A Tour of the Native API

Purpose of this document

This document is aimed at providing a general view of the reforged native API which comes with RTAI/fusion. Newcomers should find design information describing the logic behind this interface. This document should be a useful complement to the API reference manual.

What is this API for?

Since RTAI/fusion is intrinsically API-agnostic, it can run various kinds of interface flavours, for instance VxWorks, pSOS+, uITRON or VRTX-like emulators, mimicking these traditional RTOS APIs, usually for porting legacy applications. But you may not necessarily want to build your brand new application over those emulators because compatibility with such interfaces is not an issue; in contrast, you may just want to use a programming interface which leverages all the capabilities of the underlying real-time core, and makes full use of its high integration level with the GNU/Linux environment.

To this end, RTAI/fusion brings its own API of choice any application can use: this interface is simply called the *native* API, to differentiate from others, which have been defined outside of the RTAI realm.

Reforging the RTAI API

Since RTAI/fusion is about experimenting new ways of providing a stable and developer-friendly real-time system in a GNU/Linux environment, the issue of the native API it will export to applications is crucial.

A compact API

RTAI/fusion was a good opportunity to reforge a more compact native API, hopefully still providing a comfortable programming environment, in less than a hundred of distinct services (at the time those lines are written, the total number of system calls is 90).

To this end, a strict compatibility with RTAI/classic's API has not been pursued, so that complete freedom has been granted for designing the new interface. In other words, don't expect RTAI/fusion's reforged API to be 100% syntactically compatible with RTAI/classic's one: full syntactical compatibility is the business of emulators, and this API is not one of those. However, there are still a lot of common or at least close semantics between both APIs, so that porting applications between them should remain a fairly manageable task.

Orthogonality

By RTAI/classic's standards, RTAI/fusion's native API may seem rather spartan, but actually, it is not. In fact, all the fundamental services are indeed there, but an extreme

care has been taken to avoid multiple variations of semantically and functionally close services, that would end up being only differentiated by varying names or details in prototype. A typical example of this concerns the message passing support: whilst RTAI/classic provides three equivalent services for synchronous message passing between tasks (*i.e. rt_Send(), rt_rpcx_timed(), rt_rpc_timed()*), RTAI/fusion provides a single unified one (*i.e. rt_task_send()*).

The rationale behind such design choice is plain simple: we feel that the level of freedom brought to the application developer by an API does not depend on the number of available system calls, but on the capacity she is given to identify, pick and combine the existing services in a non-ambiguous and reliable fashion. For this reason, those services must be orthogonal, always have a straightforward purpose, and rely on rock-solid basic mechanisms independent from the general flavour or *window-dressing* exposed by the API.

Context-independence

Even if the preferred execution environment for RTAI/fusion applications is user-space context, there might still be a few cases where running some of the real-time code embodied into kernel modules is required, especially with legacy systems or very low-end hardware. For this reason, RTAI/fusion's native API provides the same set of real-time services in a seamless manner to application regardless of their execution space. Additionally, some applications may need real-time activities in both spaces to cooperate, therefore a special care has been taken to allow the latter to work on the exact same set of API objects.

Seamless topology

What an apparently complex definition for a rather simple idea! It just means that if we want to be able to use the same system call for performing an action without bothering about the location (e.g. the execution space) of the target object (e.g. a task, a semaphore etc.), we need to hide the nitty gritty details of how to reach such object into some opaque, normalized and shareable object identifier.

To this end, each category of services in RTAI/fusion's native API defines a uniform descriptor type to represent the objects it manages, which can be used either in kernel or user-space contexts indifferently. For instance, a task will always be represented by a *RT_TASK* descriptor, a message queue by a *RT_QUEUE* descriptor, and a semaphore by a *RT_SEM* descriptor, regardless of the execution space from which they are used. Such descriptors can be transparently shared between execution spaces, so that services can be called for objects which have been created from the opposite space; e.g. a semaphore created by a user-space application can be posted by a real-time interrupt handler in kernel space and conversely.

Then, we need to provide a mean to applications for finding those descriptors on a system-wide basis, preferably using a free-form symbolic name as the search key, since manipulating abstract information is easier than working with cryptic tokens.

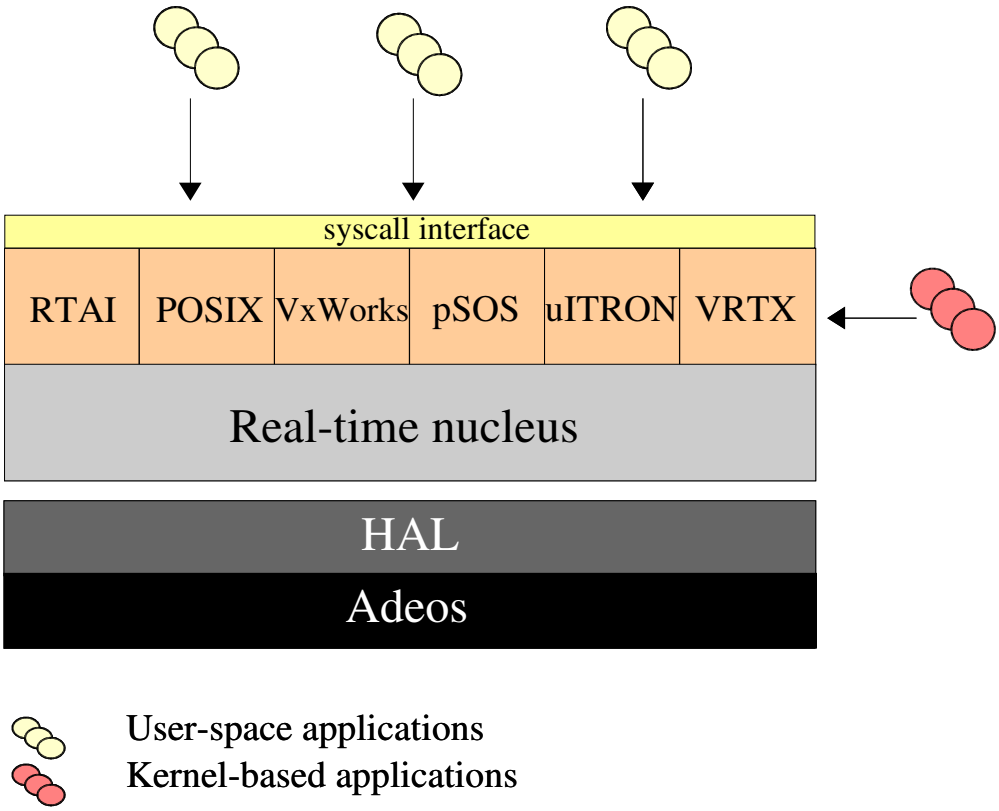
RTAI/fusion's answer to this is a unified registry, which allows all categories of real-time services to index each object they create on a unique symbolic search key defined by the application, and conversely, provides the application a mean to retrieve the unified descriptor associated with any registered object upon request.

By the time those lines are written, the native API is not network-distributable, whilst RTAI/classic's API fully supports multi-node requests, through its NETRPC component. This said, you will likely see how the combination of the unified descriptor scheme, an extended registry interfaced with a network name service, and real-time network layers such as RTNet will be able to provide well-integrated and robust multi-node capabilities to RTAI/fusion's native API.

The design of the native API

Architecture

A key characteristic of the native API stands in the fact that such API is an optional part of the RTAI/fusion system, just like any other real-time *window-dressing* interface also available (e.g. VxWorks, pSOS+, VRTX or uITRON). Like each of these interfaces, it is based on the same real-time nanokernel (aka the *nucleus*) underlying RTAI/fusion, which provides a set of generic RTOS services one can specialize to build its API flavour of choice. The figure below illustrates such layered approach:



As the figure above shows, the native RTAI interface has no special privileges compared to others, and is entirely based on the generic core's public interface for a simple reason: this is the only way to make sure that significant optimizations and bugfixes occurring in the core are immediately inherited by the outer APIs, without any additional risks of regression. Moreover, the services provided by the real-time nucleus end up being ironed by multiple client APIs exercising it in various ways, which increases the overall robustness of the whole system.

Calling the native API

The native API services are implemented in a kernel module named *rtai_native.ko* in the standard RTAI/fusion distribution. These services can be called either directly from kernel space using inter-module function calls, or from user-space context, through an interface library (*librtai.so*) issuing the proper system calls, which ends up running the

actual service provided by the kernel module.

Categories of services

RTAI/fusion's native API defines six major categories of services. Each category defines a set of system calls, most of which being available seamlessly in both kernel and user-space execution contexts.

- Task management. This category defines the set of services related to task scheduling and general management. An application needs these system calls to create tasks and control their behaviour. All of these basic services are always built in the native API module, regardless of the current configuration. A notable difference with RTAI/classic stands in the use of an increasing priority scale for tasks, which is compatible with the regular Linux scale for the SCHED_FIFO scheduling class, i.e. 1 is the lowest effective priority level for native RTAI tasks, and 99 is the highest.
- Timing services. This category groups all system calls which are related to the system timer management and queries. For instance, one needs to start the system timer before any timed service from other categories could be used (i.e. *rt_timer_start()*). Additionally, this group provides for general watchdog timers called *Alarms*.
- Synchronization support. Since tasks need to synchronize their operations to work properly, the native API provides several synchronization objects which can be used for this purpose:
  - Counting semaphores,
  - Mutexes,
  - Condition variables,
  - Event flag groups.
- Messaging and communication. This category implements several ways of exchanging bulks of data between real-time tasks, or between real-time activities in kernel space and regular Linux processes in user-space:
  - Inter-task synchronous message passing support,
  - Message queues, which can span the kernel/user-space boundary,
  - Memory heaps, which can also span the kernel/user-space boundary,
  - Message pipes, which are an improved replacement of the legacy RT-FIFOS.
- Device I/O handling. Since handling external I/O is one of the least well defined areas in RTOS design, the native API only focuses on providing simple mechanisms for dealing with interrupts, and accessing device I/O memory from user-space context.
- Registry support. This category of services is one of the corner stones of the

native API, since, as described earlier, it brings a fundamental feature with respect to providing a seamless access to system calls from different execution spaces.

Categories are made of one or more subsets of closely related services, and most of these subsets are optional; in other words, they can be selected for inclusion or not in the native API module, depending on your configuration (e.g. your application may need semaphores but not condition variables).

Moreover, it is possible to disable the entire support for the user-space execution context, in the seldom cases where all real-time activities are embodied into kernel modules.

The main idea here is to provide enough configuration flexibility to embedded setups.

Linking against the native API

RTAI/fusion provides the *rtai-config* script, which sends back various information upon request, like the valid compilation and link flags to pass to the GCC toolchain when building applications for various execution contexts (i.e. kernel, user-space, simulation), and this script should be queried the same way for getting the proper arguments in order to compile and link an application against the native API.

For instance, a trivial *Makefile* for compiling a simple user-space program named *myapp.c* which uses the native API would look like this:

```
prefix := $(shell rtai-config --prefix)

ifeq ($(prefix),)
$(error Please add <rtai-install-path>/bin to your PATH variable)
endif

CFLAGS := $(shell rtai-config --fusion-cflags)
LDFLAGS :=  -lrtai $(shell rtai-config --fusion-ldflags)

myapp: myapp.c
      $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS)
```

Mixable execution modes

RTAI/fusion allows real-time tasks embodied into user-space programs to either execute in the RTAI domain, or into the Linux domain without incurring the potential perturbations induced by the asynchronous Linux kernel activities, mainly by mean of the appropriate management of an interrupt shield. In the former case, the real-time tasks are said to run in *primary execution mode*, whilst in the latter case, those tasks are said to run in *secondary execution mode*. By convention, real-time tasks embodied into kernel modules always run in primary mode, since there is no way for them to call regular Linux kernel services directly.

When running into the RTAI domain (i.e. *primary mode*), a real-time task in user-space still has the benefit of memory protection, but is scheduled by RTAI/fusion directly, and no longer by the Linux kernel. The worst-case scheduling latency of such kind of task is

always near to the hardware limits and predictable, since RTAI/fusion is not bound to synchronizing with the Linux kernel activity in such context, and can preempt any regular Linux activity with no delay. In this execution mode, the following considerations apply:

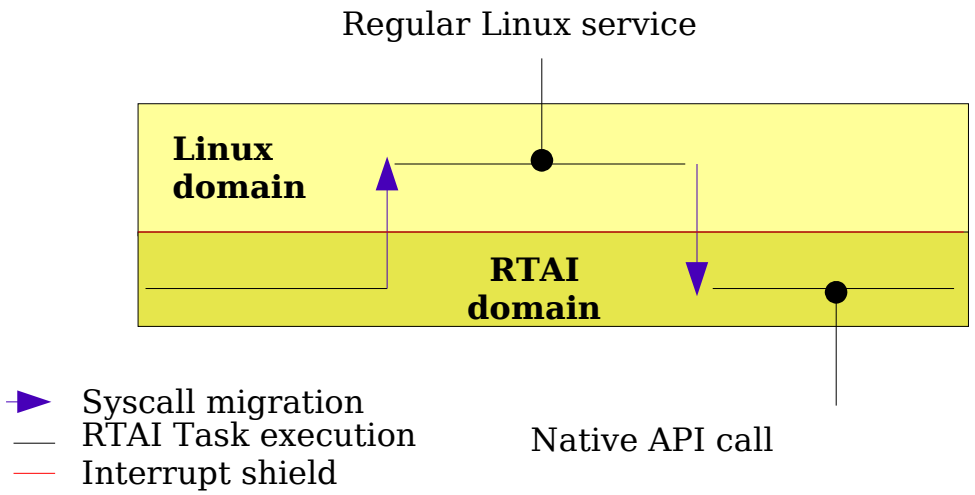
- When a real-time task in primary mode issues a regular Linux system call, it is immediately and transparently migrated by RTAI/fusion to the Linux domain, since the operation could not be fulfilled into the RTAI domain directly. At that point, the real-time task re-enters the Linux kernel at the closest rescheduling point, and resumes the standard Linux system call procedure. Time-wise, the cost of such migration obviously depends on the granularity of the Linux kernel. Specifically, the longest path between two executions of the rescheduling procedure inside the Linux kernel defines the worst case. Thanks to the continuous trend of improvements of Linux 2.6 regarding preemptability, the worst-case latency figures are constantly decreasing. This also explains why RTAI/fusion is best used over preemptible Linux kernels when there is a need for predictability even for tasks executing in secondary mode.

When running into the Linux domain (i.e. *secondary mode*), a real-time task has access to all regular Linux system calls, however, the following considerations apply:

- The behaviour of a regular Linux system call is not altered by the fact of being invoked from a real-time context. This also means that Linux system calls designed for fairness will likely have predictability issues in this context too.
- Real-time tasks calling regular Linux services are protected from unwanted preemption by Linux interrupt handlers, until they are blocked by a kernel service waiting for some I/O event to occur (e.g. regular Linux device drivers). In such a case, any interrupt-driven wake up could be delayed until no other real-time tasks is running in the Linux space, since the interrupt shield would prevent the interrupt propagation. In order to avoid such potential priority inversions between real-time tasks according to this scenario, it is still possible to provide real-time I/O device drivers which would directly run into the RTAI domain. In the latter case, the interrupt shield would not apply to the interrupts directed at this domain which has higher priority in the Adeos pipeline, thus allowing for the immediate wake up of the pending real-time task, at the expense of not reusing the regular Linux infrastructure for writing such I/O device drivers. Another option would consist of suppressing the interrupt shielding for real-time tasks in user-space (see *rt_task_set_mode()* using the *T_SHIELD* mode bit), but predictability of execution times could then be affected by regular Linux interrupt handlers.
- When a real-time task executes into the Linux domain on a given processor, the Linux kernel as a whole inherits the real-time priority of such task, and thus competes for the CPU resource by priority with other real-time tasks regardless of the domain (i.e. Linux or RTAI) they happen to belong to. In other words, the real-time priority scheme enforced by the native API is consistent across domains.
- When a real-time task executing in secondary mode calls a native RTAI service, it

may be immediately and transparently switched back to primary mode, if the invoked system call requires it. By convention, all RTAI services which might block the caller cause such transition when necessary; in other words, real-time tasks blocked on RTAI resources are always aslept in primary mode.

The following figure illustrates the migration of a real-time task across domains depending on the requested type of service:



It should be noted that domain migrations are made lazily by RTAI/fusion only when a pending system call requests it. In other words, a real-time task remains in primary mode until it calls a regular Linux service, and conversely, the secondary mode is only left whenever a native API call requests it.

To sum up, RTAI/classic's (i.e. 24.1.x and 3.x) *hard real-time mode* and RTAI/fusion's *primary mode* both deliver true real-time performances in the lowest micro-second latency range. But when it comes to calling Linux kernel services, the former consider that RTAI tasks doing so are doomed to lose their deterministic behaviour and just don't bother trying to keep them real-time; on the other hand, RTAI/fusion bets on future evolutions of Linux, like PREEMPT_RT, to improve the kernel's overall granularity, so that the secondary mode will still be real-time in the deterministic sense, with bounded albeit higher worst-case latencies.

Working with the native API

Configuring the native API

There are various ways to run the configuration system with RTAI/fusion, and there is more to setting up a complete RTAI/fusion system than configuring the native API, but we will only have a look at the steps to do the latter here, using the GTK-based graphical interface. Please see the *README.INSTALL* file from the RTAI/fusion distribution to learn how to configure RTAI/fusion using alternate ways. You can also refer to the

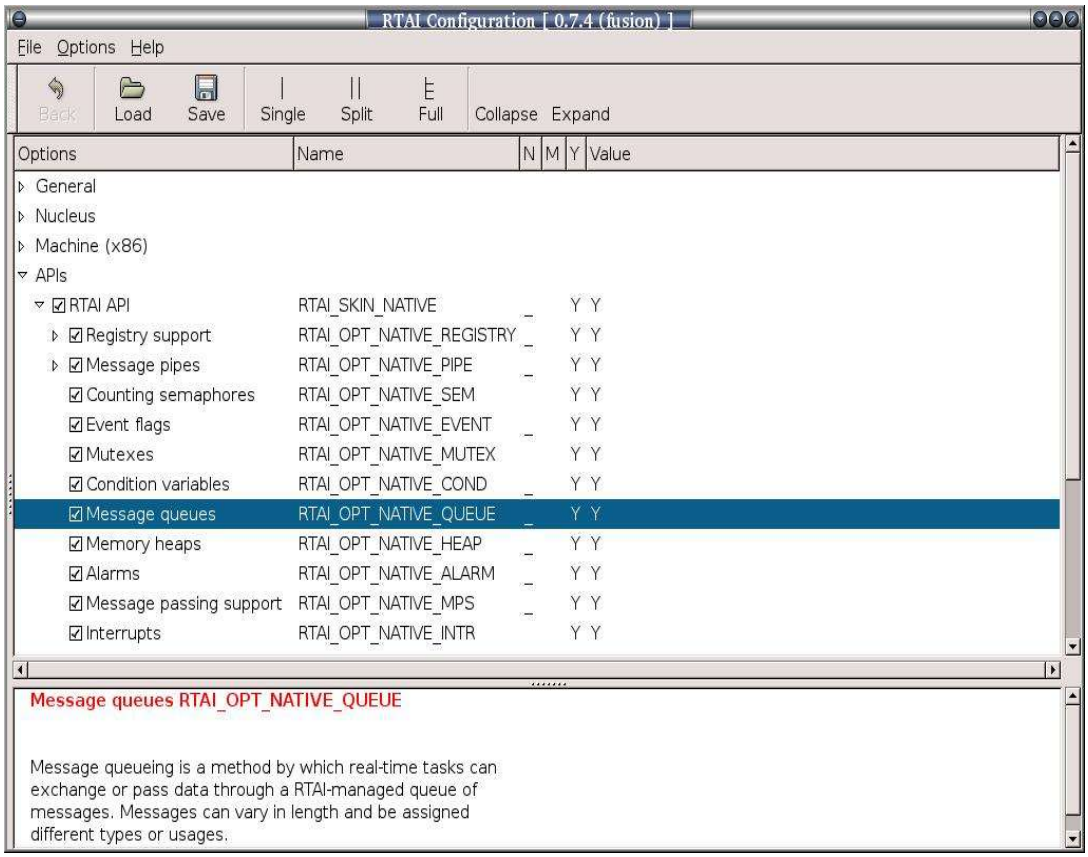
README.QUICKINSTALL file for setting up a configurable system quickly. These documents are also available on-line at:

http://download.gna.org/rtai/documentation/fusion/txt/README.INSTALL
http://download.gna.org/rtai/documentation/fusion/txt/README.QUICKINSTALL

Once you have entered the GTK-based configuration tool issuing the following command in the build tree, a typical configuration window for the native API should look like the one below:

$ mkdir $fusion_build_dir && cd $fusion_build_dir

$ make -f $fusion_src_dir/makefile gconfig



Managing tasks

The native API provides a set of multitasking mechanisms as its first category of services. The basic process object performing actions is a task, a logically complete path of application code. Each native RTAI task is an independent portion of the overall application code embodied into a C procedure, which executes on its own stack context.

The native RTAI scheduler ensures that concurrent tasks are run according to one of the supported scheduling policies. Currently, the native RTAI scheduler supports fixed

priority-based FIFO and round-robin policies.

Tasks are first created and left in a suspended state using *rt_task_create()* then started by a call to *rt_task_start()*. A shorthand combining both operations is called *rt_task_spawn ()*. There is no fundamental difference between creating a task for executing in kernel space or user-space context; actually, the only variation concerns the FPU operations support which is always enabled for user-space tasks regardless of the creation mode bits passed to *rt_task_create()*, whilst kernel based ones can choose to activate it or not.

Aside of common operations like suspend/resume or sleep requests, a task can be made periodic by programming its first release point and its period in the processor time line (see *rt_task_set_periodic()*). Combined with the high precision of the oneshot timing mode (see below), this feature makes it very simple to sustain fast sampling loops accurately.

Timing issues

There is three key points to know and keep in mind about the timing support exhibited by the native API:

- The real-time kernel needs a time source for the timed services provided by the native API to work properly. For this reason, *rt_timer_start()* must be called by the application to initialize the system timer, which will provide the needed support afterwards. In other words, unless your application actually doesn't use any of the API service involving timeouts, delays or dates, it must call *rt_timer_start()*, likely as part of its initialization chores, in order to be able to request those services subsequently.
- On hardware architectures that provide a oneshot-programmable time source, the system timer can operate either in **oneshot** or **periodic** mode. In oneshot mode, the underlying hardware will be reprogrammed after each clock tick so that the next one occurs after a (possibly non-constant) specified interval with a higher accuracy, at the expense of a larger overhead due to hardware programming duties. Periodic mode provides timing services at a lower programming cost when the underlying hardware is a true Programmable Interrupt Timer (and not a simple decrementer), but at the expense of a lower precision since all delays are rounded up to the constant interval value used to program the timer initially. The selection of the proper operation mode is made depending on the argument passed to the *rt_timer_start()* service, either the special *TM_ONESHOT* value or a valid period expressed in nanoseconds.
- Depending on the current timer operation mode as defined by the latest call to *rt_timer_start()*, all timed services which are passed timeouts, delays or dates will interpret such values either as a count of nanoseconds in oneshot mode, or as a count of *jiffies* (i.e. peiodioc ticks) in periodic mode.

There is absolutely no exception to the description above, whichever native API call is being considered. The following snippet illustrates the initialization of a 10Khz sampling

loop in a user-space application, over a RTAI task:

```
#include <rtai/task.h>
#include <rtai/timer.h>

#define TASK_PRIO  99 /* Highest RT priority */
#define TASK_MODE  0  /* No flags */
#define TASK_STKSZ 0  /* Stack size (use default one) */
#define TASK_PERIOD 100000 /* 100 usc period */

RT_TASK task_desc;

void sampling_task (void *cookie)
{
      int err;

      /* The task will undergo a 100 usc periodic timeline. */
      err = rt_task_set_periodic(NULL,TM_NOW,TASK_PERIOD);
      ...
      for (;;) {
            err = rt_task_wait_period();
            if (err)
                  break;
            /* Work for the current period */
      }
}

int main (int argc, char *argv[])
{
      int err;

      /* Disable paging for this program's memory. */
      mlockall(MCL_CURRENT|MCL_FUTURE);

      /* Start the oneshot timer. */
      err = rt_timer_start(TM_ONESHOT);

      if (err)
            fail();

      /* Create a real-time task */
      err = rt_task_create(&task_desc,
                  "MyTaskName",
                  TASK_STKSZ,
                  TASK_PRIO,
                  TASK_MODE);
      if (!err)
            /* If successfully created, start the task. */
            rt_task_start(&task_desc,&sampling_task,NULL);
```

```
        . . .
}
```

Synchronizing task operations

The native API provides four distinct classes of common synchronization objects, namely mutexes, condition variables, semaphores, and event flag groups:

- Semaphores can either count and dispatch resources, or work in pulse mode, acting as a synchronization barrier.
- An event flag group is a synchronization object represented by a long-word structure. Every available bit in such word can be used to map a user-defined event flag, tasks can either wait for to be raised in a conjunctive manner (all awaited events must have been raised for the pending task to wake up), or in a disjunctive way (at least one of the awaited events must have been raised for the pending task to wake up).
- Mutexes should be used to serialize access to some critical portion of code. They support the priority inheritance protocol, so that priority inversions among competing tasks are solved dynamically.
- Condition variables are semantically compatible with their POSIX counterparts. When properly associated with mutexes, condition variables allow tasks to wait for user-defined events to be signaled atomically. Condition variables can be used to build ad hoc synchronization tools when the pre-canned ones already provided by the native API do not fit the job.

At any time, a task pending on any of such objects can be forcibly released from its wait by a call to *rt_task_unblock()*. In such a case, a specific error code (i.e. -EINTR) will always be returned to the caller. It should be noted that a user-space task pending on RTAI-managed resources or events remains interruptible by regular Linux signals, in which case the wait is aborted, the task is switched to secondary execution mode then -EINTR is eventually returned to the caller.

Sharing memory areas

Shared memory is the fastest inter-task communication mechanism, therefore it is supported by the native API as a built-in feature to facilitate the exchange of information between tasks possibly belonging to different execution spaces.

The basic object for memory management within the native API is the *memory heap*, as created by the *rt_heap_create()* system call. Its primary purpose is to provide a region of memory usable for dynamic allocation in a time-bounded fashion (see *rt_heap_alloc()* and *rt_heap_free()*). Additionally, the native API allows the heap memory to be visible from multiple address spaces upon request by passing the *H_SHARED* mode bit at creation time to *rt_heap_create()*. In such a case, tasks which must have access to the same memory area from distinct address spaces than the creator's just need to bind to the initial heap through a call to *rt_heap_bind()*. The snippet below describes how to share a

16Kb memory heap between a kernel module and a user-space program:

From a kernel module initialization routine:

```
#include <rtai/heap.h>

RT_HEAP heap_desc;
void *shmem;

int init_module (void)
{
      int err;

      /* Create the heap in kernel space */
      err = rt_heap_create(&heap_desc,"MyHeap",16384,H_SHARED);
      ...
      /* Get the shared memory address */
      err = rt_heap_alloc(&heap_desc,0,TM_NONBLOCK,&shmem);
      ...
}
```

From a user-space context:

```
#include <rtai/heap.h>

RT_HEAP heap_desc;
void *shmem;

int main (int argc, char *argv[])
{
      int err;

      /* Disable paging for this program's memory. */
      mlockall(MCL_CURRENT|MCL_FUTURE);

      /* Bind to the heap created in kernel space */
      err = rt_heap_bind(&heap_desc,"MyHeap");
      ...
      /* Get the shared memory address */
      err = rt_heap_alloc(&heap_desc,0,TM_NONBLOCK,&shmem);
      ...
}
```

In the example above, none of the calling contexts could block on the binding call, waiting for the searched object to be eventually created, since those contexts are not underlaid by real-time tasks (*init_module()* and *main()* routines can never be part of any real-time contexts), so this example basically illustrates how to share a memory area first defined by a kernel module as part of its initialization chores, with an emerging user-space program which has been started afterwards. However, the user-space portion could

be moved over a real-time task context if necessary, so that the binding operation could block the caller until the shared memory heap is created.

Exchanging messages

Aside of shared memory heaps, there are several other ways to exchange bulks of data between tasks, within a given execution space or between kernel and user-space contexts. The native API provides the following message-oriented features to this end:

- Message queues can be created by the *rt_queue_create()* service, and used by tasks to receive variable-sized messages which have been posted by other tasks or interrupt service routines. The current implementation exhibits a zero-copy mechanism to deliver the queued messages, and also allows for broadcasting a single message to several readers without data duplication.
- A synchronous message passing extension is also available as part of the task interface. It allows a task to receive, through the *rt_task_receive()* service, a variable-sized message sent from another task using the *rt_task_send()* service. In order to provide for fully synchronous operations, the sending task is blocked until the receiver invokes *rt_task_reply()* to finish the transaction.
- Message pipes provide a mean for exchanging variable-sized messages between RTAI tasks and standard Linux processes using the regular file i/o interface. Unlike the two previous features, message pipes do not deliver real-time communications, but rather implement a practical and efficient data path allowing standard Linux processes to exchange data with real-time tasks. Practically, the real-time task connects to one side of the pipe using the *rt_pipe_create()* service, then exchange messages with the other side using the *rt_pipe_write()* and *rt_pipe_read()* system calls. The other side of the pipe is typically opened by a standard Linux process through the regular *open(2)* call which is passed the name of the associated special device file for the communication (i.e. /dev/rtp0 to /dev/rtp31), then the pipe is read and written by the process using the usual *read (2)* and *write(2)* services. Although the primary use of the message pipes is message-oriented communication, byte-oriented or streaming mode is also possible when sending data from RTAI tasks to standard Linux processes using *rt_pipe_stream()*. The snippet below describes how to create a pipe between a kernel module usable by real-time tasks and a standard Linux process:

From a kernel module initialization routine:

```
#include <rtai/pipe.h>

RT_PIPE pipe_desc;

int init_module (void)
{
      int err;
      /* Connect the kernel-side of the message pipe to the
         special device file /dev/rtp7. */
```

```
        err = rt_pipe_create(&pipe_desc,"MyPipe",7);
        ...
}
```

From a regular Linux process:

```
#include <rtai/pipe.h>

int pipe_fd;

int main (int argc, char *argv[])
{
        /* Open the Linux side of the pipe. */
        pipe_fd = open("/dev/rtp7",O_RDWR);
        ...
        /* Write a message to the pipe. */
        write(pipe_fd,"hello world",11);
        ...
}
```

Since the pipe object has been registered when created from kernel space, we could alternatively open the associated special device file using the pipe's symbolic name as exported by the registry, which would end up opening */dev/rtp7*, i.e.:

```
        pipe_fd = open("/proc/rtai/registry/pipes/MyPipe",O_RDWR);
```

Asynchronous notifications

RTAI tasks running in kernel space can be notified of various events asynchronously upon request. When applicable, a synchronous method for collecting the same events is also available to user-space tasks. We can distinguish four types of asynchronously notifiable events in the context of the native API:

- Alarms are general purpose watchdogs. Any RTAI task may create any number of alarms by calling the *rt_alarm_create()* service, and use them to run a user-defined handler, after a specified delay has elapsed. Since kernel-based asynchronous handlers cannot run user-space code, RTAI tasks in user-space can still explicitely wait for the next alarm shot instead, by calling the *rt_alarm_wait ()* service. In other words, asynchronous handler execution is replaced by a synchronous alarm server in user-space.
- Interrupts (actual device-generated events or virtual Adeos interrupts) can be associated a user-defined handler in kernel space using the *rt_intr_create()* service. Since kernel-based asynchronous handlers cannot run user-space code, RTAI tasks in user-space can still explicitely wait for the next interrupt occurrence instead, by calling the *rt_intr_wait()* service. In other words, asynchronous handler execution is replaced by a synchronous interrupt server in user-space.

- RTAI tasks have an ad hoc mechanism for sending (*rt_task_notify()*) and receiving (*rt_task_catch()*) signal notifications, in parallel to the regular one implemented by the Linux kernel for its processes. Pending signals cause the installed signal handler in kernel space to be fired each time the recipient task resumes execution.
- Under some circumstances, it may be useful to be notified whenever a task switch occurs, or whenever RTAI tasks are created or deleted. The native API exports hooks from its scheduling core in order to provide such information, through the *rt_task_add_hook()* interface.

Mixing execution spaces

As you may have already noticed, the native API provides a context-independent support to real-time tasks, regardless of their execution space, either in kernel or user-space context. Sometimes, splitting the application with - usually small - portions in kernel space and the remaining part in user-space is useful. For instance, a real-time I/O driver might be more easily implemented embodied into a kernel module, whilst the data it collects would be better processed by a user-space program, still with real-time guarantees.

In such a case, it is important to be able to share the real-time objects between kernel and user-space contexts seamlessly. To this end, all objects created by the native API can be indexed on a free-form symbolic key maintained by a system-wide *registry*. Conversely, specific objects can be searched for and subsequently made available to the caller when found, through a particular operation called *binding*. There is a *binding* system call for each class of registrable objects exported by the native API; for instance *rt_task_bind()* can be invoked to bind tasks to the caller's context, *rt_queue_bind()* for queues, *rt_sem_bind()* for semaphores and so on.

Using the registry for the purpose of sharing real-time objects is simple: creators of those objects must pass a valid and unique *name* to the object creation routine, which will in turn be used by the registry to index the new object in a global table. Meanwhile, binding requests bearing the same *name* may be issued from other user-space contexts, in order to get back the uniform object descriptor, which could in turn be used seamlessly in all relevant native API services. Binding requests can block waiting for the object to be created if the caller is a real-time task, or return immediately with the proper error code if the caller cannot wait, or complete successfully if the object is known from the registry on entry.

For instance, the following snippet illustrates how to send RTAI signals to a kernel-based task from a user-space program, and for that, we first need to get back the uniform descriptor associated to this task in the context of the program:

From a kernel module, create the task:

```
#include <rtai/task.h>
```

```
#define TASK_PRIO  99              /* Highest RT priority */
#define TASK_MODE  T_FPU|T_CPU(0)  /* Uses FPU, bound to CPU #0 */
#define TASK_STKSZ 4096            /* Stack size (in bytes) */

RT_TASK task_desc;

void task_body (void *cookie)
{
    for (;;) {
      /* Task processing loop in kernel space. */
    }
}

int init_module (void)

{
    int err;

    err = rt_task_create(&task_desc,
                  "MyTaskName",
                  TASK_STKSZ,
                  TASK_PRIO,
                  TASK_MODE);
    if (!err)
      rt_task_start(&task_desc,&task_body,NULL);

      ...
}
```

From a user-space program, bind to the task in kernel space:

```
#include <rtai/task.h>

#define SIGNALS (0x1|0x4)

RT_TASK task_desc;

int main (int argc, char *argv[])
{
      int err;

      /* Disable paging for this program's memory. */
      mlockall(MCL_CURRENT|MCL_FUTURE);

      /* Bind to the real-time task in kernel space. */
      err = rt_task_bind(&task_desc,"MyTaskName");

      if (!err)
            /* Send this task a couple of signals. */
            rt_task_notify(&task_desc,SIGNALS);
```

```
    ...
}
```

Writing user-space device drivers

Seamless integration of stringent real-time capabilities in user-space context allows to implement i/o device drivers as common Linux programs. To this end, the native API exports the following features to real-time applications embodied into regular Linux programs:

- Interrupt binding and synchronization services. The related system calls provide a mean to user-space programs to control interrupt channels created in kernel space. For this purpose, interrupt channels are named objects known from the global registry which simply need to be bound to the caller's context (*rt_intr_bind()*), then usual operations can then be applied on the associated channel using the returned uniform interrupt descriptor, such as enabling (*rt_intr_enable()*) or disabling (*rt_intr_disable()*) it. It is also possible to create the interrupt object directly from a user-space context, in case sharing it with some kernel space code is not required. Additionally, a user-space task can wait for interrupt occurrences using the *rt_intr_wait()* service.
- I/O region mapping. Accessing a region of I/O memory may be a pre-requisite to communicate with some devices. The native API provides a system call to request access to such regions (*rt_misc_get_io_region()*) and conversely another one to release them (*rt_misc_put_io_region()*). These system calls are simple wrappers to the regular kernel services which handle such requests, namely *request_region ()* and *release_region().*

The following snippet illustrates how to create an interrupt server task in user-space to process interrupt triggered by some I/O device:

```
#include <rtai/task.h>
#include <rtai/intr.h>

#define IRQ_NUMBER 7  /* Intercept interrupt #7 */
#define TASK_PRIO  99 /* Highest RT priority */
#define TASK_MODE  0  /* No flags */
#define TASK_STKSZ 0  /* Stack size (use default one) */

RT_INTR intr_desc;

RT_TASK server_desc;

void irq_server (void *cookie)

{
    for (;;) {
```

```
       /* Wait for the next interrupt on channel #7. */
       err = rt_intr_wait(&intr_desc,TM_INFINITE);

       if (!err) {
           /* Process the interrupt. */
       }
    }
}

int main (int argc, char *argv[])

{
    int err;

    /* Disable paging for this program's memory. */
    mlockall(MCL_CURRENT|MCL_FUTURE);

    /* ... */

    /* Create the interrupt object, asking for automatic
       re-enabling of the channel after each IRQ. */
    err = rt_intr_create(&intr_desc,IRQ_NUMBER,I_AUTOENA);

    /* ... */

    err = rt_task_create(&server_desc,
                  "MyIrqServer",
                  TASK_STKSZ,
                  TASK_PRIO,
                  TASK_MODE);
    if (!err)
      rt_task_start(&server_desc,&irq_server,NULL);

    /* ... */
}
```

Debugging applications

From user-space context, applications using the native API can be debugged using GDB. RTAI/fusion has been designed to work as symbiotically as possible with the Linux kernel, and as such, standard *ptracing* and Linux signal supports are kept available for use with real-time tasks in user-space created by the native API. Internally, the tasks will be switched automatically to secondary mode, i.e. under the control of the Linux kernel, whenever their respective code is stepped by GDB, a fault is raised, or a breakpoint is otherwise hit. RTAI/fusion guarantees that they will be switched back automatically to primary mode as soon as it is needed. From a user's point of view, debugging a real-time application with GDB is no different from debugging any other non-RTAI enabled

program.

From kernel context, modules embedding real-time applications using the native API should be initially written and debugged using the simulator, for their hardware independent portions, that is. Then, tedious debugging using the common kernel-based techniques could be applied to finish up the work.

Tips and tricks

Porting from RTAI/classic

Providing complete porting guidelines from RTAI/classic to RTAI/fusion's native API is outside the scope of this document; however, the following table presents the major correspondences between both programming interfaces:

RTAI/classic package	*RTAI/fusion package*
Tasks	Tasks
Tasklets	Alarms
Semaphores	Semaphores
Condition variables	Condition variables
Mutexes	Mutexes
Bits	Event flag groups
Fifos	Message pipes
Message queues (POSIXish)	Message queues
Messages	Message passing support
Shared memory	(Shared) Heaps
Local memory (malloc)	(Local) Heaps
User-space interrupts	Interrupt API
Extended mailboxes	Message queues

Detecting unwanted switches to the secondary execution mode

Sometimes, switching to the secondary execution mode for a real-time task so that it can re-enter the Linux kernel context may be an unwanted behaviour, likely due to invoking a Linux system call involuntarily. In order to trap this situation, the native API allows real-time tasks in user-space to require the SIGXCPU signal (i.e. *CPU time limit exceeded*) to be sent to them each time they switch from the primary to the secondary execution mode.

To use this feature, the application needs to install a handler for the SIGXCPU signal (e.g. using *sigaction(2)*), then set the *T_WARNSW* mode bit for the task to monitor, using the *rt_task_set_mode()* system call. If no SIGXCPU handler has been set, the whole

application is terminated upon receipt of this signal.

A simple way to identify the code issuing the Linux system call within the monitored application consists of running the latter over GDB, then issuing the “backtrace” command whenever GDB reports the SIGXCPU signal receipt. Another way, which does not require to run the application over GDB, consists of dumping the current stack frame using the standard *backtrace(3)* routine (see */usr/include/execinfo.h*) from the SIGXCPU signal handler. The statement causing the transition should appear as the outmost call frame in both cases.

Additionally, one can check the number of mode switches undergone by any active task by looking at the *MODSW* field available from a */proc/rtai/stats* dump: the number of switches from secondary to primary mode is given first, followed by the number of opposite switches (e.g. those might have triggered the SIGXCPU signal whenever *T_WARNSW* has been set for the task being considered).

Detecting runaway tasks

RTAI/fusion threads underlying the tasks created from the native API can be monitored for runaway detection by the real-time nucleus. In order to enable this feature for debugging your application, you will need to enter the configuration tool, then select the expert configuration mode (*CONFIG_RTAI_OPT_EXPERT*) from the *General* menu. This should make a set of additional configuration options appear, among which you should find the *Watchdog support* entry available from the *Nucleus* menu.

When enabled, the built-in watchdog timer prevents hard lockups due to runaway real-time threads consuming all the CPU power, by forcibly suspending the current real-time thread whenever the Linux kernel was totally starved from CPU for more than four seconds. A kernel message is also logged to report such action.

Conclusion

Integrating stringent real-time features into a GPOS system like GNU/Linux is fundamentally different from providing a standalone RTOS: those features must coexist on the same hardware with the general purpose Linux kernel and applications. This fact raises a number of structural issues which go well beyond solely implementing the mere RTOS machinery.

In this respect, RTAI/fusion's approach resides in a tight integration with the GNU/Linux environment, which keeps RTAI's internals aware of and compatible with important Linux semantics, as far as predictability is not weakened by such design decision. For instance, native RTAI tasks are mapped over Linux task contexts, escaping Linux scheduling control when running in the most stringent real-time mode, but are still able to process regular Linux events, like signals, sent to them.

For this integration to be efficient, the reforged RTAI API also attempts to address the compactness, orthogonality and usability issues, so that it eventually appears as a native support provided by the Linux kernel to the real-time application designers.

Useful Links

- The RTAI/fusion documentation tree:
  http://download.gna.org/rtai/documentation/fusion/
- The reference manual for the native API:
  http://download.gna.org/rtai/documentation/fusion/html/api/
- A detailed explanation of RTAI/fusion's fundamentals over Adeos:

  http://download.gna.org/rtai/documentation/fusion/pdf/Life-with-Adeos.pdf

rpm@xenomai.org