#include <nucleus/pod.h>
#include <nucleus/intr.h>
#include <nucleus/ltt.h>
#include <nucleus/stat.h>
#include <asm/xenomai/bits/intr.h>
```

Functions

- int xnintr_init (xnintr_t *intr, const char *name, unsigned irq, xnisr_t isr, xniack_t iack, xnflags_t flags)

  Initialize an interrupt object.

- int xnintr_destroy (xnintr_t *intr)

  Destroy an interrupt object.

- int xnintr_attach (xnintr_t *intr, void *cookie)

  Attach an interrupt object.

- int xnintr_detach (xnintr_t *intr)

  Detach an interrupt object.

- int xnintr_enable (xnintr_t *intr)

  Enable an interrupt object.

- int xnintr_disable (xnintr_t *intr)

  Disable an interrupt object.

- xnarch_cpumask_t xnintr_affinity (xnintr_t *intr, xnarch_cpumask_t cpumask)

  Set interrupt's processor affinity.

7.15 ksrc/nucleus/pod.c File Reference

7.15.1 Detailed Description

Real-time pod services.

Author:

Philippe Gerum


```
#include <stdarg.h>
#include <nucleus/pod.h>
#include <nucleus/timer.h>
#include <nucleus/synch.h>
#include <nucleus/heap.h>
#include <nucleus/intr.h>
#include <nucleus/registry.h>
#include <nucleus/module.h>
#include <nucleus/ltt.h>
#include <nucleus/stat.h>
#include <asm/xenomai/bits/pod.h>
```

Functions

- int xnpod_init (xnpod_t *pod, int minpri, int maxpri, xnflags_t flags)

  Initialize a new pod.

- void xnpod_shutdown (int xtype)

  Shutdown the current pod.

- int xnpod_init_thread (xnthread_t *thread, const char *name, int prio, xnflags_t flags, unsigned stacksize)

  Initialize a new thread.

- int xnpod_start_thread (xnthread_t *thread, xnflags_t mode, int imask, xnarch_cpumask_t affinity, void(*entry)(void *cookie), void *cookie)

  Initial start of a newly created thread.

- void xnpod_restart_thread (xnthread_t *thread)

  Restart a thread.

- xnflags_t xnpod_set_thread_mode (xnthread_t *thread, xnflags_t clrmask, xnflags_t setmask)

  Change a thread's control mode.

- void xnpod_delete_thread (xnthread_t *thread)

  Delete a thread.

- void xnpod_suspend_thread (xnthread_t *thread, xnflags_t mask, xnticks_t timeout, xnsynch_t *wchan)

  Suspend a thread.

- void xnpod_resume_thread (xnthread_t *thread, xnflags_t mask)

  Resume a thread.

- int xnpod_unblock_thread (xnthread_t *thread)

  Unblock a thread.

- void xnpod_renice_thread (xnthread_t *thread, int prio)

  Change the base priority of a thread.

- int xnpod_migrate_thread (int cpu)

  Migrate the current thread.

- void xnpod_rotate_readyq (int prio)

  Rotate a priority level in the ready queue.

- void xnpod_activate_rr (xnticks_t quantum)

  Globally activate the round-robin scheduling.

- void xnpod_deactivate_rr (void)

  Globally deactivate the round-robin scheduling.

- void xnpod_dispatch_signals (void)

  Deliver pending asynchronous signals to the running thread.

- void xnpod_welcome_thread (xnthread_t *thread, int imask)

  Thread prologue.

- static void xnpod_preempt_current_thread (xnsched_t *sched)

  Preempts the current thread.

- void xnpod_schedule (void)

Rescheduling procedure entry point.

- void xnpod_schedule_runnable (xnthread_t *thread, int flags)
  Hidden rescheduling procedure.
- void xnpod_set_time (xnticks_t newtime)
  Set the nucleus idea of time.
- xnticks_t xnpod_get_time (void)
  Get the nucleus idea of time.
- int xnpod_add_hook (int type, void(*routine)(xnthread_t *))
  Install a nucleus hook.
- int xnpod_remove_hook (int type, void(*routine)(xnthread_t *))
  Remove a nucleus hook.
- int xnpod_trap_fault (void *fltinfo)
  Default fault handler.
- int xnpod_start_timer (u_long nstick, xnisr_t tickhandler)
  Start the system timer.
- void xnpod_stop_timer (void)
  Stop the system timer.
- int xnpod_reset_timer (void)
  Reset the system timer.
- int xnpod_announce_tick (xnintr_t *intr)
  Announce a new clock tick.
- int xnpod_set_thread_periodic (xnthread_t *thread, xnticks_t idate, xnticks_t period)
  Make a thread periodic.
- int xnpod_wait_thread_period (unsigned long *overruns_r)
  Wait for the next periodic release point.

7.16 ksrc/nucleus/registry.c File Reference

7.16.1 Detailed Description

This file is part of the Xenomai project.

Note:


```
#include <nucleus/pod.h>
#include <nucleus/heap.h>
#include <nucleus/registry.h>
#include <nucleus/thread.h>
```

Functions

- int xnregistry_enter (const char *key, void *objaddr, xnhandle_t *phandle, xnpnode_t *pnode)
  Register a real-time object.
- int xnregistry_bind (const char *key, xnticks_t timeout, xnhandle_t *phandle)
  Bind to a real-time object.
- int xnregistry_remove (xnhandle_t handle)
  Forcibly unregister a real-time object.
- int xnregistry_remove_safe (xnhandle_t handle, xnticks_t timeout)
  Unregister an idle real-time object.
- void * xnregistry_get (xnhandle_t handle)
  Find and lock a real-time object into the registry.
- u_long xnregistry_put (xnhandle_t handle)
  Unlock a real-time object from the registry.
- void * xnregistry_fetch (xnhandle_t handle)
  Find a real-time object into the registry.

7.17 ksrc/nucleus/shadow.c File Reference

7.17.1 Detailed Description

Real-time shadow services.

Author:
Philippe Gerum


```
#include <stdarg.h>
#include <linux/unistd.h>
#include <linux/wait.h>
#include <linux/init.h>
#include <linux/kthread.h>
#include <asm/signal.h>
#include <nucleus/pod.h>
#include <nucleus/heap.h>
#include <nucleus/synch.h>
#include <nucleus/module.h>
#include <nucleus/shadow.h>
#include <nucleus/core.h>
#include <nucleus/ltt.h>
#include <nucleus/jhash.h>
#include <nucleus/ppd.h>
#include <nucleus/trace.h>
#include <nucleus/stat.h>
#include <asm/xenomai/features.h>
#include <asm/xenomai/syscall.h>
```

`#include <asm/xenomai/bits/shadow.h>`

Functions

- int xnshadow_harden (void)

  Migrate a Linux task to the Xenomai domain.

- void xnshadow_relax (int notify)

  Switch a shadow thread back to the Linux domain.

- int xnshadow_map (xnthread_t *thread, xncompletion_t __user *u_completion)

  Create a shadow thread context.

- xnshadow_ppd_t * xnshadow_ppd_get (unsigned muxid)

  Return the per-process data attached to the calling process.

7.18 ksrc/nucleus/synch.c File Reference

7.18.1 Detailed Description

Thread synchronization services.

Author:

Philippe Gerum


```
#include <stdarg.h>
#include <nucleus/pod.h>
#include <nucleus/synch.h>
#include <nucleus/thread.h>
#include <nucleus/module.h>
#include <nucleus/ltt.h>
```

Functions

- void xnsynch_init (xnsynch_t *synch, xnflags_t flags)

  Initialize a synchronization object.

- void xnsynch_sleep_on (xnsynch_t *synch, xnticks_t timeout)

  Sleep on a synchronization object.

- static void xnsynch_clear_boost (xnsynch_t *synch, xnthread_t *lastowner)

  Clear the priority boost.

- void xnsynch_renice_sleeper (xnthread_t *thread)

  Change a sleeper's priority.

- xnthread_t * xnsynch_wakeup_one_sleeper (xnsynch_t *synch)

  Give the resource ownership to the next waiting thread.

- xnpholder_t * xnsynch_wakeup_this_sleeper (xnsynch_t *synch, xnpholder_t *holder)

  Give the resource ownership to a given waiting thread.

- int xnsynch_flush (xnsynch_t *synch, xnflags_t reason)

  Unblock all waiters pending on a resource.

- void xnsynch_forget_sleeper (xnthread_t *thread)

  Abort a wait for a resource.

- void xnsynch_release_all_ownerships (xnthread_t *thread)

  Release all ownerships.

7.19 ksrc/nucleus/timer.c File Reference

7.19.1 Detailed Description

Note:


```
#include <nucleus/pod.h>
#include <nucleus/thread.h>
#include <nucleus/timer.h>
#include <asm/xenomai/bits/timer.h>
```

Functions

- static void xntimer_do_tick_aperiodic (void)
  Process a timer tick in aperiodic mode.
- void xntimer_init (xntimer_t *timer, void(*handler)(xntimer_t *timer))
  Initialize a timer object.
- void xntimer_destroy (xntimer_t *timer)
  Release a timer object.
- xnticks_t xntimer_get_date (xntimer_t *timer)
  Return the absolute expiration date.
- xnticks_t xntimer_get_timeout (xntimer_t *timer)
  Return the relative expiration date.
- xnticks_t xntimer_get_interval (xntimer_t *timer)
  Return the timer interval value.
- void xntimer_freeze (void)
  Freeze all timers.

Index