# Migrating from Xenomai 2.x to 2.99.0

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| | | | |

# Contents

The latest version of this document is available at this address.

For questions, corrections and improvements, write to the mailing list.

## 1 Configuration

USER PROGRAMS AND LIBRARIES

- As with Xenomai 2.x, `xeno-config` is available for retrieving the compilation and link flags for building Xenomai 3.x applications. This script will work for both the Cobalt and Mercury environments indifferently.
  - Each `--skin=<api>` option specifier can be abbreviated as --<api>. For instance, `--psos` is a shorthand for `--skin=psos` on the command line.
  - Specifying `--[skin=]cobalt` or `--[skin=]posix` on the command line is strictly equivalent. However, this does not make sense with *Mercury* which does not define these switches.
  - `--[skin=]alchemy` replaces the former `--skin=native` switch.
  - `--core` can be used to retrieve the name of the Xenomai core system for which `xeno-config` was generated. Possible output values are `cobalt` and `mercury`.
  - `--no-auto-init` can be passed to disable automatic initialization of the Copperplate library when the application process enters the `main()` routine. In such a case, the application code using any API based on the Copperplate layer, shall call the `copperplate_init()` routine manually, as part of its initialization process, *before* any real-time service is invoked.
- `--enable-x86-sep` was renamed to `--enable-x86-vsyscall` to fix a misnomer. This option should be left enabled (default), unless **linuxthreads** are used instead of **NPTL**.

## 2 Getting the system state

Querying the state of the real-time system should be done via the new Xenomai registery interface available with Xenomai 3.x, which is turned on when `--enable-registry` is passed to the configuration script for building the Xenomai libraries and programs.

The new registry support is common to the Cobalt and Mercury cores, with only marginal differences due to the presence (or lack of) co- kernel in the system.

### 2.1 /proc/xenomai interface changes

The legacy /proc/xenomai interface is still available when running over the Cobalt core. The following changes compared to Xenomai 2.x took place though:

- Thread status

All pseudo-files reporting the various thread states moved under the new `sched/` hierarchy, i.e.

`{sched, stat, acct}` → `sched/{threads, stat, acct}`

- Clocks

With the introduction of dynamic clock registration in the Cobalt core, the `clock/` hierarchy was added, to reflect the current state of all timers from the registered Xenomai clocks.

There is no kernel-based time base management anymore with Xenomai 2.99.0. Functionally speaking, only the former *master* time base remains, periodic timing is now controlled locally from the Xenomai libraries in user-space.

Xenomai 2.99.0 defines a built-in clock named *coreclk*, which has the same properties than the former *master* time base available with Xenomai 2.x (i.e. tickless with nanosecond resolution).

As a consequence of these changes, The information once available from `timerstat/master` are now obtained by reading `clock/coreclk`, and the `timebases` file was removed.

- Core clock gravity

The gravity value for a Xenomai clock gives the amount of time expressed in clock ticks, by which the next shot should be anticipated. This is a static adjustment value, to take into account the basic latency of the system for responding to an external event (hardware-wise, and software-wise). This value is commonly assessed by measuring the typical latency of an idle system.

The `latency` pseudo-file holds this value expressed as a count of nanoseconds.

`latency` now represents the sum of the scheduling **and** hardware timer reprogramming latencies of the core clock. This departs from Xenomai 2.x for which only the former was accounted for in the clock gravity value.

- `interfaces` was removed

Only the POSIX and RTDM APIs remain implemented directly in kernel space. All other APIs are implemented in user-space over the Copperplate layer. This makes the former `interfaces` contents basically useless, since the corresponding information for the POSIX/RTDM interfaces can be obtained via `sched/threads`.

- `registry/usage` changed format

The new print out is %<used slot count>/%<total slot count>.

## 3 Binary object features

### 3.1 Loading Xenomai libraries dynamically

The new `--enable-dlopen-libs` configuration switch must be turned on to allow Xenomai libaries to be dynamically loaded via dlopen(3).

This replaces the former `--enable-dlopen-skins` switch. Unlike the latter, `--enable-dlopen-libs` does not implicitly disable support for thread local storage, but rather selects a suitable TLS model (i.e. *global-dynamic*).

### 3.2 Thread local storage

The former `--with-__thread` configuration switch was renamed `--enable-tls`.

As mentioned earlier, TLS is now available to dynamically loaded Xenomai libraries, e.g. `--enable-tls --enable-dlopen-libs` on a configuration line is valid. This would select the *global-dynamic* TLS model instead of *initial-exec*, to make sure all thread-local variables may be accessed from any code module.

## 4 Process-level management

### 4.1 Main thread shadowing

By default, any application linking against `libcobalt` has its main thread attached to the real-time system automatically, this process is called *auto-shadowing*.

This behavior may be disabled at runtime, by setting the XENO_NOSHADOW variable in the application process environment, before the `libcobalt` library constructors are executed.

This replaces the former static mechanism available with Xenomai 2.x, based on turning on `--enable-dlopen-skins` when configuring. Starting with Xenomai 3.x, applications should set the XENO_NOSHADOW variable using putenv(3), before loading `libcolbalt` using dlopen(3).

When auto-shadowing is enabled, global memory locking is also performed, and remains in effect afterwards (i.e. mlockall(MCL_CURRENT|MCL_FUTURE)).

### 4.2 Shadow signal handler

Xenomai's `libcobalt` installs a handler for the SIGWINCH (aka *SIGSHADOW*) signal. This signal may be sent by the Cobalt core to any real-time application, for handling internal duties.

Applications are allowed to interpose on the SIGSHADOW handler, provided they first forward all signal notifications to this routine, then eventually handle all events the Xenomai handler won't process.

This handler was renamed from `xeno_sigwinch_handler()` (Xenomai 2.x) to `cobalt_sigshadow_handler()` in Xenomai 3.x. The function prototype did not change though, i.e.:

```
int cobalt_sigshadow_handler(int sig, siginfo_t *si, void *ctxt)
```

A non-zero value is returned whenever the event was handled internally by the Xenomai system.

### 4.3 Debug signal handler

Xenomai's `libcobalt` installs a handler for the SIGXCPU (aka *SIGDEBUG*) signal. This signal may be sent by the Cobalt core to any real-time application, for notifying various debug events.

Applications are allowed to interpose on the SIGDEBUG handler, provided they eventually forward all signal notifications they won't process to the Xenomai handler.

This handler was renamed from `xeno_handle_mlock_alert()` (Xenomai 2.x) to `cobalt_sigdebug_handler()` in Xenomai 3.x. The function prototype did not change though, i.e.:

```
void cobalt_sigdebug_handler(int sig, siginfo_t *si, void *ctxt)
```

### 4.4 Copperplate auto-initialization

Copperplate is a library layer which mediates between the real-time core services available on the platform, and the API exposed to the application. It provides typical programming abstractions for emulating real-time APIs. All non-POSIX APIs are based on Copperplate services (e.g. *alchemy*, *psos*, *vxworks*).

When Copperplate is built for running over the Cobalt core, it sits on top of the `libcobalt` library. Conversely, it is directly stacked on top of the **glibc** when built for running over the Mercury core.

Normally, Copperplate should initialize from a call issued by the `main()` application routine. To make this process transparent for the user, the `xeno-config` script emits link flags which temporarily overrides the `main()` routine with a Copperplate-based replacement, running the proper initialization code as required, before branching back to the user-defined application entry point.

This behavior may be disabled by passing the `--no-auto-init` option (see the next section).

## 5 RTDM interface changes

FILE RENAMES

- Redundant prefixes were removed from the following files:

rtdm/rtdm_driver.h → rtdm/driver.h

rtdm/rtcan.h → rtdm/can.h

rtdm/rtserial.h → rtdm/serial.h

rttdm/rttesting.h → rtdm/testing.h

rtdm/rtipc.h → rtdm/ipc.h

DRIVER API

- rtdm_task_init() shall be called from secondary mode.

## 6 POSIX interface changes

As mentioned earlier, the former **POSIX skin** is known as the **Cobalt API** in Xenomai 3.x, available as `libcobalt.{so,a}`. The Cobalt API also includes the code of the former `libxenomai`, which is no more a standalone library.

`libcobalt` exposes the set of POSIX and ISO/C standard features specifically implemented by Xenomai to honor real-time requirements using the Cobalt core.

INTERRUPT MANAGEMENT

- The former `pthread_intr` API once provided by Xenomai 2.x is gone.

  Rationale: handling real-time interrupt events from user-space can be done safely only if some top-half code exists for acknowledging the issuing device request from kernel space. This should be done via a RTDM driver, exposing a `read(2)` or `ioctl(2)` interface, for waiting for interrupt events from applications running in user-space.

  Failing this, the low-level interrupt service code in user-space would be sensitive to external thread management actions, such as being stopped because of GDB/ptrace(2) interaction. Unfortunately, preventing the device acknowledge code from running upon interrupt request may cause unfixable breakage to happen (e.g. IRQ storm typically).

  Since the application should provide proper top-half code in a dedicated RTDM driver for synchronizing on IRQ receipt, the RTDM API available in user-space is sufficient.

  Removing the `pthread_intr` API should be considered as a strong hint for keeping the top-half interrupt handling code in kernel space.

  **Tip**
  For receiving interrupt notifications within your application, you should create a small RTDM driver handling the corresponding IRQ (see `rtdm_irq_request()`). The IRQ handler should wake a thread up in your application by posting some event (see `rtdm_sem_up()`, `rtdm_sem_timeddown()`). The application should sleep on this event by calling into the RTDM driver, either via a `read(2)` or `ioctl(2)` request, handled by a dedicated operation handler (see `rtdm_dev_register()`).

SCHEDULING

- Calling `pthread_setschedparam()` may cause a secondary mode switch for the caller, but does not cause any mode switch for the target thread unlike with Xenomai 2.x.

  This is a requirement for maintaining both the **glibc** and the Xenomai scheduler in sync, with respect to thread priorities, since the former maintains a process-local priority cache for the threads it knows about. Therefore, an explicit call to the the regular `pthread_setschedparam()` shall be issued upon each priority change Xenomai-wise, for maintaining consistency.

  In the Xenomai 2.x implementation, the thread being set a new priority would receive a SIGSHADOW signal, eventually handled as a request to call **glibc**'s `pthread_setschedparam()` immediately.

  Rationale: the target Xenomai thread may hold a mutex or any resource which may only be held in primary mode, in which case switching to secondary mode for applying the priority change at any random location over a signal handler may create a pathological issue. In addition, **glibc**'s `pthread_setschedparam()` is not async-safe, which makes the former method fragile.

  Conversely, a thread which calls `pthread_setschedparam()` does know unambiguously whether the current calling context is safe for the incurred migration.

- A new SCHED_WEAK class is available to POSIX threads, which may be optionally turned on using the `CONFIG_XENO_OPT_SCHED_WEAK` kernel configuration switch.

  By this feature, Xenomai now accepts Linux real-time scheduling policies (SCHED_FIFO, SCHED_RR) to be weakly scheduled by the Cobalt core, within a low priority scheduling class (i.e. below the Xenomai real-time classes, but still above the idle class).

  Xenomai 2.x already had a limited form of such policy, based on scheduling SCHED_OTHER threads at the special SCHED_FIFO,0 priority level in the Xenomai core. SCHED_WEAK is a generalization of such policy, which provides for 99 priority levels, to cope with the full extent of the regular Linux SCHED_FIFO/RR priority range.

For instance, a (non real-time) Xenomai thread within the SCHED_WEAK class at priority level 20 in the Cobalt core, may be scheduled with policy SCHED_FIFO/RR at priority 20, by the Linux kernel. The code fragment below would set the scheduling parameters accordingly, assuming the Cobalt version of `pthread_setschedparam()` is invoked:

```
        struct sched_param param = {
               .sched_priority = -20,
        };

        pthread_setschedparam(tid, SCHED_FIFO, &param);
```

Switching a thread to the SCHED_WEAK class can be done by negating the priority level in the scheduling parameters sent to the Cobalt core. For instance, SCHED_FIFO, prio=-7 would be scheduled as SCHED_WEAK, prio=7 by the Cobalt core.

SCHED_OTHER for a Xenomai-enabled thread is scheduled as SCHED_WEAK,0 by the Cobalt core. When the SCHED_WEAK support is disabled in the kernel configuration, only SCHED_OTHER is available for weak scheduling of threads by the Cobalt core.

- A new SCHED_QUOTA class is available to POSIX threads, which may be optionally turned on using the `CONFIG_XENO_OPT_SCHED_QUOTA` kernel configuration switch.

  This policy enforces a limitation on the CPU consumption of threads over a globally defined period, known as the quota interval. This is done by pooling threads with common requirements in groups, and giving each group a share of the global period (see CONFIG_XENO_OPT_SCHED_QUOTA_PERIOD).

  When threads have entirely consumed the quota allotted to the group they belong to, the latter is suspended as a whole, until the next quota interval starts. At this point, a new runtime budget is given to each group, in accordance with its share.

- When called from primary mode, sched_yield() now relinquishes the CPU in case no context switch happened as a result of the manual round-robin.

  Typically, a Xenomai thread undergoing the SCHED_FIFO or SCHED_RR policy with no contender at the same priority level would still be delayed for a while. This delay ends next time the regular Linux kernel switches tasks, or a kernel (virtual) tick has elapsed (TICK_NSEC), whichever comes first.

  Rationale: it is most probably unwanted that sched_yield() does not cause any context switch, since this service is commonly used for implementing a poor man's cooperative scheduling. A typical use case involves a Xenomai thread running in primary mode which needs to yield the CPU to another thread running in secondary mode. By waiting for a context switch to happen in the regular kernel, we guarantee that the CPU has been relinquished for a while. By limiting this delay, we prevent a regular high priority SCHED_FIFO thread stuck in a tight loop, from locking out the delayed Xenomai thread indefinitely.

### THREAD MANAGEMENT

- The default POSIX thread stack size was raised to `PTHREAD_STACK_MIN * 4`. The minimum stack size enforced by the `libcobalt` library is `PTHREAD_STACK_MIN + getpagesize()`.
- pthread_set_mode_np() now accepts PTHREAD_DISABLE_LOCKBREAK, which disallows breaking the scheduler lock. When unset (default case), a thread which holds the scheduler lock drops it temporarily while sleeping.

**Warning**
A Xenomai thread running with PTHREAD_DISABLE_LOCKBREAK and PTHREAD_LOCK_SCHED both set may enter a runaway loop when attempting to sleep on a resource or synchronization object (e.g. mutex, condition variable).

### PROCESS MANAGEMENT

- In a `fork()` $\rightarrow$ `exec()` sequence, all Cobalt API objects created by the child process before it calls `exec()` are automatically flushed by the Xenomai core.

### REAL-TIME SIGNALS

- Support for Xenomai real-time signals is available.

Cobalt replacements for `sigwait()`, `sigwaitinfo()`, `sigtimedwait()` and `kill()` are available. `pthread_kill()` was changed to send thread-directed Xenomai signals (instead of regular Linux signals).

Cobalt-based signals are stricly real-time. Both the sender and receiver sides work exclusively from the primary domain. However, only synchronous handling is available, with a thread waiting explicitly for a set of signals, using one of the `sigwait` calls. There is no support for asynchronous delivery of signals to handlers.

Signals from SIGRTMIN..SIGRTMAX are queued.

COBALT_DELAYMAX is defined as the maximum number of overruns which can be reported by the Cobalt core in the siginfo.si_overrun field, for any signal.

- `kill()` supports group signaling.

Cobalt's implementation of kill() behaves identically to the regular system call for non thread-directed signals (i.e. pid $\Leftarrow$ 0). In this case, the caller switches to secondary mode.

Otherwise, Cobalt first attempts to deliver a thread-directed signal to the thread whose kernel TID matches the given process id. If this thread is not waiting for signals at the time of the call, kill() then attempts to deliver the signal to a thread from the same process, which currently waits for a signal.

- `pthread_kill()` is a conforming call.

When Cobalt's replacement for `pthread_kill()` is invoked, a Xenomai-enabled caller is automatically switched to primary mode on its way to sending the signal, under the control of the real-time co-kernel. Otherwise, the caller keeps running under the control of the regular Linux kernel.

This behavior also applies to the new Cobalt-based replacement for the `kill()` system call.

TIMERS

- POSIX timers are no more dropped when the creator thread exits. However, they are dropped when the container process exits.
- If the thread signaled by a POSIX timer exits, the timer is automatically stopped at the first subsequent timeout which fails sending the notification. The timer lingers until it is deleted by a call to `timer_delete()` or when the process exits, whichever comes first.
- timer_settime() may be called from a regular thread (i.e. which is not Xenomai-enabled).
- EPERM is not returned anymore by POSIX timer calls. EINVAL is substituted in the corresponding situation.
- The clock to be used for enabling periodic timing for a thread may now be specified in the call to `pthread_make_periodic_np()`:

```
int pthread_make_periodic_np(pthread_t thread, clockid_t clk_id,
    struct timespec *starttp, struct timespec *periodtp);
```

MESSAGE QUEUES

- `mq_open()` default attributes align on the regular kernel values, i.e. 10 msg x 8192 bytes (instead of 128 x 128).
- `mq_send()` now enforces a maximum priority value for messages (32768).

**Real-time suitable STDIO** The former `include/rtdk.h` header is gone in Xenomai 3.x. Applications should include `include/stdio.h` instead. Similarly, the real-time suitable STDIO routines are now part of `libcobalt`.

# 7 Former Native interface changes (now Alchemy)

GENERAL

- The API calls supporting a wait operation may return the -EIDRM error code only when the target object was deleted while pending. Otherwise, passing a deleted object identifier to an API call will result in -EINVAL being returned.

INTERRUPT MANAGEMENT

- The `RT_INTR` API is gone. Please see the rationale for not handling low-level interrupt service code from user-space.

**Tip**
It is still possible to have the application wait for interrupt receipts, as explained here.

I/O REGIONS

- The RT_IOREGION API is gone. I/O memory resources should be controlled from a RTDM driver instead.

TIMING SERVICES

- `rt_timer_set_mode()` was removed. The clock resolution has become a per-process setting, which can be tuned by passing the `--alchemy-clock-resolution` switch on the command line.

**Tip**
Tick-based timing can be obtained by setting the resolution of the Alchemy clock for the application, here to one millisecond (the argument expresses a count nanoseconds per tick). As a result of this, all timeout and date values passed to Alchemy API calls will be interpreted as counts of milliseconds.

```
# xenomai-application --alchemy-clock-resolution=1000000
```

By default, the Alchemy API sets the clock resolution for the new process to one nanosecond (i.e. tickless, highest resolution).

- TM_INFINITE also means infinite wait with all `rt_*_until()` call forms.
- `rt_task_set_periodic()` does not suspend the target task anymore.

If a start date is specified, then `rt_task_wait_period()` will apply the initial delay.

Rationale: a periodic task has to call `rt_task_wait_period()` from within its work loop for sleeping until the next release point is reached. Therefore, in most cases, there is no point in waiting for the initial release point from a different location.

**Tip**
In the unusual case where you do need to have the target task wait for the initial release point outside of its periodic work loop, you can issue a call to `rt_task_wait_period()` separately, exclusively for this purpose, i.e.

```
                /* wait for the initial release point. */
                ret = rt_task_wait_period(&overruns);
                /* ...more preparation work... */
                for (;;) {
                        /* wait for the next release point. */
                        ret = rt_task_wait_period(&overruns);
                        /* ...do periodic work... */
                }
```

`rt_task_set_periodic()` still switches to primary as previously over Cobalt. However, it does not return -EWOULDBLOCK anymore.

- TM_ONESHOT was dropped, because the operation mode of the hardware timer has no meaning for the application. The core Xenomai system always operates the available timer chip in oneshot mode anyway.

**Tip**
A tickless clock has a period of one nanosecond.

MUTEXES

- For consistency with the standard glibc implementation, deleting a RT_MUTEX object in locked state is no more a valid operation.
- `rt_mutex_inquire()` does not return the count of waiters anymore.

Rationale: obtaining the current count of waiters only makes sense for debugging purpose. Keeping it in the API would introduce a significant overhead to maintain internal consistency.

The `owner` field of a RT_MUTEX_INFO structure now reports the owner's task handle, instead of its name. When the mutex is unlocked, a NULL handle is returned, which has the same meaning as a zero value in the former `locked` field.

CONDITION VARIABLES

- For consistency with the standard glibc implementation, deleting a RT_COND object currently pended by other tasks is no more a valid operation.
- Like `rt_mutex_inquire()`, `rt_cond_inquire()` does not return the count of waiting tasks anymore.

TASK MANAGEMENT

- `rt_task_notify()` and `rt_task_catch()` have been removed. They are meaningless in a userland-only context.
- As a consequence of the previous change, the T_NOSIG flag to `rt_task_set_mode()` was dropped in the same move.
- T_SUSP cannot be passed to rt_task_create() anymore.

Rationale: This behavior can be achieved by not calling `rt_task_start()` immediately after `rt_task_create()`, or by calling `rt_task_suspend()` before `rt_task_start()`.

- `rt_task_shadow()` now accepts T_LOCK, T_WARNSW.
- `rt_task_create()` now accepts T_LOCK, T_WARNSW and T_JOINABLE.
- The RT_TASK_INFO structure returned by `rt_task_inquire()` has changed:
  - fields `relpoint` and `cprio` have been removed, since the corresponding information is too short-lived to be valuable to the caller. The task's base priority is still available from the `prio` field.
  - other fields which represent runtime statistics are now avail from a core-specific `stat` field sub-structure.
- New `rt_task_send_until()`, `rt_task_receive_until()` calls are available, as variants of `rt_task_send()` and `rt_task_receive()` respectively, with absolute timeout specification.
- rt_task_receive() does not inherit the priority of the sender, although the requests will be queued by sender priority.

Instead, the application decides about the server priority instead of the real-time core applying implicit dynamic boosts.

- `rt_task_slice()` now returns -EINVAL if the caller currently holds the scheduler lock.

- T_CPU disappears from the `rt_task_create()` mode flags. The new `rt_task_set_affinity()` service is available for setting the CPU affinity of a task.
- `rt_task_sleep_until()` does not return -ETIMEDOUT anymore. Waiting for a date in the past blocks the caller indefinitely.

MESSAGE QUEUES

- `rt_queue_create()` mode bits updated:
  - Q_SHARED is implicit for all queues, defined as nop.
  - Q_DMA has become meaningless, and therefore was dropped. A driver should be implemented for delivering chunks of DMA-suitable memory.

HEAPS

- `rt_heap_create()` mode bits updated:
  - H_MAPPABLE has become meaningless.
  - H_SHARED is deprecated, but still wraps to H_SINGLE.
  - H_NONCACHED and H_DMA became meaningless.

**Tip**
If you need to allocate a chunk of DMA-suitable memory, then you should create a RTDM driver for this purpose.

- `rt_heap_alloc_until()` is a new call for waiting for a memory chunk, specifying an absolute timeout date.
- `rt_heap_inquire()` with the removal of H_DMA, phys_addr makes no sense anymore, removed.

ALARMS

- `rt_alarm_wait()` has been removed.

Rationale: An alarm handler can be specified to `rt_alarm_create()` instead.

- The RT_ALARM_INFO structure returned by `rt_alarm_inquire()` has changed:
  - field `expiration` has been removed, since the corresponding information is too short-lived to be valuable to the caller.

MESSAGE PIPES

- Writing to a message pipe is allowed from all contexts, including from alarm handlers.
- `rt_pipe_read_until()` is a new call for waiting for input from a pipe, specifying an absolute timeout date.

## 8 pSOS interface changes

MEMORY REGIONS

- `rn_create()` may return ERR_NOSEG if the region control block cannot be allocated internally.

SCHEDULING

- The emulator converts priority levels between the core POSIX and pSOS scales using normalization (pSOS → POSIX) and denormalization (POSIX → pSOS) handlers.

Applications may override the default priority normalization/denormalization handlers, by implementing the following routines.

```
int psos_task_normalize_priority(unsigned long psos_prio);

unsigned long psos_task_denormalize_priority(int core_prio);
```

Over Cobalt, the POSIX scale is extended to 257 levels, which allows to map pSOS over the POSIX scale 1:1, leaving normalization/denormalization handlers as no-ops by default.

## 9 VxWorks interface changes

TASK MANAGEMENT

- `WIND_*` status bits are synced to the user-visible TCB only as a result of a call to `taskTcb()` or `taskGetInfo()`.

As a consequence of this change, any reference to a user-visible TCB should be refreshed by calling `taskTcb()` anew, each time reading the `status` field is required.

SCHEDULING

- The emulator converts priority levels between the core POSIX and VxWorks scales using normalization (VxWorks → POSIX) and denormalization (POSIX → VxWorks) handlers.

Applications may override the default priority normalization/denormalization handlers, by implementing the following routines.

```
int wind_task_normalize_priority(int wind_prio);

int wind_task_denormalize_priority(int core_prio);
```