Migrating from Xenomai 2.x to 3.x

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Configuration

1.1 User programs and libraries

Changes in `xeno-config`

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.
- Over Cobalt, only **xeno-config --posix --ldflags** (or **--rtdm** as an alias) returns the proper linker flags to cause POSIX routines invoked by the application to be wrapped to their respective Xenomai implementation. No other API will imply such wrapping. For this reason, **--cobalt --ldflags** should be used for linking exclusively against the Cobalt library (i.e. `libcobalt.so`) **without** symbol wrapping. Conversely, mentioning **--posix** along with other API switches with **--ldflags** will cause POSIX symbol wrapping to take place, e.g. use **--posix --alchemy --ldflags** for mixed API support with POSIX symbol wrapping.
- Over *Mercury*, `--posix` and `--rtdm` are ignored placeholders, since the full POSIX API is available with the glibc and the threading library.
- `--[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`.
- `--ccld` retrieves a C compiler command suitable for linking a Xenomai 3.x application.
- `--info` retrieves the current system information, including the Xenomai release detected on the platform.

Auto-initialization

`--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(int *argcp, char *const **argvp)` routine manually, as part of its initialization process, *before* any real-time service is invoked.

This routine takes the addresses of the argument count and vector passed to the main() routine respectively. copperplate_init() handles the Xenomai options present in the argument vector, stripping them out, leaving only unprocessed options on return in the vector, updating the count accordingly.

`xeno-config` enables the Copperplate auto-init feature by default.

x86 vsyscall support

The `--enable-x86-sep` configuration switch was renamed to `--enable-x86-vsyscall` to fix a misnomer. This option should be left enabled (default), unless **linuxthreads** are used instead of **NPTL**.

1.2 Kernel parameters (Cobalt)

System parameters renamed

- xeno_hal.supported_cpus → xenomai.supported_cpus
- xeno_hal.clockfreq → xenomai.clockfreq
- xeno_hal.disable → xenomai.state=disabled
- xeno_hal.timerfreq → xenomai.timerfreq
- xeno_hal.cpufreq → xenomai.cpufreq
- xeno_nucleus.watchdog_timeout → xenomai.watchdog_timeout

- xeno_nucleus.xenomai_gid → xenomai.allowed_group
- xeno_nucleus.sysheap_size → xenomai.sysheap_size
- xeno_hal.smi (x86 only) → xenomai.smi
- xeno_hal.smi_mask (x86 only) → xenomai.smi_mask

Obsolete parameters dropped

- xeno_rtdm.tick_arg
- rtdm.devname_hashtab_size
- rtdm.protocol_hashtab_size

Rationale

Periodic timing is directly handled from the API layer in user-space. Cobalt kernel timing is tickless.

2 Getting the system state

When running Copperplate-based APIs (i.e. all but pure POSIX), 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 New FUSE-based registry interface

The Xenomai system state is now fully exported via a FUSE-based filesystem. The hierarchy of the Xenomai registry is organized as follows:

```
/mount-point              /* registry fs root, defaults to /var/run/xenomai */
 /user                    /* user name */
    /session              /* shared session name or anon@<pid> */
      /pid                /* application (main) pid */
        /skin             /* API name: alchemy/vxworks/psos/... */
          /family         /* object class (task, semaphore, ...) */
             { exported objects... }
      /system             /* session-wide information */
```

Each leaf entry under a session hierarchy is normally viewable, for retrieving the information attached to the corresponding object, such as its state, and/or value. There can be multiple sessions hosted under a single registry mount point.

The /system hierarchy provides information about the current state of the Xenomai core, aggregating data from all processes which belong to the parent session. Typically, the status of all threads and heaps created by the session can be retrieved.

The registry daemon is a companion tool managing exactly one registry mount point, which is specified by the --root option on the command line. This daemon is automatically spawned by the registry support code as required. There is normally no action required from users for managing it.

2.2 /proc/xenomai interface

The /proc/xenomai interface is still available when running over the Cobalt core, mainly for pure POSIX-based applications. The following changes took place:

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 3.1.2. Functionally speaking, only the former *master* time base remains, periodic timing is now controlled locally from the Xenomai libraries in user-space.

Xenomai 3.1.2 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).

The settings of existing clocks can be read from entries under the new clock/ hierarchy. Active timers for each clock can be read from entries under the new `timer/` hierarchy.

As a consequence of these changes:

- the information previously available from the `timer` entry is now obtained by reading `clock/coreclk`.
- the information previously available from `timerstat/master` is now obtained by reading `timer/coreclk`.

Core clock gravity
The gravity value for a Xenomai clock gives the amount of time by which the next timer shot should be anticipated. This is a static adjustment value, to account for the basic latency of the target system for responding to external events. Such latency may be introduced by hardware effects (e.g. bus or cache latency), or software issues (e.g. code running with interrupts disabled).

The clock gravity management departs from Xenomai 2.x as follows:

- different gravity values are applied, depending on which context a timer activates. This may be a real-time IRQ handler (*irq*), a RTDM driver task (*kernel*), or a Xenomai application thread running in user-space (*user*). Xenomai 2.x does not differentiate, only applying a global gravity value regardless of the activated context.
- in addition to the legacy `latency` file which now reports the *user* timer gravity (in nanoseconds), i.e. used for timers activating user-space threads, the full gravity triplet applied to timers running on the core clock can be accessed by reading `clock/coreclk` (also in nanoseconds).
- at reset, the *user* gravity for the core clock now represents the sum of the scheduling **and** hardware timer reprogramming time as a count of nanoseconds. This departs from Xenomai 2.x for which only the former was accounted for as a global gravity value, regardless of the target context for the timer.

The following command reports the current gravity triplet for the target system, along with the setup information for the core timer:

```
# cat xenomai/clock/coreclk
gravity: irq=848 kernel=8272 user=35303
devices: timer=decrementer, clock=timebase
 status: on+watchdog
  setup: 151
  ticks: 220862243033
```

Conversely, writing to this file manually changes the gravity values of the Xenomai core clock:

```
    /* change the user gravity (default) */
# echo 3000 > /proc/xenomai/clock/coreclck
    /* change the IRQ gravity */
# echo 1000i > /proc/xenomai/clock/coreclck
    /* change the user and kernel gravities */
# echo "2000u 1000k" > /proc/xenomai/clock/coreclck
```

`interfaces` removed
Only the POSIX and RTDM APIs remain implemented directly in kernel space, and are always present when the Cobalt core enabled in the configuration. 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` unconditionally.

`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-li` 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

Any application linked against `libcobalt` has its main thread attached to the real-time system automatically, this operation is called *auto-shadowing*. As a side-effect, the entire process's memory is locked, for current and future mappings (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** or **uClibc** 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.

5 RTDM interface changes

5.1 Files renamed

- 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

rtdm/rttesting.h → rtdm/testing.h

rtdm/rtipc.h → rtdm/ipc.h

5.2 Driver API

5.2.1 New device description model

Several changes have taken place in the device description passed to `rtdm_dev_register()` (i.e. `struct rtdm_device`). Aside of fixing consistency issues, the bulk of changes is aimed at narrowing the gap between the regular Linux device driver model and RTDM.

To this end, RTDM in Xenomai 3 shares the Linux namespace for named devices, which are now backed by common character device objects from the regular Linux device model. As a consequence of this, file descriptors obtained on named RTDM devices are regular file descriptors, visible from the `/proc/<pid>/fd` interface.

Named device description

The major change required for supporting this closer integration of RTDM into the regular Linux driver model involved splitting the device driver properties from the device instance definitions, which used to be combined in Xenomai 2.x into the `rtdm_device` descriptor.

Xenomai 2.x named device description

```
static struct rtdm_device foo_device0 = {
        .struct_version         =       RTDM_DEVICE_STRUCT_VER,
        .device_flags           =       RTDM_NAMED_DEVICE|RTDM_EXCLUSIVE,
        .device_id              =       0
        .context_size           =       sizeof(struct foo_context),
        .ops = {
                .open           =       foo_open,
                .ioctl_rt       =       foo_ioctl_rt,
                .ioctl_nrt      =       foo_ioctl_nrt,
                .close          =       foo_close,
        },
        .device_class           =       RTDM_CLASS_EXPERIMENTAL,
        .device_sub_class       =       RTDM_SUBCLASS_FOO,
        .profile_version        =       42,
        .device_name            =       "foo0",
        .driver_name            =       "foo driver",
        .driver_version         =       RTDM_DRIVER_VER(1, 0, 0),
        .peripheral_name        =       "Ultra-void IV board driver",
        .proc_name              =       device.device_name,
        .provider_name          =       "Whoever",
};

static struct rtdm_device foo_device1 = {
        .struct_version         =       RTDM_DEVICE_STRUCT_VER,
        .device_flags           =       RTDM_NAMED_DEVICE|RTDM_EXCLUSIVE,
        .device_id              =       1
        .context_size           =       sizeof(struct foo_context),
        .ops = {
                .open           =       foo_open,
                .ioctl_rt       =       foo_ioctl_rt,
                .ioctl_nrt      =       foo_ioctl_nrt,
                .close          =       foo_close,
        },
        .device_class           =       RTDM_CLASS_EXPERIMENTAL,
        .device_sub_class       =       RTDM_SUBCLASS_FOO,
        .profile_version        =       42,
        .device_name            =       "foo1",
        .device_data            =       NULL,
        .driver_name            =       "foo driver",
        .driver_version         =       RTDM_DRIVER_VER(1, 0, 0),
        .peripheral_name        =       "Ultra-void IV board driver",
        .proc_name              =       device.device_name,
        .provider_name          =       "Whoever",
};

foo0.device_data = &some_driver_data0;
ret = rtdm_dev_register(&foo0);
...
foo1.device_data = &some_driver_data1;
ret = rtdm_dev_register(&foo1);
```

The legacy description above would only create "virtual" device entries, private to the RTDM device namespace, with no visible counterparts into the Linux device namespace.

Xenomai 3.x named device description

```
static struct rtdm_driver foo_driver = {
        .profile_info           =       RTDM_PROFILE_INFO(foo,
                                                          RTDM_CLASS_EXPERIMENTAL,
                                                          RTDM_SUBCLASS_FOO,
                                                          42),
        .device_flags           =       RTDM_NAMED_DEVICE|RTDM_EXCLUSIVE,
        .device_count           =       2,
        .context_size           =       sizeof(struct foo_context),
        .ops = {
                .open           =       foo_open,
                .ioctl_rt       =       foo_ioctl_rt,
                .ioctl_nrt      =       foo_ioctl_nrt,
                .close          =       foo_close,
        },
};

static struct rtdm_device foo_devices[2] = {
        [ 0 ... 1 ] = {
                .driver = &foo_driver,
                .label = "foo%d",
        },
};

MODULE_VERSION("1.0.0");
MODULE_DESCRIPTION("Ultra-void IV board driver");
MODULE_AUTHOR'"Whoever");

foo_devices[0].device_data = &some_driver_data0;
ret = rtdm_dev_register(&foo_devices[0]);
...
foo_devices[1].device_data = &some_driver_data1;
ret = rtdm_dev_register(&foo_devices[1]);
```

The current description above will cause the device nodes /dev/rtdm/foo0 and /dev/rtdm/foo1 to be created in the Linux device namespace. Application may open these device nodes for interacting with the RTDM driver, as they would do with any regular *chrdev* driver.

Protocol device description

Similarly, the registration data for protocol devices have been changed to follow the new generic layout:

Xenomai 2.x protocol device description

```
static struct rtdm_device foo_device = {
        .struct_version =       RTDM_DEVICE_STRUCT_VER,
        .device_flags   =       RTDM_PROTOCOL_DEVICE,
        .context_size   =       sizeof(struct foo_context),
        .device_name    =       "foo",
        .protocol_family=       PF_FOO,
        .socket_type    =       SOCK_DGRAM,
        .socket_nrt     =       foo_socket,
        .ops = {
                .close_nrt      =       foo_close,
                .recvmsg_rt     =       foo_recvmsg,
                .sendmsg_rt     =       foo_sendmsg,
                .ioctl_rt       =       foo_ioctl,
                .ioctl_nrt      =       foo_ioctl,
                .read_rt        =       foo_read,
                .write_rt       =       foo_write,
                .select_bind    =       foo_select,
        },
        .device_class           =       RTDM_CLASS_EXPERIMENTAL,
        .device_sub_class       =       RTDM_SUBCLASS_FOO,
        .profile_version        =       1,
        .driver_name            =       "foo",
        .driver_version         =       RTDM_DRIVER_VER(1, 0, 0),
        .peripheral_name        =       "Unexpected protocol driver",
        .proc_name              =       device.device_name,
        .provider_name          =       "Whoever",
        .device_data            =       &some_driver_data,
};

ret = rtdm_dev_register(&foo_device);
...
```

Xenomai 3.x protocol device description

```
static struct rtdm_driver foo_driver = {
        .profile_info           =       RTDM_PROFILE_INFO(foo,
                                                          RTDM_CLASS_EXPERIMENTAL,
                                                          RTDM_SUBCLASS_FOO,
                                                          1),
        .device_flags           =       RTDM_PROTOCOL_DEVICE,
        .device_count           =       1,
        .context_size           =       sizeof(struct foo_context),
        .protocol_family        =       PF_FOO,
        .socket_type            =       SOCK_DGRAM,
        .ops = {
                .socket         =       foo_socket,
                .close          =       foo_close,
                .recvmsg_rt     =       foo_recvmsg,
                .sendmsg_rt     =       foo_sendmsg,
                .ioctl_rt       =       foo_ioctl,
                .ioctl_nrt      =       foo_ioctl,
                .read_rt        =       foo_read,
                .write_rt       =       foo_write,
                .select         =       foo_select,
        },
};

static struct rtdm_device foo_device = {
        .driver = &foo_driver,
        .label = "foo",
        .device_data = &some_driver_data,
};

ret = rtdm_dev_register(&foo_device);
...

MODULE_VERSION("1.0.0");
MODULE_DESCRIPTION("Unexpected protocol driver");
MODULE_AUTHOR'"Whoever");
```

- `.device_count` has been added to reflect the (maximum) number of device instances which may be managed by the driver. This information is used to dynamically reserve a range of major/minor numbers for named RTDM devices in the Linux device namespace, by a particular driver. Device minors are assigned to RTDM device instances in order of registration starting from minor #0, unless RTDM_FIXED_MINOR is present in the device flags. In the latter case, rtdm_device.minor is used verbatim by the RTDM core when registering the device.
- `.device_id` was removed from the device description, as the minor number it was most commonly holding is now available from a call to rtdm_fd_minor(). Drivers should use `.device_data` for storing private information attached to device instances.

- `.struct_version` was dropped, as it provided no additional feature to the standard module versioning scheme.
- `.proc_name` was dropped, as it is redundant with the device name. Above all, using a /proc information label different from the actual device name is unlikely to be a good idea.
- `.device_class`, `.device_sub_class` and `.profile_version` numbers have been grouped in a dedicated profile information descriptor (`struct rtdm_profile_info`), one **must** initialize using the `RTDM_PROFILE_INFO()` macro.
- `.driver_name` was dropped, as it adds no value to the plain module name (unless the module name is deliberately obfuscated, that is).
- `.peripheral_name` was dropped, as this information should be conveyed by MODULE_DESCRIPTION().
- `.provider_name` was dropped, as this information should be conveyed by MODULE_AUTHOR().
- `.driver_version` was dropped, as this information should be conveyed by MODULE_VERSION().

5.2.2 Introduction of file descriptors

Xenomai 3 introduces a file descriptor abstraction for RTDM drivers. For this reason, all RTDM driver handlers and services which used to receive a `user_info` opaque argument describing the calling context, now receive a `rtdm_fd` pointer standing for the target file descriptor for the operation.

As a consequence of this:

- The `rtdm_context_get/put()` call pair has been replaced by `rtdm_fd_get/put()`.
- Likewise, the `rtdm_context_lock/unlock()` call pair has been replaced by `rtdm_fd_lock/unlock()`.
- `rtdm_fd_to_private()` is available to fetch the context-private memory allocated by the driver for a particular RTDM file descriptor. Conversely, `rtdm_private_to_fd()` returns the file descriptor owning a particular context-private memory area.
- +rtdm_fd_minor() retrieves the minor number assigned to the current named device instance using its file descriptor.
- `xenomai/rtdm/open_files` and `xenomai/rtdm/fildes` now solely report file descriptors obtained using the driver-to-driver API. RTDM file descriptors obtained from applications appear under the regular /proc/<pid>/fd hierarchy. All RTDM file descriptors obtained by an application are automatically released when the latter exits.

Caution
Because RTDM file descriptors may be released and destroyed asynchronously, rtdm_fd_get() and rtdm_fd_lock() may return -EIDRM if a file descriptor fetched from some driver-private registry becomes stale prior to calling these services. Typically, this may happen if the descriptor is released from the →close() handler implemented by the driver. Therefore, make sure to always carefully check the return value of these services.

Note
Unlike Xenomai 2.x, RTDM file descriptors returned to Xenomai 3 applications fall within the regular Linux range. Each open RTDM connection is actually mapped over a regular file descriptor, which RTDM services from *libcobalt* recognize and handle.

5.2.3 Updated device operation descriptor

As visible from the previous illustrations, a few handlers have been moved to the device operation descriptor, some dropped, other renamed, mainly for the sake of consistency:

- `.select_bind` was renamed as `.select` in the device operation descriptor.

- `.open_rt` was dropped, and `.open_nrt` renamed as `.open`. Opening a named device instance always happens from secondary mode. In addition, the new handler is now part of the device operation descriptor `.ops`.

Rationale

Opening a device instance most often requires allocating resources managed by the Linux kernel (memory mappings, DMA etc), which is only available to regular calling contexts.

- Likewise, `.socket_rt` was dropped, and `.socket_nrt` renamed as `.socket`. Opening a protocol device instance always happens from secondary mode. In addition, the new handler is now part of the device operation descriptor `.ops`.
- As a consequence of the previous changes, `.close_rt` was dropped, and `.close_nrt` renamed as `.close`. Closing a device instance always happens from secondary mode.
- .open, .socket and .close handlers have become optional in Xenomai 3.x.
- The device operation descriptor `.ops` shows two new members, namely `.mmap` for handling memory mapping requests to the RTDM driver, and `get_unmapped_area`, mainly for supporting such memory mapping operations in MMU-less configurations. These handlers - named after the similar handlers defined in the regular file_operation descriptor - always operate from secondary mode on behalf of the calling task context, so that they may invoke regular kernel services safely.

Note

See the documentation in the Programming Reference Manual covering the device registration and operation handlers for a complete description.

5.2.4 Changes to RTDM services

- rtdm_dev_unregister() loses the poll_delay argument, and its return value. Instead, this service waits indefinitely for all ongoing connection to be drop prior to unregistering the device. The new prototype is therefore:

```
void rtdm_dev_unregister(struct rtdm_device *device);
```

Rationale

Drivers are most often not willing to deal with receiving a device busy condition from a module exit routine (which is the place devices should be unregistered from). Drivers which really want to deal with such condition should simply use module refcounting in their own code.

- rtdm_task_init() shall be called from secondary mode.

Rationale

Since Xenomai 3, rtdm_task_init() involves creating a regular kernel thread, which will be given real-time capabilities, such as running under the control of the Cobalt kernel. In order to invoke standard kernel services, rtdm_task_init() must be called from a regular Linux kernel context.

- rtdm_task_join() has been introduced to wait for termination of a RTDM task regardless of the caller's execution mode, which may be primary or secondary. In addition, rtdm_task_join() does not need to poll for such event unlike rtdm_task_join_nrt().

Rationale

rtdm_task_join() supersedes rtdm_task_join_nrt() feature-wise with less usage restrictions, therefore the latter has become pointless. It is therefore deprecated and will be phased out in the next release.

- A RTDM task cannot be forcibly removed from the scheduler by another thread for immediate deletion. Instead, the RTDM task is notified about a pending cancellation request, which it should act upon when detected. To this end, RTDM driver tasks should call the new `rtdm_task_should_stop()` service to detect such notification from their work loop, and exit accordingly.

Rationale

Since Xenomai 3, a RTDM task is based on a regular kernel thread with real-time capabilities when controlled by the Cobalt kernel. The Linux kernel requires kernel threads to exit at their earliest convenience upon notification, which therefore applies to RTDM tasks as well.

- `rtdm_task_set_period()` now accepts a start date for the periodic timeline. Zero can be passed to emulate the previous call form, setting the first release point when the first period after the current date elapses.
- `rtdm_task_wait_period()` now copies back the count of overruns into a user-provided variable if -ETIMEDOUT is returned. NULL can be passed to emulate the previous call form, discarding this information.
- Both `rtdm_task_set_period()` and `rtdm_task_wait_period()` may be invoked over a Cobalt thread context.
- RTDM_EXECUTE_ATOMICALLY() is deprecated and will be phased out in the next release. Drivers should prefer the newly introduced RTDM wait queues, or switch to the Cobalt-specific cobalt_atomic_enter/leave() call pair, depending on the use case.

Rationale

This construct is not portable to a native implementation of RTDM, and may be replaced by other means. The usage patterns of RTDM_EXECUTE_ATOMICALLY() used to be:

- somewhat abusing the big nucleus lock (i.e. nklock) grabbed by RTDM_EXECUTE_ATOMICALLY(), for serializing access to a section that should be given its own lock instead, improving concurrency in the same move. Such section does not call services from the Xenomai core, and does NOT specifically require the nucleus lock to be held. In this case, a RTDM lock (rtdm_lock_t) should be used to protect the section instead of RTDM_EXECUTE_ATOMICALLY().
- protecting a section which calls into the Xenomai core, which exhibits one or more of the following characteristics:
  - Some callee within the section may require the nucleus lock to be held on entry (e.g. Cobalt registry lookup). In what has to be a Cobalt-specific case, the new cobalt_atomic_enter/leave() call pair can replace RTDM_EXECUTE_ATOMICALLY(). However, this construct remains by definition non-portable to Mercury.
  - A set-condition-and-wakeup pattern has to be carried out atomically. In this case, RTDM_EXECUTE_ATOMICALLY() can be replaced by the wakeup side of a RTDM wait queue introduced in Xenomai 3 (e.g. rtdm_waitqueue_signal/broadcast()).
  - A test-condition-and-wait pattern has to be carried out atomically. In this case, RTDM_EXECUTE_ATOMICALLY() can be replaced by the wait side of a RTDM wait queue introduced in Xenomai 3 (e.g. rtdm_wait_condition()).

Please refer to kernel/drivers/ipc/iddp.c for an illustration of the RTDM wait queue usage.

- rtdm_irq_request/free() and rtdm_irq_enable/disable() call pairs must be called from a Linux task context, which is a restriction that did not exist previously with Xenomai 2.x.

Rationale

Recent evolutions of the Linux kernel with respect to IRQ management involve complex processing for basic operations (e.g. enabling/disabling the interrupt line) with some interrupt types like MSI. Such processing cannot be made dual-kernel safe at a reasonable cost, without encurring measurable latency or significant code updates in the kernel.

Since allocating, releasing, enabling or disabling real-time interrupts is most commonly done from driver initialization/-cleanup context already, the Cobalt core has simply inherited those requirements from the Linux kernel.

- The leading *user_info* argument to rtdm_munmap() has been removed.

Rationale

With the introduction of RTDM file descriptors (see below) replacing all *user_info* context pointers, this argument has become irrelevant, since this operation is not related to any file descriptor, but rather to the current address space.

The new prototype for this routine is therefore

```
int rtdm_munmap(void *ptr, size_t len);
```

- Additional memory mapping calls

The new following routines are available to RTDM drivers, for mapping memory over a user address space. They are intended to be called from a →mmap() handler:

- rtdm_mmap_kmem() for mapping logical kernel memory (i.e. having a direct physical mapping).
- rtdm_mmap_vmem() for mapping purely virtual memory (i.e. with no direct physical mapping).
- rtdm_mmap_iomem() for mapping I/O memory.

```
static int foo_mmap(struct rtdm_fd *fd, struct vm_area_struct *vma)
{
        ...
        switch (memory_type) {
        case MEM_PHYSICAL:
                ret = rtdm_mmap_iomem(vma, addr);
                break;
        case MEM_LOGICAL:
                ret = rtdm_mmap_kmem(vma, (void *)addr);
                break;
        case MEM_VIRTUAL:
                ret = rtdm_mmap_vmem(vma, (void *)addr);
                break;
        default:
                return -EINVAL;
        }
        ...
}
```

- the rtdm_nrtsig API has changed, the rtdm_nrtsig_init() function no longer returns errors, it has the void return type. The rtdm_nrtsig_t type has changed from an integer to a structure. In consequence, the nrtsig handler first argument is now a pointer to the rtdm_nrtsig_t structure.

Rationale

Recent versions of the I-pipe patch support an ipipe_post_work_root() service, which has the advantage over the VIRQ support, that it does not require allocating one different VIRQ for each handler. As a consequence drivers may use as many rtdm_nrtsig_t structures as they like, there is no chance of running out of VIRQs.

```
The new relevant prototypes are therefore:
```

```
typedef void (*rtdm_nrtsig_handler_t)(rtdm_nrtsig_t *nrt_sig, void *arg);

void rtdm_nrtsig_init(rtdm_nrtsig_t *nrt_sig,
     rtdm_nrtsig_handler_t handler, void *arg);
```

- a new rtdm_schedule_nrt_work() was added to allow scheduling a Linux work queue from primary mode.

Rationale

Scheduling a Linux workqueue maybe a convenient way for adriver to recover for an error which requires synchronization with Linux. Typically, recovering from a PCI error may involve accessing the PCI config space, which requires access to a Linux spinlock so can not be done from primary mode.

`The prototype of this new service is:`

```
void rtdm_schedule_nrt_work(struct work_struct *work);
```

5.2.5 Adaptive syscalls

`ioctl()`, `read()`, `write()`, `recvmsg()` and `sendmsg()` have become conforming RTDM calls, which means that Xenomai threads running over the Cobalt core will be automatically switched to primary mode prior to running the driver handler for the corresponding request.

Rationale

Real-time handlers from RTDM drivers serve time-critical requests by definition, which makes them preferred targets of adaptive calls over non real-time handlers.

Note
This behavior departs from Xenomai 2.x, which would run the call from the originating context instead (e.g. `ioctl_nrt()` would be fired for a caller running in secondary mode, and conversely `ioctl_rt()` would be called for a request issued from primary mode).

Tip
RTDM drivers implementing differentiated `ioctl()` support for both domains should serve all real-time only requests from `ioctl_rt()`, returning `-ENOSYS` for any unrecognized request, which will cause the adaptive switch to take place automatically to the `ioctl_nrt()` handler. The `ioctl_nrt()` should then implement all requests which may be valid from the regular Linux domain exclusively.

5.3 Application interface

Unlike with Xenomai 2.x, named RTDM device nodes in Xenomai 3 are visible from the Linux device namespace. These nodes are automatically created by the *hotplug* kernel facility. Application must open these device nodes for interacting with RTDM drivers, as they would do with any regular *chrdev* driver.

All RTDM device nodes are created under the `rtdm/` sub-root from the standard `/dev` hierarchy, to eliminate potential name clashes with standard drivers.

Important
Enabling DEVTMPFS in the target kernel is recommended so that the standard `/dev` tree immediately reflects updates to the RTDM device namespace. You may want to enable CONFIG_DEVTMPFS and CONFIG_DEVTMPFS_MOUNT.

Opening a named device instance with Xenomai 2.x

```
fd = open("foo", O_RDWR);
   or
fd = open("/dev/foo", O_RDWR);
```

Opening a named device instance with Xenomai 3

```
fd = open("/dev/rtdm/foo", O_RDWR);
```

Tip
Applications can enable the CONFIG_XENO_OPT_RTDM_COMPAT_DEVNODE option in the kernel configuration to enable legacy pathnames for named RTDM devices. This compatibility option allows applications to open named RTDM devices using the legacy naming scheme used by Xenomai 2.x.

5.3.1 Retrieving device information

Device information can be retrieved via *sysfs*, instead of *procfs* as with Xenomai 2.x. As a result of this change, `/proc/xenomai/rtd` disappeared entirely. Instead, the RTDM device information can now be reached as follows:

- /sys/devices/virtual/rtdm contains entries for all RTDM devices present in the system (including named and protocol device types). This directory is aliased to /sys/class/rtdm.
- each /sys/devices/virtual/rtdm/<device-name> directory gives access to device information, available from virtual files:
  - reading `profile` returns the class and subclass ids.
  - reading `refcount` returns the current count of outstanding connections to the device driver.
  - reading `flags` returns the device flags as defined by the device driver.
  - reading `type` returns the device type (*named* or *protocol*).

5.4 Inter-Driver API

The legacy (and redundant) rt_dev_*() API for calling the I/O services exposed by a RTDM driver from another driver was dropped, in favour of a direct use of the existing rtdm_*() API in kernel space. For instance, calls to `rt_dev_open()` should be converted to `rtdm_open()`, `rt_dev_socket()` to `rtdm_socket()` and so on.

Rationale

Having two APIs for exactly the same purpose is uselessly confusing, particularly for kernel programming. Since the user-space version of the rt_dev_*() API was also dropped in favor of the regular POSIX I/O calls exposed by `libcobalt`, the choice was made to retain the most straightforward naming for the RTDM-to-RTDM API, keeping the `rtdm_` prefix.

6 Analogy interface changes

6.1 Files renamed

- DAQ drivers in kernel space now pull all Analogy core header files from <rtdm/analogy/*.h>. In addition:

analogy/analogy_driver.h → rtdm/analogy/driver.h

analogy/driver.h → rtdm/analogy/driver.h

analogy/analogy.h → rtdm/analogy.h

- DAQ drivers in kernel space should include <rtdm/analogy/device.h> instead of <rtdm/analogy/driver.h>.
- Applications need to include only a single file for pulling all routine declarations and constant definitions required for invoking the Analogy services from user-space, namely <rtdm/analogy.h>, i.e.

analogy/types.h analogy/command.h analogy/device.h analogy/subdevice.h analogy/instruction.h analogy/ioctl.h → all files merged into rtdm/analogy.h

As a consequence of these changes, the former include/analogy/ file tree has been entirely removed.

7 RTnet changes

RTnet is integrated into Xenomai 3, but some of its behaviour and interfaces were changed in an attempt to simplify it.

- a network driver kernel module can not be unloaded as long as the network interface it implements is up
- the RTnet drivers API changed, to make it simpler, and closer to the mainline API
  - module refcounting is now automatically done by the stack, no call is necessary to RTNET_SET_MODULE_OWNER, RTNET_MOD_INC_USE_COUNT, RTNET_MOD_DEC_USE_COUNT
  - per-driver skb receive pools were removed from drivers, they are now handled by the RTnet stack. In consequence, drivers now need to pass an additional argument to the rt_alloc_etherdev() service: the number of buffers in the pool. The new prototype is:

```
struct rtnet_device *rt_alloc_etherdev(unsigned sizeof_priv, unsigned rx_pool_size);
```

- in consequence, any explicit call to rtskb_pool_init() can be removed. In addition, drivers should now use the rtnetdev_alloc_rtskb() to allocate buffers from the network device receive pool; much like its counterpart netdev_alloc_skb(), it takes as first argument a pointer to a network device structure. Its prototype is:

```
struct rtskb *rtnetdev_alloc_rtskb(struct rtnet_device *dev, unsigned int size);
```

- for driver which wish to explicitly handle skb pools, the signature of rtskb_pool_init has changed: it takes an additional pointer to a structure containing callbacks called when the first buffer is allocated and when the last buffer is returned, so that the rtskb_pool() can implicitly lock a parent structure. The new prototype is:

```
struct rtskb_pool_lock_ops {
    int (*trylock)(void *cookie);
    void (*unlock)(void *cookie);
};

unsigned int rtskb_pool_init(struct rtskb_pool *pool,
                             unsigned int initial_size,
                             const struct rtskb_pool_lock_ops *lock_ops,
                             void *lock_cookie);
```

- for the typical case where an skb pool locks the containing module, the function rtskb_module_pool_init() was added which has the same interface as the old rtskb_poll_init() function. Its prototype is:

```
unsigned int rtskb_module_pool_init(struct rtskb_pool *pool,
                                    unsigned int initial_size);
```

- in order to ease the port of recent drivers, the following services were added, which work much like their Linux counterpart: rtnetdev_priv(), rtdev_emerg(), rtdev_alert(), rtdev_crit(), rtdev_err(), rtdev_warn(), rtdev_notice(), rtdev_info(), rtdev_dbg(), rtdev_vdbg(), RTDEV_TX_OK, RTDEV_TX_BUSY, rtskb_tx_timestamp(). Their declarations are equivalent to:

```
#define RTDEV_TX_OK      0
#define RTDEV_TX_BUSY    1

void *rtndev_priv(struct rtnet_device *dev);

void rtdev_emerg(struct rntet_device *dev, const char *format, ...);
void rtdev_alert(struct rntet_device *dev, const char *format, ...);
void rtdev_crit(struct rntet_device *dev, const char *format, ...);
void rtdev_err(struct rntet_device *dev, const char *format, ...);
void rtdev_warn(struct rntet_device *dev, const char *format, ...);
void rtdev_notice(struct rntet_device *dev, const char *format, ...);
void rtdev_info(struct rntet_device *dev, const char *format, ...);
void rtdev_dbg(struct rntet_device *dev, const char *format, ...);
void rtdev_vdbg(struct rntet_device *dev, const char *format, ...);

void rtskb_tx_timestamp(struct rtskb *skb);
```

8 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 longer 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.

8.1 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, particularly when the interrupt line is shared. 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 driver code in kernel space, where it naturally belongs to.

Tip

This said, in the seldom cases where running a device driver in user-space is the best option, one may rely on the RTDM-based UDD framework shipped with Xenomai 3. UDD stands for *User-space Device Driver*, enabling interrupt control and I/O memory access interfaces to applications in a safe manner. It is reminiscent of the UIO framework available with the Linux kernel, adapted to the dual kernel Cobalt environment.

8.2 Scheduling

- Cobalt implements the following POSIX.1-2001 services not present in Xenomai 2.x: `sched_setscheduler(2)`, `sched_gets`
- The `SCHED_FIFO`, `SCHED_RR`, `SCHED_SPORADIC` and `SCHED_TP` classes now support up to 256 priority levels, instead of 99 as previously with Xenomai 2.x. However, `sched_get_priority_max(2)` still returns 99. Only the Cobalt extended call forms (e.g. `pthread_attr_setschedparam_ex()`, `pthread_create_ex()`) recognize these additional levels.
- The new `sched_get_priority_min_ex()` and `sched_get_priority_max_ex()` services should be used for querying the static priority range of Cobalt policies.
- `pthread_setschedparam(3)` may cause a secondary mode switch for the caller, but will 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(3)` 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, triggering a call to `pthread_setschedparam(3)` 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, `pthread_setschedparam(3)` is not async-safe, which makes the former method fragile.

Conversely, a thread which calls `pthread_setschedparam(3)` 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_SCHE` 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(3)` 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_SCH` 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(2) now delays the caller for a short while **only in case** no context switch happened as a result of the manual round-robin. The delay ends next time the regular Linux kernel switches tasks, or a kernel (virtual) tick has elapsed (TICK_NSEC), whichever comes first.

  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.

Rationale

In most case, it is unwanted that sched_yield(2) 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 manual round-robin takes place between both threads, despite the execution mode mismatch. By limiting the incurred delay, we prevent a regular high priority SCHED_FIFO thread stuck in a tight loop, from locking out the delayed Xenomai thread indefinitely.

8.3 Thread management

- The minimum and default stack size is set to `max(64k, PTHREAD_STACK_MIN)`.
- pthread_set_name_np() has been renamed to pthread_setname_np() with the same arguments, to conform with the GNU extension equivalent.
- pthread_set_mode_np() has been renamed to pthread_setmode_np() for naming consistency with pthread_setname_np(). In addition, the call introduces the PTHREAD_DISABLE_LOCKBREAK mode flag, which disallows breaking the scheduler lock.

  When unset (default case), a thread which holds the scheduler lock drops it temporarily while sleeping. When set, any attempt to block while holding the scheduler lock will cause a break condition to be immediately raised, with the caller receiving EINTR.

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 or condition variable).

8.4 Semaphores

- With Cobalt, sem_wait(3), sem_trywait(3), sem_timedwait(3), and sem_post(3) have gained fast acquisition/release operations not requiring any system call, unless a contention exists on the resource. As a consequence, those services may not systematically switch callers executing in relaxed mode to real-time mode, unlike with Xenomai 2.x.

8.5 Process management

- In a `fork(2)` → `exec(2)` sequence, all Cobalt API objects created by the child process before it calls `exec(2)` are automatically flushed by the Xenomai core.

8.6 Real-time signals

- Support for Xenomai real-time signals is available.

Cobalt replacements for `sigwait(3)`, `sigwaitinfo(2)`, `sigtimedwait(2)`, `sigqueue(3)` and `kill(2)` are available. `pthread_kill(3)` 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. For this reason, there is no provision in the Cobalt API for masking signals, as Cobalt signals are implicitly blocked for a thread until the latter invokes one of the `sigwait` calls.

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.

- Cobalt's `kill(2)` implementation supports group signaling.

Cobalt's implementation of kill(2) behaves identically to the regular system call for non thread-directed signals (i.e. pid ⇐ 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(2) then attempts to deliver the signal to a thread from the same process, which currently waits for a signal.

- `pthread_kill(3)` is a conforming call.

When Cobalt's replacement for `pthread_kill(3)` 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(2)` system call.

8.7 Timers

- POSIX timers are no longer 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(2)` or when the process exits, whichever comes first.
- timer_settime(2) 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.
- Cobalt replacements for `timerfd_create(2)`, `timerfd_settime(2)` and `timerfd_gettime(2)` have been introduced. The implementation delivers I/O notifications to RTDM file descriptors upon Cobalt-originated real-time signals.
- `pthread_make_periodic_np()` and `pthread_wait_np()` have been removed from the API.

Rationale

With the introduction of services to support real-time signals, those two non-portable calls have become redundant. Instead, Cobalt-based applications should set up a periodic timer using the `timer_create(2)+timer_settime(2)` call pair, then wait for release points via `sigwaitinfo(2)`. Overruns can be detected by looking at the siginfo.si_overrun field.

Alternatively, applications may obtain a file descriptor referring to a Cobalt timer via the `timerfd_create(2)` call, and `read(2)` from it to wait for timeouts.

In addition, applications may include a timer in a synchronous multiplexing operation involving other event sources, by passing a file descriptor returned by the `timerfd_create(2)` service to a `select(2)` call.

Tip
A limited emulation of the pthread_make_periodic_np() and pthread_wait_np() calls is available from the Transition Kit.

8.8 Clocks

- The internal identifier of CLOCK_HOST_REALTIME has changed from 42 to 8.

Caution
This information should normally remain opaque to applications, as it is subject to change with ABI revisions.

8.9 Message queues

- `mq_open(3)` default attributes align on the regular kernel values, i.e. 10 msg x 8192 bytes (instead of 128 x 128).
- `mq_send(3)` now enforces a maximum priority value for messages (32768).

8.10 POSIX I/O services

- A Cobalt replacement for mmap(2) has been introduced. The implementation invokes the .mmap operation handler from the appropriate RTDM driver the file descriptor is connected to.
- A Cobalt replacement for fcntl(2) has been introduced. The implementation currently deals with the O_NONBLOCK flag exclusively.
- Cobalt's select(2) service is not automatically restarted anymore upon Linux signal receipt, conforming to the POSIX standard (see man signal(7)). In such an event, -1 is returned and errno is set to EINTR.
- 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`.

9 Alchemy interface (formerly *native API*)

9.1 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.

9.2 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.

9.3 I/O regions

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

Tip
UDD provides a simple way to implement mini-drivers exposing any kind of memory regions to applications in user-space, via Cobalt's mmap(2) call.

9.4 Timing services

- `rt_timer_tsc()`, `rt_timer_ns2tsc()` and `rt_timer_tsc2ns()` have been removed from the API.

Rationale

Due to the accumulation of rounding errors, using raw timestamp values from the underlying clock source hardware for measuring long timespans may yield (increasingly) wrong results.

Either we guarantee stable computations with counts of nanoseconds from within the application, or with raw timestamps instead, regardless of the clock source frequency, but we can't provide such guarantee for both. From an API standpoint, the nanosecond unit is definitely the best option as the meaning won't vary between clock sources.

Avoiding the overhead of the tsc→ns conversion as a justification to use raw TSC counts does not fly anymore, as all architectures implement fast arithmetics for this operation over Cobalt, and Mercury's (virtual) timestamp counter is actually mapped over CLOCK_MONOTONIC.

Tip
Alchemy users should measure timespans (or get timestamps) as counts of nanoseconds as returned by rt_timer_read() instead.

- `rt_timer_inquire()` has a void return type, instead of always returning zero as previously. As a consequence of the previously documented change regarding TSC values, the current TSC count is no more returned into the RT_TIMER_INFO structure.
- `rt_timer_set_mode()` is obsolete. The clock resolution has become a per-process setting, which should be set using 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_perioc`
will apply the initial delay.

Rationale

A periodic Alchemy task has to call `rt_task_wait_period()` from within its work loop for sleeping until the next release point is reached. Since waiting for the initial and subsequent release points will most often happen at the same code location in the application, the semantics of rt_task_set_periodic() can be simplified so that only rt_task_wait_period() may block the caller.

Tip

In the unusual case where you do need to have the current 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... */
}
```

However, this work around won't work if the caller is not the target task of rt_task_set_periodic(), which is fortunately unusual for most applications.

`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. A tickless clock has a period of one nanosecond.
- Unlike with Xenomai 2.x, the target task to `rt_task_set_periodic()` must be local to the current process.

Tip

A limited emulation of the deprecated rt_task_set_periodic() behavior is available from the Transition Kit.

9.5 Mutexes

- For consistency with the standard glibc implementation, deleting a RT_MUTEX object in locked state is no longer 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.

9.6 Condition variables

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

9.7 Events

- Event flags (RT_EVENT) are represented by a regular integer, instead of a long integer as with Xenomai 2.x. This change impacts the following calls:
  - rt_event_create()
  - rt_event_signal()
  - rt_event_clear()
  - rt_event_wait()
  - rt_event_wait_until()

Rationale

Using long integers for representing event bit masks potentially creates a portability issue for applications between 32 and 64bit CPU architectures. This issue is solved by using 32bit integers on 32/64 bit machines, which is normally more than enough for encoding the set of events received by a single RT_EVENT object.

Tip
These changes are covered by the Transition Kit.

9.8 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() or rt_task_spawn() anymore.
- T_FPU is obsolete. FPU management is automatically enabled for Alchemy tasks if the hardware supports it, disabled otherwise.

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.
  - new field `pid` represents the Linux kernel task identifier for the Alchemy task, as obtained from syscall(__NR_gettid).
  - 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, or attempts to change the round-robin settings of a thread which does not belong to the current process.
- 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.

Tip
An emulation of rt_task_create() and rt_task_spawn() accepting the deprecated flags is available from the Transition Kit.

- `rt_task_sleep_until()` does not return -ETIMEDOUT anymore. Waiting for a date in the past blocks the caller indefinitely.

9.9 Message queues

- As Alchemy-based applications run in user-space, the following `rt_queue_create()` mode bits from the former *native* API are obsolete:
  - Q_SHARED
  - Q_DMA

Tip
Placeholders for those deprecated definitions are available from the Transition Kit.

9.10 Heaps

- As Alchemy-based applications run in user-space, the following `rt_heap_create()` mode bits from the former *native* API are obsolete:
  - H_MAPPABLE
  - H_SHARED
  - H_NONCACHED
  - H_DMA

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.
- with the removal of H_DMA, returning a physical address (phys_addr) in `rt_heap_inquire()` does not apply anymore.

Tip
Placeholders for those deprecated definitions are available from the Transition Kit.

9.11 Alarms

- `rt_alarm_wait()` has been removed.

> **Rationale**
>
> An alarm handler can be passed 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.
  - field `active` has been added, to reflect the current state of the alarm object. If non-zero, the alarm is enabled (i.e. started).

> **Tip**
>
> An emulation of rt_alarm_wait() is available from the Transition Kit.

9.12 Message pipes

- `rt_pipe_create()` now returns the minor number assigned to the connection, matching the /dev/rtp<minor> device usable by the regular threads. As a consequence of this, any return value higher or equal to zero denotes a successful operation, a negative return denotes an error.
- 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.

10 pSOS interface changes

10.1 Memory regions

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

10.2 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.

11 VxWorks interface changes

11.1 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.

11.2 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);
```

12 Using the Transition Kit

Xenomai 2 applications in user-space may use a library and a set of compatibility headers, aimed at easing the process of transitioning to Xenomai 3.

Enabling this compatibility layer is done via passing specific compilation and linker flags when building the application. `xeno-config` can retrieve those flags using the `--cflags` and `--ldflags` switches as usual, with the addition of the `--compat` flag. Alternatively, passing the `--[skin=]native` switch as to `xeno-config` implicitly turns on the compatibility mode for the Alchemy API.

Note
The transition kit does not currently cover *all* the changes introduced in Xenomai 3 yet, but a significant subset of them nevertheless.

A typical Makefile fragment implicitly turning on backward compatibility

```
PREFIX := /usr/xenomai
CONFIG_CMD := $(PREFIX)/bin/xeno-config
CFLAGS= $(shell $(CONFIG_CMD) --skin=native --cflags) -g
LDFLAGS= $(shell $(CONFIG_CMD) --skin=native --ldflags)
CC = $(shell $(CONFIG_CMD) --cc)
```

Another example for using with the POSIX API

```
PREFIX := /usr/xenomai
CONFIG_CMD := $(PREFIX)/bin/xeno-config
CFLAGS= $(shell $(CONFIG_CMD) --skin=posix --cflags --compat) -g
LDFLAGS= $(shell $(CONFIG_CMD) --skin=posix --ldflags --compat)
CC = $(shell $(CONFIG_CMD) --cc)
```