DIAPM-RTAI

Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano
Real Time Application Interface (for Linux)

A Hard Real Time support for LINUX

This document explains how to call the functions available in DIAPM-RTAI

The RTAI distribution (www.aero.polimi.it/projects/rtai/) contains a wealth of examples showing how to use services and APIs described herein.

Document written by: E. Bianchi, L. Dozio, P. Mantegazza.
Dipartimento di Ingegneria Aerospaziale
Politecnico di Milano
e-mail: bianchi@aero.polimi.it
e-mail: dozio@aero.polimi.it
e-mail: mantegazza@aero.polimi.it

Appendices contributed also by Pierre Cloutier and Steve Papacharalabous:
e-mail: pcloutier@poseidoncontrols.com
e-mail: stevep@zentropix.com

Send comments and fixes to the manual coordinator Giuseppe Renoldi:
e-mail: grenoldi@usa.net

Help by Gábor Kiss, Computer and Automation Institute of Hungarian Academy of Sciences, in updating and revising this doc is acknowledged.

Functions provided by rtai_sched modules for:

- **UniProcessor (UP),**
- **SymmetricMultiProcessors (SMP),**
- **MultiUniPprocessors (MUP).**

See Appendix A for a quick overview of **RTAI** schedulers.

Task functions:

- rt_task_init
- rt_task_init_cpuid
- rt_task_delete
- rt_task_make_periodic
- rt_task_make_periodic_relative_ns
- rt_task_wait_period
- rt_task_yield
- rt_task_suspend
- rt_task_resume
- rt_get_task_state
- rt_whoami
- rt_task_signal_handler
- rt_set_runneable_on_cpus
- rt_set_runnable_on_cpuid
- rt_task_use_fpu
- rt_linux_use_fpu
- rt_preempt_always
- rt_preempt_always_cpuid

NAME

rt_task_init
rt_task_init_cpuid

FUNCTION

Create a new real time task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_task_init (RT_TASK *task, void (*rt_thread)(int), int data,
      int stack_size, int priority, int uses_fpu,
      void(*signal)(void));

int rt_task_init_cpuid (RT_TASK *task, void (*rt_thread)(int), int
      data, int stack_size, int priority, int uses_fpu,
      void(*signal)(void), unsigned int cpuid);
```

DESCRIPTION

rt_task_init and **rt_task_init_cpuid** create a real time task..
task is a pointer to an RT_TASK type structure whose space must be provided by the application. It must be kept during the whole lifetime of the real time task.
rt_thread is the entry point of the task function. The parent task can pass a single integer value ***data*** to the new task being created. Recall that an appropriately type casting allows ***data*** to be a pointer to whatever data structure one would like to pass to the task, so you can indirectly pass whatever you want to the task.
stack_size is the size of the stack to be used by the new task, in sizing it recall to make room for any real time interrupt handler, as real time interrupts run on the stack of the task they interrupt. So try to avoid being too sparing.
priority is the priority to be given to the task. The highest priority is 0, while the lowest is RT_LOWEST_PRIORITY.
uses_fpu is a flag. A nonzero value indicates that the task will use the floating point unit.
signal is a function that is called, within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch. Note however that ***signal*** is not called at the very first scheduling of the task. Such a function can be assigned and/or changed dynamically whenever needed , see function **rt_task_signal_handler**.
The newly created real time task is initially in a suspend state. It can be made active by calling: **rt_task_make_periodic**, **rt_task_make_periodic_relative_ns,** **rt_task_resume**.
When used with the MUP scheduler **rt_task_init** automatically selects which CPU the task will run on, while with the SMP scheduler the task defaults to using any of the available CPUs. This assignment may be changed by calling **rt_set_runnable_on_cpus** or **rt_set_runnable_on_cpuid**. If ***cpuid*** is invalid **rt_task_init_cpuid** falls back to automatic CPU selection.
Whatever scheduler is used on multiprocessor systems **rt_task_init_cpuid** allows to create a task and assign it to a single specific CPU ***cpuid*** from its very beginning, without any need to call **rt_set_runnable_on_cpuid** later on.

RETURN VALUE

On success 0 is returned. On failure a negative value is returned as described below.

ERRORS

EINVAL	Task structure pointed by ***task*** is already in use.
ENOMEM	***stack_size*** bytes could not be allocated for the stack.

NAME

rt_task_delete

FUNCTION

Delete a real time task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_task_delete (RT_TASK *task);
```

DESCRIPTION

rt_task_delete deletes a real time task previously created by **rt_task_init** or **rt_task_init_cpuid**.
task is the pointer to the task structure.
If task ***task*** was waiting on a queue, i.e. semaphore, mailbox, etc, it is removed from such a queue and messaging tasks pending on its message queue are unblocked with an error return.

RETURN VALUE

On success 0 is returned. On failure a negative value is returned as described below.

ERRORS

EINVAL ***task*** does not refer to a valid task.

NAME

rt_task_make_periodic
rt_task_make_periodic_relative_ns

FUNCTION

Make a task run periodically

SYNOPSIS

```
#include "rtai_sched.h"

int rt_task_make_periodic (RT_TASK *task, RTIME start_time, RTIME
      period);

int rt_task_make_periodic_relative_ns (RT_TASK *task, RTIME
      start_delay, RTIME period);
```

DESCRIPTION

rt_task_make_periodic and **rt_task_make_periodic_relative_ns** mark the task ***task***, previously created with **rt_task_init**, as suitable for a periodic execution, with period ***period***, when **rt_task_wait_period** is called.
The time of first execution is given by ***start_time*** or ***start_delay***.
start_time is an absolute value measured in clock ticks.
start_delay is relative to the current time and measured in nanoseconds.

RETURN VALUE

On success `0` is returned. On failure a negative value is returned as described below.

ERRORS

EINVAL ***task*** does not refer to a valid task.

NAME

rt_task_wait_period

FUNCTION

Wait till next period

SYNOPSIS

```
#include "rtai_sched.h"

void rt_task_wait_period (void);
```

DESCRIPTION

rt_task_wait_period suspends the execution of the currently running real time task until the next period is reached. The task must have been previously marked for a periodic execution by calling **rt_task_make_periodic** or **rt_task_make_periodic_relative_ns**. Note that the task is suspended only temporarily, i.e. it simply gives up control until the next time period

NAME

rt_task_yield

FUNCTION

Yield the current task

SYNOPSIS

```
#include "rtai_sched.h"

void rt_task_yield (void);
```

DESCRIPTION

rt_task_yield stops the current task and takes it at the end of the list of ready tasks having its same priority. The scheduler makes the next ready task of the same priority active.
Recall that RTAI schedulers allow only higher priority tasks to preempt the execution of lower priority ones. So equal priority tasks cannot preempt each other and **rt_task_yield** should be used if a user needs a cooperative time slicing among equal priority tasks. The implementation of the related policy is wholly in the hand of the user. It is believed that time slicing is too much an overhead for the most demanding real time applications, so it is left up to you.

NAME

rt_task_suspend

FUNCTION

Suspend a task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_task_suspend (RT_TASK *task);
```

DESCRIPTION

rt_task_suspend suspends execution of the task ***task***.
It will not be executed until a call to **rt_task_resume** or **rt_task_make_periodic** is made. No account is made for multiple suspends, i.e. a multiply suspended task is made ready as soon as it is **rt_task_resume**d, thus immediately resuming its execution if it is the highest in priority.

RETURN VALUE

On success 0 is returned. On failure a negative value is returned as described below.

ERRORS

EINVAL ***task*** does not refer to a valid task.

NOTES

The new RTAI 24.1.xx development releases take into account multiple suspend and require as many **rt_task_resume**s as the **rt_task_suspend**s placed on a task.

NAME

rt_task_resume

FUNCTION

Resume a task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_task_resume (RT_TASK *task);
```

DESCRIPTION

rt_task_resume resumes execution of the task ***task*** previously suspended by **rt_task_suspend,** or makes a newly created task ready to run, if it makes the task ready. Since no account is made for multiple suspend **rt_task_resume** unconditionally resumes any task it makes ready.

RETURN VALUE

On success 0 is returned. On failure a negative value is returned as described below.

ERRORS

EINVAL ***task*** does not refer to a valid task.

NOTES

The new RTAI 24.1.xx development releases take into account multiple suspend and require as many **rt_task_resume**s as the **rt_task_suspend**s placed on a task.

NAME

rt_get_task_state

FUNCTION

Query task state

SYNOPSIS

```
#include "rtai_sched.h"

int rt_get_task_state (RT_TASK *task);
```

DESCRIPTION

rt_get_task_state returns the state of a real time task.
task is a pointer to the task structure.

RETURN VALUE

Task state is formed by the bitwise OR of one or more of the following flags:

READY	Task ***task*** is ready to run (i.e. unblocked). Note that on a UniProcessor machine the currently running task is just in READY state, while on MultiProcessors can be (READY | RUNNING), see below.
SUSPENDED	Task ***task*** blocked waiting for a resume.
DELAYED	Task ***task*** blocked waiting for its next running period or expiration of a timeout.
SEMAPHORE	Task ***task*** blocked on a semaphore, waiting for the semaphore to be signaled.
SEND	Task ***task*** blocked on sending a message, receiver was not in RECEIVE state.
RECEIVE	Task ***task*** blocked waiting for an incoming messages, sends or rpcs.
RPC	Task ***task*** blocked on a Remote Procedure Call, receiver was not in RECEIVE state.
RETURN	Task ***task*** blocked waiting for a return from a Remote Procedure Call, receiver got the RPC but has not replied yet.
RUNNING	Task ***task*** is running, used only for SMP schedulers.

NOTES

The returned task state is just an approximate information. Timer and other hardware interrupts may cause a change in the state of the queried task before the caller could evaluate the returned value. Caller should disable interrupts if it wants reliable info about an other task.
rt_get_task_state does not perform any check on pointer ***task***.

NAME

rt_whoami

FUNCTION

Get the task pointer of the current task

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK *rt_whoami (void);
```

DESCRIPTION

Calling **rt_whoami** a task can get a pointer to its own task structure.

RETURN VALUE

The pointer to the current task is returned.

NAME

rt_task_signal_handler

FUNCTION

Set the signal handler of a task

SYNOPSIS

```
#include "rtai_sched.h"

void rt_task_signal_handler (RT_TASK *task, void (*handler)(void));
```

DESCRIPTION

rt_task_signal_handler installs, or changes, the signal function of a real time task.
task is a pointer to the real time task.
handler is the entry point of the signal function. A signal handler function can be set also when the task is newly created with **rt_task_init**.
The signal handler is a function called, within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch, except at its very first scheduling. It allows you to implement whatever signal management policy you think useful, and many other things as well.

RETURN VALUE

On success 0 is returned. On failure a negative value is returned as described below.

ERRORS

EINVAL ***task*** does not refer to a valid task.

NAME

rt_set_runnable_on_cpus
rt_set_runnable_on_cpuid

FUNCTION

Assign CPUs to a task

SYNOPSIS

```
#include "rtai_sched.h"

void rt_set_runnable_on_cpus (RT_TASK *task, unsigned int
       cpu_mask);
void rt_set_runnable_on_cpuid (RT_TASK *task, unsigned int cpuid);
```

DESCRIPTION

rt_set_runnable_on_cpus, **rt_set_runnable_on_cpuid** select one or more CPUs which are allowed to run task ***task***.
rt_set_runnable_on_cpuid assigns a task to a single specific CPU.
rt_set_runnable_on_cpus behaves differently for MUP and SMP schedulers. Under the SMP scheduler bit<n> of ***cpu_mask*** enables the task to run on CPU<n>. Under the MUP scheduler it selects the CPU with less running tasks among those allowed by ***cpu_mask***. Recall that with MUP a task must be bounded to run on a single CPU.
If no CPU, as selected by ***cpu_mask*** or ***cpuid***, is available, both functions choose a possible CPU automatically, following the same rule as above.

NOTES

This call has no effect on UniProcessor (UP) systems.

NAME

rt_task_use_fpu
rt_linux_use_fpu

FUNCTION

Set indication of FPU usage

SYNOPSIS

```
#include "rtai_sched.h"

int rt_task_use_fpu (RT_TASK* task, int use_fpu_flag);
void rt_linux_use_fpu (int use_fpu_flag);
```

DESCRIPTION

rt_task_use_fpu informs the scheduler that floating point arithmetic operations will be used by the real time task ***task***.
rt_linux_use_fpu informs the scheduler that floating point arithmetic operations will be used also by foreground Linux processes, i.e. the Linux kernel itself (unlikely) and any of its processes.
If ***use_fpu_flag*** has a nonzero value, the Floating Point Unit (FPU) context is also switched when ***task*** or the kernel become active. This makes task switching slower, negligibly, on all 32 bits CPUs but 386s and the oldest 486s. This flag can be set also by **rt_task_init** when the real time task is created. With UP and MUP schedulers care is taken to avoid useless saves/restores of the FPU environment. Under SMP tasks can be moved from CPU to CPU so saves/restores for tasks using the FPU are always carried out.
Note that by default Linux has this flag cleared. Beside by using **rt_linux_use_fpu** you can change the Linux FPU flag when you insmod any **RTAI** scheduler module by setting the **LinuxFpu** command line parameter of the **rtai_sched** module itself.

RETURN VALUE

On success 0 is returned. On failure a negative value is returned as described below.

ERRORS

EINVAL ***task*** does not refer to a valid task.

NAME

rt_preempt_always
rt_preempt_always_cpuid

FUNCTION

Enable hard preemption

SYNOPSIS

```
#include "rtai_sched.h"

void rt_preempt_always (int yes_no);
void rt_preempt_always_cpuid (int yes_no, unsigned int cpu_id);
```

DESCRIPTION

In the oneshot mode the next timer expiration is programmed after a timer shot by choosing among the timed tasks the one with a priority higher than the task chosen to run as current, with the constraint of always assuring a correct Linux timing. In such a view there is no need to fire the timer immediately. In fact it can happen that the current task can be so fast to get suspended and rerun before the one that was devised to time the next shot when it was made running. In such a view **RTAI** schedulers try to shoot only when strictly needed. This minimizes the number of slow setups of the 8254 timer used with UP and 8254 based SMP schedulers. While such a policy minimizes the number of actual shots, greatly enhancing efficiency, it can be unsuitable when an application has to be guarded against undesired program loops or other unpredicted error causes. Calling these functions with a nonzero value assures that a timed high priority preempting task is always programmed to be fired while another task is currently running. The default is no immediate preemption in oneshot mode, i.e. firing of the next shot programmed only when strictly needed to satisfy tasks timings.
Note that with UP and SMP schedulers there is always only a timing source so that ***cpu_id*** in **rt_preempt_always_cpuid** is not used. With the MUP scheduler you have an independent timer for each CPU, so **rt_preempt_always** applies to all the CPUs while **rt_preempt_always_cpuid** should be used when preemption is to be forced only on a specific CPU.

NAME

rt_sched_lock
rt_sched_unlock

FUNCTION

Lock the scheduling of tasks.

SYNOPSIS

```
#include "rtai_sched.h"

void rt_sched_lock (void);
void rt_sched_unlock (void);
```

DESCRIPTION

rt_sched_lock, **rt_sched_unclock** lock/unlock, on the CPU on which they are called, any scheduler activity, thus preventing a higher priority task to preempt a lower priority one. They can be nested, provided unlocks are paired to locks in reversed order. It can be used for synchronization access to data among tasks. Note however that under MP the lock is active only for the CPU on which is has been issued, so it cannot be used to avoid races with tasks that can run on any other available CPU. Interrupts are not affected by such calls. Any task that needs rescheduling while a scheduler lock is in place will be only at the issuing of the last unlock

RETURN VALUE

None.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_change_prio
rt_get_prio
rt_get_inher_prio

FUNCTION

Change/check a task priority.

SYNOPSIS

```
#include "rtai_sched.h"

int rt_change_prio (RT_TASK* task, int prio);
int rt_get_prio (RT_TASK* task);
int rt_get_inher_prio (RT_TASK* task);
```

DESCRIPTION

rt_change_prio changes the base priority of task ***task*** to ***prio***.
rt_get_prio returns the base priority of task ***task***.
rt_get_inher_prio returns the priority task ***task*** has inherited from other tasks, either blocked on resources owned by or waiting to pass a message to task ***task***.
task is the affected task.
prio is the new priority, it can range within 0 < ***prio*** < RT_LOWEST_PRIORITY.
Recall that a task has a base native priority, assigned at its birth or by **rt_change_prio,** and an actual, inherited, priority. They can be different because of priority inheritance.

RETURN VALUE

All functions return the priority requested, **rt_change_prio** returns the base priority task ***task*** had before the change.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

Timer functions:

- rt_set_oneshot_mode
- rt_set_periodic_mode
- start_rt_timer
- stop_rt_timer
- start_rt_apic_timers
- stop_rt_apic_timers
- count2nano
- count2nano_cpuid
- nano2count
- nano2count_cpuid
- rt_get_time
- rt_get_time_cpuid
- rt_get_time_ns
- rt_get_cpu_time_ns
- next_period
- rt_busy_sleep
- rt_sleep
- rt_sleep_until

NAME

rt_set_oneshot_mode
rt_set_periodic_mode

FUNCTION

Set timer mode

SYNOPSIS

```
#include "rtai_sched.h"

void rt_set_oneshot_mode (void);
void rt_set_periodic_mode (void);
```

DESCRIPTION

rt_set_oneshot_mode sets the oneshot mode for the timer. It consists in a variable timing based on the CPU clock frequency. This allows tasks to be timed arbitrarily. It must be called before using any time related function, including time conversions. Note that on i386s, i486s and earlier Pentiums, and compatibles, there is no CPU Time Stamp Clock (TSC) to be used as a continuously running time base for oneshot timings. For such machines a continuously running counter 2 of the 8254 timer is used to emulate the TSC. No wrap around danger exists because of the need of keeping Linux jiffies at HZ hz (HZ is a macros found in Linux param.h and is usually set to 100). Note however that reading an 8254 counter takes a lot of time. So on such machines the oneshot mode should be used only if strictly needed and for not too high frequencies. Moreover for such a case the timer resolution is clearly that of the 8254, i.e. 1193180 Hz.
rt_set_periodic_mode sets the periodic mode for the timer. It consists of a fixed frequency timing of the tasks in multiple of the period set with a call to **start_rt_timer**. The resolution is that of the 8254 (1193180 Hz) on a UP machine, or if the 8254 based SMP scheduler is being used. For the SMP scheduler timed by the local APIC timer and for the MUP scheduler the timer resolution is that of the local APIC timer frequency, generally the bus frequency divided 16. Any timing request not being an integer multiple of the set timer period is satisfied at the closest period tick. It is the default mode when no call is made to set the oneshot mode.
Oneshot mode can be set initially also with the **OneShot** command line parameter of the **rtai_sched** module.

NOTES

Stopping the timer by **stop_rt_timer** sets the timer back into its default (periodic) mode. Always call **rt_set_oneshot_mode** before each **start_rt_timer** if you want to be sure to have it oneshot on multiple insmod without rmmoding the **RTAI** scheduler in use..

NAME

start_rt_timer
stop_rt_timer

FUNCTION

Start/stop timer

SYNOPSIS

```
#include "rtai_sched.h"

RTIME start_rt_timer (int period);
void stop_rt_timer (void);
```

DESCRIPTION

start_rt_timer starts the timer with a period ***period***. The period is in internal count units and is required only for the periodic mode. In the oneshot mode the period value is ignored.
This functions uses the 8254 with the UP and the 8254 based SMP scheduler. Otherwise uses a single local APIC with the APIC based SMP schedulers and an APIC for each CPU with the MUP scheduler. In the latter case all local APIC timers are paced in the same way, according to the timer mode set.
stop_rt_timer stops the timer. The timer mode is set to periodic.

RETURN VALUE

The period in internal count units.

NAME

start_rt_apic_timers
stop_rt_apic_timers

FUNCTION

Start/stop local APIC timers

SYNOPSIS

```
#include "rtai_sched.h"

void start_rt_apic_timers (struct apic_timer_setup_data
       *setup_data, unsigned int rcvr_jiffies_cpuid);
void stop_rt_apic_timers (void);
```

DESCRIPTION

start_rt_apic_timers starts local APIC timers according to what is found in ***setup_data***. ***setup_data*** is a pointer to an array of structures `apic_timer_setup_data`, see function **rt_setup_apic_timers** in **RTAI** module functions described further on in this manual. ***rcvr_jiffies_cpuid*** is the CPU number whose time log has to be used to keep Linux timing and pacing in tune.
This function is specific to the MUP scheduler. If it is called with either the UP or SMP scheduler it will use:

- a periodic timer if all local APIC timers are periodic with the same period;
- a oneshot timer if all the local PAIC timers are oneshot, or have different timing modes, are periodic with different periods.

stop_rt_apic_timers stops all of the local APIC timers.

NAME

count2nano
count2nano_cpuid
nano2count
nano2count_cpuid

FUNCTION

Convert internal count units to nanosecs and back

SYNOPSIS

```
#include "rtai_sched.h"

RTIME count2nano (RTIME timercounts);
RTIME count2nano_cpuid (RTIME timercounts, int cpuid);
RTIME nano2count (RTIME nanosecs);
RTIME nano2count_cpuid (RTIME nanosecs, int cpuid);
```

DESCRIPTION

count2nano converts the time of ***timercounts*** internal count units into nanoseconds.
nano2count converts the time of ***nanosecs*** nanoseconds into internal counts units.
Remember that the count units are related to the time base being used (see functions **rt_set_oneshot_mode** and **rt_set_periodic_mode** for an explanation).
The versions ending with **_cpuid** are to be used with the MUP scheduler since with such a scheduler it is possible to have independent timers, i.e. periodic of different periods or a mixing of periodic and oneshot, so that it is impossible to establish which conversion units should be used in the case one asks for a conversion from any CPU for any other CPU. All these functions have the same behavior with UP and SMP schedulers.

RETURN VALUE

The given time in nanoseconds/internal count units is returned.

NAME

rt_get_time
rt_get_time_cpuid
rt_get_time_ns
rt_get_cpu_time_ns

FUNCTION

Get the current time

SYNOPSIS

```
#include "rtai_sched.h"

RTIME rt_get_time (void);
RTIME rt_get_time_cpuid (int cpuid);
RTIME rt_get_time_ns (void);
RTIME rt_get_cpu_time_ns (void);
RTIME rt_get_time_ns_cpuid (int cpuid);
```

DESCRIPTION

rt_get_time returns the time, in internal count units, since **start_rt_timer** was called. In periodic mode this number is in multiples of the periodic tick. In oneshot mode it is directly the TSC count for CPUs having a time stamp clock (TSC), while it is a on 8254 units for those not having it (see functions **rt_set_oneshot_mode and rt_set_periodic_mode** for an explanation).
rt_get_time_ns is the same as **rt_get_time** but the returned time is converted to nanoseconds.
rt_get_cpu_time_ns always returns the CPU time in nanoseconds whatever timer is in use.
The version ending with **_cpuid** must be used with the MUP scheduler when there is the need to declare from which ***cpuid*** the time must be got. In fact one can need to get the time of another CPU and timers can differ from CPU to CPU. All these functions have the same behavior with UP and SMP schedulers.

RETURN VALUE

The current time in internal count units/nanoseconds is returned.

NAME

next_period

FUNCTION

Get the time a periodic task will be resume after calling **rt_task_wait_period**

SYNOPSIS

```
#include "rtai_sched.h"

RTIME next_period (void);
```

DESCRIPTION

next_period returns the time when the caller task will run next. Combined with the appropriate **rt_get_time** function it can be used for checking the fraction of period used or any period overrun.

RETURN VALUE

Next period time in internal count units.

NAME

rt_busy_sleep
rt_sleep
rt_sleep_until

FUNCTION

Delay/suspend execution for a while

SYNOPSIS

```
#include "rtai_sched.h"

void rt_busy_sleep (int nanosecs);
void rt_sleep (RTIME delay);
void rt_sleep_until (RTIME time);
```

DESCRIPTION

rt_busy_sleep delays the execution of the caller task without giving back the control to the scheduler. This function burns away CPU cycles in a busy wait loop so it should be used only for very short synchronization delays. On machine not having a TSC clock it can lead to many microseconds uncertain busy sleeps because of the need of reading the 8254 timer.
nanosecs is the number of nanoseconds to wait.
rt_sleep suspends execution of the caller task for a time of ***delay*** internal count units. During this time the CPU is used by other tasks.
rt_sleep_until is similar to **rt_sleep** but the parameter ***time*** is the absolute time till the task have to be suspended. If the given time is already passed this call has no effect.

NOTES

A higher priority task or interrupt handler can run before the task goes to sleep, so the actual time spent in these functions may be longer than that specified.

Semaphore functions:

- rt_typed_sem_init
- rt_sem_init
- rt_sem_delete
- rt_sem_signal
- rt_sem_wait
- rt_sem_wait_if
- rt_sem_wait_until
- rt_sem_wait_timed

NAME

rt_typed_sem_init

FUNCTION

Initialize a specifically typed (counting, binary, resource) semaphore

SYNOPSIS

```
#include "rtai_sched.h"

void rt_typed_sem_init (SEM* sem, int value, int type);
```

DESCRIPTION

rt_typed_sem_init initializes a semaphore ***sem*** of type ***type***.
A semaphore can be used for communication and synchronization among real time tasks. Negative value of a semaphore shows how many tasks are blocked on the semaphore queue, waiting to be awaken by calls to **rt_sem_signal**.
sem must point to an allocated SEM structure.
value is the initial value of the semaphore, always set to `1` for a resource semaphore.
type is the semaphore type and can be: `CNT_SEM` for counting semaphores, `BIN_SEM` for binary semaphores, `RES_SEM` for resource semaphores.
Counting semaphores can register up to `0xFFFE` events.
Binary semaphores do not count signalled events, their count will never exceed `1` whatever number of events is signalled to them.
Resource semaphores are special binary semaphores suitable for managing resources. The task that acquires a resource semaphore becomes its owner, also called resource owner, since it is the only one capable of manipulating the resource the semaphore is protecting. The owner has its priority increased to that of any task blocking on a wait to the semaphore. Such a feature, called priority inheritance, ensures that a high priority task is never slaved to a lower priority one, thus allowing to avoid any deadlock due to priority inversion. Resource semaphores can be recursed, i.e. their task owner is not blocked by nested waits placed on an owned resource. The owner must insure that it will signal the semaphore, in reversed order, as many times as he waited on it. Note that that full priority inheritance is supported both for resource semaphores and inter task messages, for a singly owned resource. Instead it becomes an adaptive priority ceiling when a task owns multiple resources, including messages sent to him. In such a case in fact its priority is returned to its base one only when all such resources are released and no message is waiting for being received. This is a compromise design choice aimed at avoiding extensive searches for the new priority to be inherited across multiply owned resources and blocked tasks sending messages to him. Such a solution will be implemented only if it proves necessary. Note also that, to avoid deadlocks, a task owning a resource semaphore cannot be suspended. Any **rt_task_suspend** posed on it is just registered. An owner task will go into suspend state only when it releases all the owned resources.

RETURN VALUE

None

ERRORS

None

NOTES

RTAI counting semaphores assume that their counter will never exceed `0xFFFF`, such a number being used to signal returns in error. Thus also the initial count ***value*** cannot be greater than `0xFFFF`.
To be used only with **RTAI** 24.x.xx**.**

NAME

rt_sem_init

FUNCTION

Initialize a counting semaphore

SYNOPSIS

```
#include "rtai_sched.h"

void rt_sem_init (SEM* sem, int value);
```

DESCRIPTION

rt_sem_init initializes a semaphore ***sem***..
A semaphore can be used for communication and synchronization among real time tasks.
sem must point to an allocated SEM structure.
value is the initial value of the semaphore.
Positive values of the semaphore variable show how many tasks can do a **rt_sem_wait** call without blocking. Negative value of a semaphore shows how many tasks are blocked on the semaphore queue, waiting to be awaken by calls to **rt_sem_signal**.

RETURN VALUE

None

ERRORS

None

NOTES

RTAI counting semaphores assume that their counter will never exceed 0xFFFF, such a number being used to signal returns in error. Thus also the initial count ***value*** cannot be greater than 0xFFFF.

RTAI 24.1.xx has also **rt_typed_sem_init**, allowing to chose among counting, binary and resource semaphores. Resource semaphores have priority inherithance.

NAME

rt_sem_delete

FUNCTION

Delete a semaphore

SYNOPSIS

```
#include "rtai_sched.h"

int rt_sem_delete (SEM* sem);
```

DESCRIPTION

rt_sem_delete deletes a semaphore previously created with **rt_sem_init**.
sem points to the structure used in the corresponding call to **rt_sem_init**.
Any tasks blocked on this semaphore is returned in error and allowed to run when semaphore is destroyed.

RETURN VALUE

On success, 0 is returned. On failure, a nonzero value is returned, as described below.

ERRORS

0xFFFF ***sem*** does not refer to a valid semaphore.

NOTES

In principle 0xFFFF could theoretically be a usable semaphore's events count, so it could be returned also under normal circumstances. It is unlikely you are going to count up to such number of events, in any case avoid counting up to 0xFFFF.

NAME

rt_sem_signal

FUNCTION

Signalling a semaphore

SYNOPSIS

```
#include "rtai_sched.h"

int rt_sem_signal (SEM* sem);
```

DESCRIPTION

rt_sem_signal signal an event to a semaphore. It is typically called when the task leaves a critical region. The semaphore value is incremented and tested. If the value is not positive, the first task in semaphore's waiting queue is allowed to run.
rt_sem_signal never blocks the caller task.
sem points to the structure used in the call to **rt_sem_create**.

RETURN VALUE

On success, 0 is returned. On failure, a nonzero value is returned as described below.

ERRORS

0xFFFF ***sem*** does not refer to a valid semaphore.

NOTES

In principle 0xFFFF could theoretically be a usable semaphore's events count, so it could be returned also under normal circumstances. It is unlikely you are going to count up to such number of events, in any case avoid counting up to 0xFFFF.
See **rt_sem_wait** notes for some curiosities.

NAME

rt_sem_wait

FUNCTION

Take a semaphore

SYNOPSIS

```
#include "rtai_sched.h"

int rt_sem_wait (SEM* sem);
```

DESCRIPTION

rt_sem_wait waits for a event to be signalled to a semaphore. It is typically called when a task enters a critical region. The semaphore value is decremented and tested. If it is still non-negative **rt_sem_wait** returns immediately. Otherwise the caller task is blocked and queued up. Queuing may happen in priority order or on FIFO base. This is determined by the compile time option SEM_PRIORD. In this case **rt_sem_wait** returns if

- The caller task is in the first place of the waiting queue and an other task issues a **rt_sem_signal** call;
- An error occurs (e.g. the semaphore is destroyed);

sem points to the structure used in the call to **rt_sem_create**.

RETURN VALUE

On success, the number of number of events already signalled is returned.
On failure, the special value 0xFFFF is returned as described below.

ERRORS

0xFFFF ***sem*** does not refer to a valid semaphore.

NOTES

In principle 0xFFFF could theoretically be a usable semaphore's events count, so it could be returned also under normal circumstances. It is unlikely you are going to count up to such number of events, in any case avoid counting up to 0xFFFF.

Just for curiosity: the original Dijkstra notation for **rt_sem_wait** was a "P" operation, and **rt_sem_signal** was a "V" operation.
The name for P comes from the Dutch "prolagen", a combination of "proberen" (to probe) and "verlagen" (to decrement). Also from the word "passeren" (to pass).
The name for V comes from the Dutch "verhogen" (to increase) or "vrygeven" (to release).
(Source: Daniel Tabak - Multiprocessors, Prentice Hall, 1990).
It should be also remarked that real time programming practitioners were using semaphores a long time before Dijkstra formalized P and V.
In Italian "semaforo" means a traffic light, so that semaphores have an intuitive appeal and their use and meaning is easily understood.

NAME

rt_sem_wait_if

FUNCTION

Take a semaphore, only if the calling task is not blocked

SYNOPSIS

```
#include "rtai_sched.h"

int rt_sem_wait_if (SEM* sem);
```

DESCRIPTION

rt_sem_wait_if is a version of the semaphore wait operation is similar to **rt_sem_wait** but it is never blocks the caller. If the semaphore is not free, **rt_sem_wait_if** returns immediately and the semaphore value remains unchanged.

RETURN VALUE

On success, the number of number of events already signalled is returned.
On failure, the special value 0xFFFF is returned as described below.

ERRORS

0xFFFF ***sem*** does not refer to a valid semaphore.

NOTES

In principle 0xFFFF could theoretically be a usable semaphore's events count so it could be returned also under normal circumstances. It is unlikely you are going to count up to such number of events, in any case avoid counting up to 0xFFFF.

NAME

rt_sem_wait_until
rt_sem_wait_timed

FUNCTION

Wait a semaphore with timeout

SYNOPSIS

```
#include "rtai_sched.h"

int rt_sem_wait_until (SEM* sem, RTIME time);
int rt_sem_wait_timed (SEM* sem, RTIME delay);
```

DESCRIPTION

rt_sem_wait_until and **rt_sem_wait_timed** are timed version of the standard semaphore wait call. The semaphore value is decremented and tested. If it is still non-negative these functions return immediately. Otherwise the caller task is blocked and queued up. Queuing may happen in priority order or on FIFO base. This is determined by the compile time option SEM_PRIORD. In this case these functions return if

- The caller task is in the first place of the waiting queue and an other task issues a **rt_sem_signal** call;
- Timeout occurs;
- An error occurs (e.g. the semaphore is destroyed);

In case of timeout the semaphore value is incremented before return.
time is an absolute value, ***delay*** is relative to the current time.

RETURN VALUE

On success, the number of number of events already signalled is returned.
On failure, the special value 0xFFFF is returned as described below.

ERRORS

0xFFFF ***sem*** does not refer to a valid semaphore.

NOTES

In principle 0xFFFF could theoretically be a usable semaphore's events count so it could be returned also under normal circumstances. It is unlikely you are going to count up to such number of events, in any case avoid counting up to 0xFFFF.

Inter tasks message handling functions:

- rt_send
- rt_send_if
- rt_send_until
- rt_send_timed
- rt_receive
- rt_receive_if
- rt_receive_until
- rt_receive_timed

NAME

rt_send

FUNCTION

Send a message

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK* rt_send (RT_TASK* task, unsigned int msg);
```

DESCRIPTION

rt_send sends the message ***msg*** to the task ***task***. If the receiver task is ready to get the message **rt_send** does not block the sending task, but its execution can be preempted if the receiving task has a higher priority. Otherwise the caller task is blocked and queued up. (Queuing may happen in priority order or on FIFO base. This is determined by the compile time option MSG_PRIORD.)

RETURN VALUE

On success, ***task***, the pointer to the task that received the message, is returned.
If the caller is unblocked but the message has not been sent, e.g. the task ***task*** was killed before receiving the message, 0 is returned.
On other failure, a special value 0xFFFF is returned as described below.

ERRORS

0	The receiver task was killed before receiving the message.
0xFFFF	***task*** does not refer to a valid task.

NOTES

Since all the messaging functions return a task address 0xFFFF could seem an inappropriate return value. However on all the CPUs **RTAI** runs on 0xFFFF is not an address that can be used by any **RTAI** task, so it is should be always safe.

NAME

rt_send_if

FUNCTION

Send a message, only if the calling task is not blocked

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK* rt_send_if (RT_TASK* task, unsigned int msg);
```

DESCRIPTION

rt_send_if sends the message ***msg*** to the task ***task*** if the latter is ready to receive, so that the caller task is never blocked, but its execution can be preempted if the messaged task is ready to receive and has a higher priority.

RETURN VALUE

On success, ***task*** (the pointer to the task that received the message) is returned.
If message has not been sent, 0 is returned.
On other failure, a special value 0xFFFF is returned as described below.

ERRORS

0	The task ***task*** was not ready to receive the message.
0xFFFF	***task*** does not refer to a valid task.

NOTES

Since all the messaging functions return a task address 0xFFFF could seem an inappropriate return value. However on all the CPUs **RTAI** runs on 0xFFFF is not an address that can be used by any **RTAI** task, so it is should be always safe.

NAME

rt_send_until
rt_send_timed

FUNCTION

Send a message with timeout

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK* rt_send_until (RT_TASK* task, unsigned int msg, RTIME
       time);
RT_TASK* rt_send_timed (RT_TASK* task, unsigned int msg, RTIME
       delay);
```

DESCRIPTION

rt_send_until and **rt_send_timed** send the message ***msg*** to the task ***task***. If the receiver task is ready to get the message these functions do not block the sending task, but its execution can be preempted if the receiving task has a higher priority. Otherwise the caller task is blocked and queued up. (Queuing may happen in priority order or on FIFO base. This is determined by the compile time option MSG_PRIORD). In this case these functions return if

- The caller task is in the first place of the waiting queue and the receiver gets the message and has a lower priority;
- Timeout occurs;
- An error occurs (e.g. the receiver task is killed);

time is an absolute value, ***delay*** is relative to the current time.

RETURN VALUE

On success, i.e. message received before timout expiration, ***task*** (the pointer to the task that received the message) is returned.
If message has not been sent, `0` is returned.
On other failure, a special value `0xFFFF` is returned as described below.

ERRORS

`0`	Operation timed out, message was not delivered.
`0xFFFF`	***task*** does not refer to a valid task.

NOTES

Since all the messaging functions return a task address `0xFFFF` could seem an inappropriate return value. However on all the CPUs **RTAI** runs on `0xFFFF` is not an address that can be used by any **RTAI** task, so it is should be always safe.

NAME

rt_receive

FUNCTION

Receive a message

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK* rt_receive (RT_TASK* task, unsigned int *msg);
```

DESCRIPTION

rt_receive gets a message from the task specified by ***task***.
If ***task*** is equal to 0, the caller accepts messages from any task. If there is a pending message, **rt_receive** does not block but can be preempted if the task that sent the just received message has a higher priority. Otherwise the caller task is blocked and queued up. (Queuing may happen in priority order or on FIFO base. This is determined by the compile time option MSG_PRIORD.) ***msg*** points to any 4 bytes word buffer provided by the caller.

RETURN VALUE

On success, a pointer to the sender task is returned.
If the caller is unblocked but no message has been received (e.g. the task ***task*** was killed before sending the message) 0 is returned.
On other failure, a special value 0xFFFF is returned as described below.

ERRORS

0	The sender task was killed before sending the message.
0xFFFF	***task*** does not refer to a valid task.

NOTES

Since all the messaging functions return a task address 0xFFFF could seem an inappropriate return value. However on all the CPUs **RTAI** runs on 0xFFFF is not an address that can be used by any **RTAI** task, so it is should be always safe.

NAME

rt_receive_if

FUNCTION

Receive a message, only if the calling task is not blocked

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK* rt_receive_if (RT_TASK* task, unsigned int *msg);
```

DESCRIPTION

rt_receive_if tries to get a message from the task specified by ***task***. If ***task*** is equal to 0, the caller accepts messages from any task. The caller task is never blocked but can be preempted if the receiving task is ready to receive and has a higher priority.
msg points to a buffer provided by the caller.

RETURN VALUE

On success, a pointer to the sender task is returned.
If no message has been received, 0 is returned.
On other failure, a special value 0xFFFF is returned as described below.

ERRORS

0	There was no message to receive.
0xFFFF	***task*** does not refer to a valid task.

NOTES

Since all the messaging functions return a task address 0xFFFF could seem an inappropriate return value. However on all the CPUs **RTAI** runs on 0xFFFF is not an address that can be used by any **RTAI** task, so it is should be always safe.

NAME

rt_receive_until
rt_receive_timed

FUNCTION

Receive a message with timeout

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK* rt_receive_until (RT_TASK* task, unsigned int *msg, RTIME
       time);
RT_TASK* rt_receive_timed (RT_TASK* task, unsigned int *msg, RTIME
       delay);
```

DESCRIPTION

rt_receive_until and **rt_receive_timed** receive a message from the task specified by ***task***. If ***task*** is equal to 0, the caller accepts messages from any task. If there is a pending message, **rt_receive** does not block but can be preempted if the task that sent the just received message has a higher priority. Otherwise the caller task is blocked and queued up. (Queuing may happen in priority order or on FIFO base. This is determined by the compile time option MSG_PRIORD). In this case these functions return if

- The caller task is in the first place of the waiting queue and the sender sends a message and has a lower priority;
- Timeout occurs;
- An error occurs (e.g. the sender task is killed);

msg points to a buffer provided by the caller.
time is an absolute value, ***delay*** is relative to the current time.

RETURN VALUE

On success, a pointer to the sender task is returned.
If no message has been received, 0 is returned.
On other failures, a special value 0xFFFF is returned as described below.

ERRORS

0	Operation timed out, no message was received.
0xFFFF	***task*** does not refer to a valid task.

NOTES

Since all the messaging functions return a task address 0xFFFF could seem an inappropriate return value. However on all the CPUs **RTAI** runs on 0xFFFF is not an address that can be used by any **RTAI** task, so it is should be always safe.

Inter tasks Remote Procedure Call (RPC) functions:

- rt_rpc
- rt_rpc_if
- rt_rpc_until
- rt_rpc_timed
- rt_isrpc
- rt_return

NAME

rt_rpc

FUNCTION

Make a remote procedure call

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK *rt_rpc (RT_TASK *task, unsigned int msg, unsigned int
      *reply);
```

DESCRIPTION

rt_rpc makes a Remote Procedure Call (RPC). **rt_rpc** is used for synchronous inter task messaging as it sends the message ***msg*** to the task ***task*** then it always block waiting until a return is received from the called task. So the caller task is always blocked and queued up. (Queuing may happen in priority order or on FIFO base. This is determined by the compile time option MSG_PRIORD). The receiver task may get the message with any **rt_receive** function. It can send an answer with **rt_return**. ***reply*** points to a buffer provided by the caller were the returned result message, any 4 bytes integer, is to be place.

RETURN VALUE

On success, ***task*** (the pointer to the task that received the message) is returned.
If message has not been sent (e.g. the task ***task*** was killed before receiving the message) `0` is returned.
On other failure, a special value `0xFFFF` is returned as described below.

ERRORS

`0`	The receiver task was killed before receiving the message.
`0xFFFF`	***task*** does not refer to a valid task.

SEE ALSO

rt_receive_*, **rt_return**, **rt_isrpc**.

NOTES

Since all the messaging functions return a task address `0xFFFF` could seem an inappropriate return value. However on all the CPUs **RTAI** runs on `0xFFFF` is not an address that can be used by any **RTAI** task, so it is should be always safe.

The trio **rt_rpc, rt_receive, rt_return** implement functions similar to its peers send-receive-replay found in QNX, except that in RTAI only four bytes messages contained in any integer can be exchanged. That's so because we never needed anything different. Note also that we prefer the idea of calling a function by using a message and then wait for a return value since it is believed to give a better idea of what is meant for synchronous message passing. For a truly QNX like way of inter task messaging use the support module found in directory **lxrt-informed**.

NAME

rt_rpc_if

FUNCTION

Make a remote procedure call, only if the calling task is not blocked

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK *rt_rpc_if (RT_TASK *task, unsigned int msg, unsigned int
      *reply);
```

DESCRIPTION

rt_rpc_if tries to make a Remote Procedure Call (RPC). If the receiver task is ready to accept a message **rt_rpc_if** sends the message ***msg*** then it always block until a return is received. In this case the caller task is blocked and queued up. (Queuing may happen in priority order or on FIFO base. This is determined by the compile time option MSG_PRIORD). If the receiver is not ready **rt_rpc_if** returns immediately. The receiver task may get the message with any **rt_receive** function. It can send the answer with **rt_return**. ***reply*** points to a buffer provided by the caller.

RETURN VALUE

On success, ***task*** (the pointer to the task that received the message) is returned.
If message has not been sent, `0` is returned.
On other failure, a special value `0xFFFF` is returned as described below.

ERRORS

`0`	The task ***task*** was not ready to receive the message or it was killed before sending the reply.
`0xFFFF`	***task*** does not refer to a valid task.

SEE ALSO

rt_receive, **rt_return**, **rt_isrpc**.

NOTES

Since all the messaging functions return a task address `0xFFFF` could seem an inappropriate return value. However on all the CPUs **RTAI** runs on `0xFFFF` is not an address that can be used by any **RTAI** task, so it is should be always safe.
See also the NOTES under **rt_rpc.**

NAME

rt_rpc_until
rt_rpc_timed

FUNCTION

Make a remote procedure call with timeout

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK *rt_rpc_until (RT_TASK *task, unsigned int msg, unsigned
       int *reply, RTIME time);
RT_TASK *rt_rpc_timed (RT_TASK *task, unsigned int msg, unsigned
       int *reply, RTIME delay);
```

DESCRIPTION

rt_rpc_until and **rt_rpc_timed** make a Remote Procedure Call. They send the message ***msg*** to the task ***task*** then always wait until a return is received or a timeout occurs. So the caller task is always blocked and queued up. (Queuing may happen in priority order or on FIFO base. This is determined by the compile time option MSG_PRIORD). The receiver task may get the message with any **rt_receive** function. It can send the answer with **rt_return**. ***reply*** points to a buffer provided by the caller.
time is an absolute value, ***delay*** is relative to the current time.

RETURN VALUE

On success, ***task*** (the pointer to the task that received the message) is returned.
If message has not been sent or no answer arrived, 0 is returned.
On other failure, a special value 0xFFFF is returned as described below.

ERRORS

0	The message could not be sent or the answer did not arrived in time.
0xFFFF	***task*** does not refer to a valid task.

SEE ALSO

rt_receive, **rt_return**, **rt_isrpc**.

NOTES

Since all the messaging functions return a task address 0xFFFF could seem an inappropriate return value. However on all the CPUs **RTAI** runs on 0xFFFF is not an address that can be used by any **RTAI** task, so it is should be always safe.
See also the NOTES under **rt_rpc.**

NAME

rt_isrpc

FUNCTION

Check if sender waits for reply or not

SYNOPSIS

```
#include "rtai_sched.h"

int rt_isrpc (RT_TASK *task);
```

DESCRIPTION

After receiving a message, by calling **rt_isrpc** a task can figure out whether the sender task ***task*** is waiting for a reply or not. That can be needed in the case one needs a server task that must provide services both to **send**s and **rt_rtc**s.
No answer is required if the message sent by a **rt_send** function or the sender called **rt_rpc_timed** or **rt_rpc_until** but it is already timed out.

RETURN VALUE

If the ***task*** waits for a reply, a nonzero value is returned.
Otherwise `0` is returned.

NOTES

rt_isrpc does not perform any check on pointer ***task***.
rt_isrpc cannot figure out what RPC result the sender is waiting for.

rt_return is intelligent enough to not send an answer to a task which is not waiting for it. Therefore using **rt_isrpc** is not necessary and discouraged.

NAME

rt_return

FUNCTION

Send (return) the result back to the task that made the related remote procedure call

SYNOPSIS

```
#include "rtai_sched.h"

RT_TASK *rt_return (RT_TASK *task, unsigned int result);
```

DESCRIPTION

rt_return sends the result ***result*** to the task ***task***. If the task calling **rt_rpc** previously is not waiting the answer (i.e. killed or timed out) this return message is silently discarded.

RETURN VALUE

On success, ***task*** (the pointer to the task that is got the reply) is returned.
If the reply message has not been sent, 0 is returned.
On other failure, a special value 0xFFFF is returned as described below.

ERRORS

0	The reply message was not delivered.
0xFFFF	***task*** does not refer to a valid task.

NOTES

Since all the messaging functions return a task address 0xFFFF could seem an inappropriate return value. However on all the CPUs **RTAI** runs on 0xFFFF is not an address that can be used by any **RTAI** task, so it is should be always safe.
See also the NOTES under **rt_rpc.**

Mailbox functions:

- rt_mbx_init
- rt_mbx_delete
- rt_mbx_send
- rt_mbx_send_wp
- rt_mbx_send_if
- rt_mbx_send_until
- rt_mbx_send_timed
- rt_mbx_receive
- rt_mbx_receive_wp
- rt_mbx_receive_if
- rt_mbx_receive_until
- rt_mbx_receive_timed

NAME

rt_mbx_init

FUNCTION

Initialize mailbox

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_init (MBX* mbx, int size);
```

DESCRIPTION

rt_mbx_init initializes a mailbox of size ***size***. ***mbx*** must point to a user allocated MBX structure.
Using mailboxes is a flexible method for inter task communications. Tasks are allowed to send arbitrarily sized messages by using any mailbox buffer size. There is even no need to use a buffer sized at least as the largest message you envisage, even if efficiency is likely to suffer from such a decision. However if you expect a message larger than the average message size very rarely you can use a smaller buffer without much loss of efficiency. In such a way you can set up your own mailbox usage protocol, e.g. using fix sized messages with a buffer that is an integer multiple of such a size guarantees maximum efficiency by having each message sent/received atomically to/from the mailbox. Multiple senders and receivers are allowed and each will get the service it requires in turn, according to its priority.
Thus mailboxes provide a flexible mechanism to allow you to freely implement your own policy.

RETURN VALUE

On success 0 is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL Space could not be allocated for the mailbox buffer.

NAME

rt_mbx_delete

FUNCTION

Delete mailbox

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_delete (MBX* mbx);
```

DESCRIPTION

rt_mbx_delete removes a mailbox previously created with **rt_mbox_init**. ***mbx*** points to the structure used in the corresponding call to **rt_mbox_init**.

RETURN VALUE

On success 0 is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL	***mbx*** points to not a valid mailbox.
EFAULT	mailbox data were found in an invalid state.

NAME

rt_mbx_send

FUNCTION

Send message unconditionally

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_send (MBX* mbx, void* msg, int msg_size);
```

DESCRIPTION

rt_mbx_send sends a message ***msg*** of ***msg_size*** bytes to the mailbox ***mbx***. The caller will be blocked until the whole message is copied into the mailbox or an error occurs. Even if the message can be sent in a single shot the sending task can be blocked if there is a task of higher priority waiting to receive from the mailbox.

RETURN VALUE

On success, the number of unsent bytes is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

NAME

rt_mbx_send_wp

FUNCTION

Send as many bytes as possible, without blocking the calling task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_send_wp (MBX* mbx, void* msg, int msg_size);
```

DESCRIPTION

rt_mbx_send_wp atomically sends as many bytes of message ***msg*** as possible to mailbox ***mbx*** then returns immediately. The message length is ***msg_size***.

RETURN VALUE

On success, the number of unsent bytes is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

NAME

rt_mbx_send_if

FUNCTION

Send a message, only if the whole message can be passed without blocking the calling task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_send_if (MBX* mbx, void* msg, int msg_size);
```

DESCRIPTION

rt_mbx_send_if tries to atomically send the message ***msg*** of ***msg_size*** bytes to the mailbox ***mbx***. It returns immediately, the caller is never blocked.

RETURN VALUE

On success, the number of unsent bytes (0 or ***msg_size***) is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

NAME

rt_mbx_send_until
rt_mbx_send_timed

FUNCTION

Send a message with timeout

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_send_until (MBX* mbx, void* msg, int msg_size, RTIME
      time);
int rt_mbx_send_timed (MBX* mbx, void* msg, int msg_size, RTIME
      delay);
```

DESCRIPTION

rt_mbx_send_until and **rt_mbx_send_timed** send a message ***msg*** of ***msg_size*** bytes to the mailbox ***mbx***. The caller will be blocked until all bytes of message is enqueued, timeout expires or an error occurs.
time is an absolute value.
delay is relative to the current time.

RETURN VALUE

On success, the number of unsent bytes is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

NAME

rt_mbx_receive

FUNCTION

Receive message unconditionally

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_receive (MBX* mbx, void* msg, int msg_size);
```

DESCRIPTION

rt_mbx_receive receives a message of ***msg_size*** bytes from the mailbox ***mbx***. The caller will be blocked until all bytes of the message arrive or an error occurs.
msg points to a buffer provided by the caller.

RETURN VALUE

On success, the number of received bytes is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

NAME

rt_mbx_receive_wp

FUNCTION

Receive bytes as many as possible, without blocking the calling task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_receive_wp (MBX* mbx, void* msg, int msg_size);
```

DESCRIPTION

rt_mbx_receive_wp receives at most ***msg_size*** of bytes of message from mailbox ***mbx*** then returns immediately.
msg points to a buffer provided by the caller.

RETURN VALUE

On success, the number of received bytes is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

NAME

rt_mbx_receive_if

FUNCTION

Receive a message, only if the whole message can be passed without blocking the calling task

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_receive_if (MBX* mbx, void* msg, int msg_size);
```

DESCRIPTION

rt_mbx_receive_if receives a message from the mailbox ***mbx*** if the whole message of ***msg_size*** bytes is available immediately.
msg points to a buffer provided by the caller.

RETURN VALUE

On success, the number of received bytes (0 or ***msg_size***) is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

NAME

rt_mbx_receive_until
rt_mbx_receive_timed

FUNCTION

Receive a message with timeout

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_receive_until (MBX* mbx, void* msg, int msg_size, RTIME
      time);
int rt_mbx_receive_timed (MBX* mbx, void* msg, int msg_size, RTIME
      delay);
```

DESCRIPTION

rt_mbx_receive_until and **rt_mbx_receive_timed** receive a message of ***msg_size*** bytes from the mailbox ***mbx***. The caller will be blocked until all bytes of the message arrive, timeout expires or an error occurs.
time is an absolute value. ***delay*** is relative to the current time.
msg points to a buffer provided by the caller.

RETURN VALUE

On success, the number of received bytes is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL ***mbx*** points to not a valid mailbox.

Functions provided by rtai module

Following there are some function calls, that can be used by **RTAI** tasks, for managing interrupts and communication services with Linux processes.

RTAI service functions:

- rt_global_cli
- rt_global_sti
- rt_global_save_flags
- rt_global_save_flags_and_cli
- rt_global_restore_flags
- rt_startup_irq
- rt_shutdown_irq
- rt_enable_irq
- rt_disable_irq
- rt_mask_and_ack_irq
- rt_unmask_irq
- rt_ack_irq
- send_ipi_shorthand
- send_ipi_logical
- rt_assign_irq_to_cpu
- rt_reset_irq_to_sym_mode
- rt_request_global_irq
- rt_free_global_irq
- request_RTirq
- free_RTirq
- rt_request_linux_irq
- rt_free_linux_irq
- rt_pend_linux_irq
- rt_request_srq
- rt_free_srq
- rt_pend_linux_srq
- rt_request_timer
- rt_free_timer
- rt_request_apic_timers
- rt_free_apic_timers
- rt_mount_rtai
- rt_umount_rtai

NAME

rt_global_cli
rt_global_sti

FUNCTION

Disable/enable interrupts across all CPUs

SYNOPSIS

```
#include "rtai.h"

void rt_global_cli (void);
void rt_global_sti (void);
```

DESCRIPTION

rt_global_cli hard disables interrupts (cli) on the requesting CPU and acquires the global spinlock to the calling CPU so that any other CPU synchronized by this method is blocked. Nested calls to **rt_global_cli** within the owner CPU will not cause a deadlock on the global spinlock, as it would happen for a normal spinlock.
rt_global_sti hard enables interrupts (sti) on the calling CPU and releases the global lock.

NAME

rt_global_save_flags
rt_global_save_flags_and_cli
rt_global_restore_flags

FUNCTION

Save/restore CPU flags

SYNOPSIS

```
#include "rtai.h"

void rt_global_save_flags (unsigned long *flags);
int rt_global_save_flags_and_cli (void);
void rt_global_restore_flags (unsigned long flags);
```

DESCRIPTION

rt_global_save_flags saves the CPU interrupt flag (IF) bit 9 of ***flags*** and ORs the global lock flag in the first 8 bits of ***flags***. From that you can rightly infer that **RTAI** does not support more than 8 CPUs.
rt_global_save_flags_and_cli combines **rt_global_save_flags** and **rt_global_cli**.
rt_global_restore_flags restores the CPU hard interrupt flag (IF) and the global lock flag as given by ***flags***, freeing or acquiring the global lock according to the state of the global flag bit found in the bit corresponding to the CPU it is called.

NAME

rt_startup_irq
rt_shutdown_irq
rt_enable_irq
rt_disable_irq
rt_mask_and_ack_irq
rt_unmask_irq
rt_ack_irq

FUNCTION

Programmable Interrupt Controllers (PIC) management functions.

SYNOPSIS

```
#include "rtai.h"

void rt_startup_irq (unsigned int irq);
void rt_shutdown_irq (unsigned int irq);
void rt_enable_irq (unsigned int irq);
void rt_disable_irq (unsigned int irq);
void rt_mask_and_ack_irq (unsigned int irq);
void rt_unmask_irq (unsigned int irq);
void rt_ack_irq (unsigned int irq);
```

DESCRIPTION

rt_startup_irq start and initialize the PIC to accept interrupt request ***irq***.
rt_shoutdown_irq shut down the PIC so that no further interrupt request ***irq*** can be accepted.
rt_enable_irq enable PIC interrupt request ***irq***.
rt_disable_irq disable PIC interrupt request ***irq***.
rt_mask_and_ack_irq mask PIC interrupt request ***irq*** and acknowledge it so that other interrupts can be accepted, once also the CPU will enable interrupts, which ones depends on the PIC at hand and on how it is programmed
rt_unmask_irq unmask PIC interrupt request ***irq*** so that the related request can interrupt the CPU again, provided it has also been acknowledged.
rt_ack_irq acknowledge PIC interrupt request ***irq*** so that the related request can interrupt the CPU again, provided it has not been masked.
The above functions allow you to manipulate the PIC at hand, but you must know what you are doing. Such a duty does not pertain to this manual and you should refer to your PIC datasheet. Note that Linux has the same functions, but they must be used only for its interrupts. Only the above ones can be safely used in real time handlers.
It must also be remarked that when you install a real time interrupt handler, **RTAI** already calls either **rt_mask_and_ack_irq**, for level triggered interrupts, or **rt_ack_irq**, for edge triggered interrupts, before passing control to you interrupt handler. Thus generally you should just call **rt_unmask_irq** at due time, for level triggered interrupts, while nothing should be done for edge triggered ones. Recall that in the latter case you allow also any new interrupts on the same request as soon as you enable interrupts at the CPU level.
Often some of the above functions do equivalent things. Once more there is no way of doing it right except by knowing the hardware you are manipulating.
Furthermore you must also remember that when you install a hard real time handler the related interrupt is usually disabled, unless you are overtaking one already owned by Linux which has been enabled by it. Recall that if have done it right, and interrupts do not show up, it is likely you have just to **rt_enable_irq** your ***irq***.

RETURN VALUE

None.

ERRORS

None.

NOTES

In 24.1.xx **`rt_startup_irq`** is not of type `void`, instead it returns an `unsigned long`.

NAME

send_ipi_shorthand
send_ipi_logical

FUNCTION

Send an inter processors message

SYNOPSIS

```
#include "rtai.h"

void send_ipi_shorthand (unsigned int shorthand, int irq);
void send_ipi_logical (unsigned long dest, int irq);
```

DESCRIPTION

send_ipi_shorthand sends an inter processors message corresponding to ***irq*** on:

- all CPUs if ***shorthand*** is equal to APIC_DEST_ALLINC;
- all but itself if ***shorthand*** is equal to APIC_DEST_ALLBUT;
- itself if ***shorthand*** is equal to APIC_DEST_SELF.

send_ipi_logical sends an inter processor message to ***irq*** on all CPUs defined by ***dest***. ***dest*** is given by an unsigned long corresponding to a bits mask of the CPUs to be sent. It is used for local APICs programmed in flat logical mode, so the max number of allowed CPUs is 8, a constraint that is valid for all functions and data of **RTAI**. The flat logical mode is set when **RTAI** is installed by calling **rt_mount_rtai**. Linux 2.4.xx needs no more to be reprogrammed has it has adopted the same idea.

NOTES

Inter processor messages are not identified by an irq number but by the corresponding vector. Such a correspondence is wired internally in **RTAI** internal tables.

NAME

rt_assign_irq_to_cpu
rt_reset_irq_to_sym_mode

FUNCTION

Set/reset IRQ->CPU assignment

SYNOPSIS

```
#include "rtai.h"

int rt_assign_irq_to_cpu (int irq, int cpu);
int rt_reset_irq_to_sym_mode (int irq);
```

DESCRIPTION

rt_assign_irq_to_cpu forces the assignment of the external interrupt ***irq*** to the CPU ***cpu***.
rt_reset_irq_to_sym_mode resets the interrupt ***irq*** to the symmetric interrupts management. The symmetric mode distributes the IRQs over all the CPUs.

RETURN VALUE

If there is one CPU in the system, `1` returned.
If there are at least 2 CPUs, on success `0` is returned.
If ***cpu*** refers to a non-existent CPU, the number of CPUs is returned.
On other failures, a negative value is returned as described below.

ERRORS

`EINVAL` ***irq*** is not a valid IRQ number or some internal data inconsistency is found.

NOTES

These functions have effect only on multiprocessors systems.
With Linux 2.4.xx such a service has finally been made available natively within the raw kernel. With such Linux releases **rt_reset_irq_to_sym_mode** resets the original Linux delivery mode, or deliver affinity as they call it. So be warned that such a name is kept mainly for compatibility reasons, as for such a kernel the reset operation does not necessarily implies a symmetric external interrupt delivery.

NAME

rt_request_global_irq
request_RTirq
rt_free_global_irq
free_RTirq

FUNCTION

Install/uninstall IT service routine

SYNOPSIS

```
#include "rtai.h"

int rt_request_global_irq (unsigned int irq, void
       (*handler)(void));
int rt_free_global_irq (unsigned int irq);
int request_RTirq (unsigned int irq, void (*handler)(void));
int free_RTirq (unsigned int irq);
```

DESCRIPTION

rt_request_global_irq installs function ***handler*** as a real time interrupt service routine for IRQ level ***irq,*** eventually stealing it to Linux.
handler is then invoked whenever interrupt number ***irq*** occurs. The installed handler must take care of properly activating any Linux handler using the same ***irq*** number he stole, by calling **rt_pend_linux_irq**.
rt_free_global_irq uninstalls the interrupt service routine, resetting it for Linux if it was previously owned by the kernel.
request_RTirq and **free_RTirq** are macros defined in `rtai.h` and is supported only for backwards compatibility with our variant of RT_linux for 2.0.35. They are fully equivalent of the two functions above.

RETURN VALUE

On success `0` is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL	***irq*** is not a valid IRQ number or ***handler*** is NULL.
EBUSY	There is already a handler of interrupt ***irq***.

NAME

rt_request_linux_irq
rt_free_linux_irq

FUNCTION

Install/uninstall shared Linux interrupt handler

SYNOPSIS

```
#include "rtai.h"

int rt_request_linux_irq (unsigned int irq, void (*handler)(int
      irq, void *dev_id, struct pt_regs *regs), char
      *linux_handler_id, void *dev_id);
int rt_free_linux_irq (unsigned int irq, void *dev_id);
```

DESCRIPTION

rt_request_linux_irq installs function ***handler*** as a standard Linux interrupt service routine for IRQ level ***irq*** forcing Linux to share the IRQ with other interrupt handlers, even if it does not want. The handler is appended to any already existing Linux handler for the same ***irq*** and is run by Linux ***irq*** as any of its handler. In this way a real time application can monitor Linux interrupts handling at its will. The handler appears in `/proc/interrupts`.
linux_handler_id is a name for `/proc/interrupts`. The parameter ***dev_id*** is to pass to the interrupt handler, in the same way as the standard Linux ***irq*** request call.
The interrupt service routine can be uninstalled with **rt_free_linux_irq**.

RETURN VALUE

On success `0` is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL	***irq*** is not a valid IRQ number or ***handler*** is NULL.
EBUSY	There is already a handler of interrupt ***irq***.

NAME

rt_pend_linux_irq

FUNCTION

Make Linux service an interrupt

SYNOPSIS

```
#include "rtai.h"

void rt_pend_linux_irq (unsigned int irq);
```

DESCRIPTION

rt_pend_linux_irq appends a Linux interrupt ***irq*** for processing in Linux IRQ mode, i.e. with hardware interrupts fully enabled.

NOTES

rt_pend_linux_irq does not perform any check on ***irq***.

NAME

rt_request_srq
rt_free_srq

FUNCTION

Install/Uninstall a system request handler

SYNOPSIS

```
#include "rtai.h"
#include "rtai_srq.h"

int rt_request_srq (unsigned int label, void (*rtai_handler)(void),
       long long (*user_handler)(unsigned int whatever));
int rt_free_srq (unsigned int srq);
```

DESCRIPTION

rt_request_srq installs a two way **RTAI** system request (srq) by assigning ***user_handler***, a function to be used when a user calls srq from user space, and ***rtai_handler*** , the function to be called in kernel space following its activation by a call to **rt_pend_linux_srq**. ***rtai_handler*** is in practice used to request a service from the kernel. In fact Linux system requests cannot be used safely from **RTAI** so you can setup a handler that receives real time requests and safely executes them when Linux is running.
user_handler can be used to effectively enter kernel space without the overhead and clumsiness of standard Unix/Linux protocols. This is very flexible service that allows you to personalize your use of **RTAI**.

rt_free_srq uninstalls the specified system call ***srq***, returned by installing the related handler with a previous call to **rt_request_srq**

RETURN VALUE

On success the number of the assigned system request is returned.
On failure a negative value is returned as described below.

ERRORS

EINVAL	***rtai_handler*** is NULL or ***srq*** is invalid.
EBUSY	No free ***srq*** slot is available.

NAME

rt_pend_linux_srq

FUNCTION

Append a Linux IRQ

SYNOPSIS

```
#include "rtai.h"

void rt_pend_linux_srq (unsigned int srq);
```

DESCRIPTION

rt_pend_linux_srq appends a system call request ***srq*** to be used as a service request to the Linux kernel. ***srq*** is the value returned by **rt_request_srq**.

NOTES

rt_pend_linux_srq does not perform any check on ***irq***.

NAME

rt_request_timer
rt_free_timer

FUNCTION

Install a timer interrupt handler

SYNOPSIS

```
#include "rtai.h"

void rt_request_timer (void (*handler)(void), int tick, int apic);
void rt_free_timer (void);
```

DESCRIPTION

rt_request_timer requests a timer of period ***tick*** ticks, and installs the routine ***handler*** as a real time interrupt service routine for the timer. Set ***tick*** to 0 for oneshot mode (in oneshot mode it is not used). If ***apic*** has a nonzero value the local APIC timer is used. Otherwise timing is based on the 8254.
rt_free_timer uninstalls the timer previously set by **rt_request_timer**.

NAME

rt_request_apic_timers
rt_free_apic_timers

FUNCTION

Install a local APICs timer interrupt handler

SYNOPSIS

```
#include "rtai.h"

void rt_request_apic_timers (void (*handler)(void), struct
       apic_timer_setup_data *apic_timer_data);
void rt_free_apic_timers (void);
```

DESCRIPTION

rt_request_apic_timers requests local APICs timers and defines the mode and count to be used for each local APIC timer. Modes and counts can be chosen arbitrarily for each local APIC timer. ***apic_timer_data*** is a pointer to a vector of structures `struct apic_timer_setup_data { int mode, count; }` sized with the number of CPUs available.
Such a structure defines:

- `mode`: `0` for a oneshot timing, `1` for a periodic timing,
- `count`: is the period in nanoseconds you want to use on the corresponding timer, not used for oneshot timers. It is in nanoseconds to ease its programming when different values are used by each timer, so that you do not have to care converting it from the CPU on which you are calling this function.

The start of the timing should be reasonably synchronized. You should call this function with due care and only when you want to manage the related interrupts in your own handler. For using local APIC timers in pacing real time tasks use the usual **rt_start_timer**, which under the MUP scheduler sets the same timer policy on all the local APIC timers, or **start_rt_apic_timers** that allows you to use `struct apic_timer_setup_data` directly.

NAME

rt_mount_rtai
rt_umount_rtai

FUNCTION

Initialize/uninitialize real time application interface

SYNOPSIS

```
#include "rtai.h"

void rt_mount_rtai (void);
void rt_umount_rtai (void);
```

DESCRIPTION

rt_mount_rtai initializes the real time application interface, i.e. grabs anything related to the hardware, data or service, pointed by at by the Real Time Hardware Abstraction Layer **RTHAL** (`struct rt_hal rthal;`).
rt_umount_rtai unmounts the real time application interface resetting Linux to its normal state.

NOTES

When you do "`insmod rtai`" **RTAI** is not active yet, it needs to be specifically switched on by calling **rt_mount_rtai**.

Functions provided by rtai_shm module

Following are some function calls that allows sharing memory inter-intra real time tasks and Linux processes. In fact it can be an alternative to SYSTEM V shared memory, the services are symmetrical, i.e. similar calls can be used both in real time tasks, i.e. within the kernel, and Linux processes. The function calls for Linux processes are inlined in the file "rtai_shm.h". This approach has been preferred to a library since: is simpler, more effective ,the calls are short, simple and just a few per process.

RTAI_SHM service functions:

- rtai_malloc_adr
- rtai_malloc
- rtai_kmalloc
- rtai_free
- rtai_kfree
- nam2num
- num2nam

NAME

rtai_malloc_adr
rtai_malloc
rtai_kmalloc

FUNCTION

Allocate a chunk of memory to be shared inter-intra kernel modules and Linux processes

SYNOPSIS

```
#include "rtai_shm.h"

void *rtai_malloc_adr (void *adr, unsigned long name, int size);
void *rtai_malloc (unsigned long name, int size);
void *rtai_kmalloc (unsigned long name, int size);
```

DESCRIPTION

rtai_malloc_adr and **rtai_malloc** are used to allocate in user space while **rtai_kmalloc** is used to allocate is kernel space.
adr is a user desired address where the allocated memory should be mapped in user space.
name is an unsigned long identifier and ***size*** is the amount of required shared memory. Since ***name*** can be a clumsy identifier, services are provided to convert 6 characters identifiers to unsigned long, and vice versa. See the functions **nam2num** and **num2nam**..
It must be remarked that the first allocation does a real allocation, any subsequent call to allocate with the same name from Linux processes just maps the area to the user space, or return the related pointer to the already allocated space in kernel space.
The functions return a pointer to the allocated memory, appropriately mapped to the memory space in use.

RETURN VALUE

On success a valid address is returned.
On failure a `0` is returned.

NOTES

If the same process calls **rtai_malloc_adr** and **rtai_malloc** twice in the same process it get a zero return value on the second call.

NAME

rtai_free
rtai_kfree

FUNCTION

Free a chunk of shared memory being shared inter-intra kernel modules and Linux processes

SYNOPSIS

```
#include "rtai_shm.h"

void rtai_free (unsigned long name, void *adr);
void rtai_kfree(void *adr);
```

DESCRIPTION

rtai_free is used to free from the user space a previously allocated shared memory while **rtai_kfree** is used for doing the same operation in kernel space.
name is the unsigned long identifier used when the memory was allocated and ***adr*** is the related address.
Analogously to what done by the allocation functions the freeing calls have just the effect of unmapping any shared memory being freed till the last is done, as that is the one the really frees any allocated memory.

NAME

nam2num
num2nam

FUNCTION

Convert a 6 characters string to un unsigned long, and vice versa, to be used as an identifier for **RTAI** services symmetrically available in user and kernel space, e.g. shared memory and **LXRT** and **LXRT-INFORMED** .

SYNOPSIS

```
#include "rtai_shm.h"

unsigned long nam2num (const char *name);
void num2nam(unsigned long id, const char* name);
```

DESCRIPTION

nam2num converts a 6 characters string ***name*** containing the alpha numeric identifier to its corresponding unsigned long identifier. **num2nam** does the opposite.
id is the unsigned long identifier whose alphanumeric name string has to be evaluated.
Allowed characters are:

- English letters (no difference between upper and lower case);
- decimal digits;
- underscore (_) and another character of your choice. The latter will be always converted back to a $ by **num2nam**.

LXRT services (soft-hard real time in user space)

LXRT is a module that allows you to use all the services made available by **RTAI** and its schedulers in user space, both for soft and hard real time. At the moment it is a feature you'll find nowhere but with **RTAI**. For an explanation of how it works see the Appendix B0, containing Pierre Cloutier's **LXRT-INFORMED** FAQs, and Appendix B1 for an explanation of the implementation of hard real time in user space (contributed by: Pierre Cloutier, Paolo Mantegazza, Steve Papacharalambous).
LXRT-INFORMED should be the production version of **LXRT**, the latter being the development version. So it can happen that **LXRT-INFORMED** could be lagging slightly behind **LXRT**. If you need to hurry to the services not yet ported to **LXRT-INFORMED** do it without pain. Even if you are likely to miss some useful services found only in **LXRT-INFORMED,** we release only when a feature is relatively stable.
From what said above there should be no need for anything specific as all the functions you can use in user space have been already documented in this manual. There are however a few exceptions that need to be explained.
Note also that, as already done for the shared memory services in user space, the function calls for Linux processes are inlined in the file "rtai_lxrt.h". This approach has been preferred to a library since it is simpler, more effective, the calls are short and simple so that, even if it is likely that there can be more than just a few per process, they could never be charged of making codes too bigger. Also common to shared memory is the use of unsigned int to identify **LXRT** objects. If you want to use string identifiers the same support functions, i.e. **`nam2num`** and **`num2nam`**, can be used.

LXRT service functions:

- rt_task_init
- rt_sem_init
- rt_mbx_init
- rt_register
- rt_get_adr
- rt_get_name
- rt_drg_on_adr
- rt_drg_on_name
- rt_make_hard_real_time
- rt_make_soft_real_time
- rt_allow_nonroot_hrt

NAME

rt_task_init

FUNCTION

Create a new real time task in user space.

SYNOPSIS

```
#include "rtai_lxrt.h"

LX_TASK *rt_task_init(unsigned int name, int priority, int
        stack_size, int max_msg_size);
```

DESCRIPTION

rt_task_init provides a real time buddy, also called proxy, task to the Linux process that wants to access **RTAI** scheduler services. It needs no task function as none is used, but it does need to setup a task structure and initialize it appropriately as the provided services are carried out as if the Linux process has become an **RTAI** task. Because of that it requires less arguments and returns the pointer to the task that is to be used in related calls.

name is a unique identifier that is possibly used by easing referencing the buddy **RTAI** task, and thus its peer Linux process.

priority is the priority of the buddy's priority.

stack_size is just what is implied by such a name and refers to the stack size used by the buddy.

max_msg_size is a hint for the size of the most lengthy message than is likely to be exchanged

stack_size and ***max_msg*** can be zero, in which case the default internal values are used. The assignment of a different value should be required only if you want to use task signal functions. In such a case note that these signal functions are intended to catch asyncrounous events in kernel space and, as such, must be programmed into a companion module and interfaced to their parent Linux process through the available services.

Keep an eye on the default stack (512) and message (256) sizes as they seem to be acceptable, but this API has not been used extensively with complex interrupt service routines. Since the latter are served on the stack of any task being interrupted, and more than one can pile up on the same stack, it can be possible that a larger stack is required. In such a case either recompile `lxrt.c` with macros STACK_SIZE and MSG_SIZE set appropriately, or explicitly assign larger values at your buddy tasks inits. Note that while the stack size can be critical the message size will not. In fact the module reassigns it, appropriately sized, whenever it is needed. The cost is a kmalloc with GFP_KERNEL that can block, but within the Linux environment. Note also that ***max_msg_size*** is for a buffer to be used to copy whatever message, either mailbox or inter task, from user to kernel space, as messages are not necessarily copied immediately, and has nothing to do directly with what you are doing.

It is important to remark that the returned task pointers cannot be used directly, they are for kernel space data, but just passed as arguments when needed.

RETURN VALUE

On success a pointer to the task structure initialized in kernel space is returned.

On failure a `0` value is returned as described below.

ERRORS

`0` It was not possible to setup the buddy task or something using the same ***name*** was found.

NAME

rt_sem_init

FUNCTION

Initialize a counting semaphore

SYNOPSIS

```
#include "rtai_sched.h"

SEM *rt_sem_init (unsigned long name, int initial_count);
```

DESCRIPTION

rt_sem_init allocates and initializes a semaphore to be referred by ***name***..
initial_count is the initial value of the semaphore
It is important to remark that the returned task pointer cannot be used directly, they are for kernel space data, but just passed as arguments when needed.

RETURN VALUE

pointer to the semaphore to be used in related calls or `0` if an error has occured.

ERRORS

`0` It was not possible to setup the semaphore or something using the same ***name*** was found.

NAME

rt_mbx_init

FUNCTION

Initialize mailbox

SYNOPSIS

```
#include "rtai_sched.h"

int rt_mbx_init (unsigned long name, int size);
```

DESCRIPTION

rt_mbx_init initializes a mailbox referred to by ***name*** of size ***size***.
It is important to remark that the returned task pointer cannot be used directly, they are for kernel space data, but just passed as arguments when needed.

RETURN VALUE

On success a pointer to the mail box to be used in related calls.
On failure a `0` value is returned as explained below.

ERRORS

`0` It was not possible to setup the semaphore or something using the same ***name*** was found.

NAME

rt_register
rt_get_adr
rt_get_name
rt_drg_on_adr
rt_drg_on_name

FUNCTION

Get properties, register and deregister objects, **RTAI** and **LXRT** names and related addresses.

SYNOPSIS

```
#include "rtai_shm.h"

int rt_register (unsigned long name, void *adr);
void *rt_get_adr (unsigned long name);
unsigned long rt_get_name (void *adr);
int rt_drg_on_adr (void *adr);
int rt_drg_on_name (unsigned long name);
```

DESCRIPTION

rt_register registers the object to be identified with ***name***, which is pointed by ***adr***.
rt_get_adr returns the address associated to ***name***.
rt_get_name returns the name pointed by the address ***adr***.
rt_drg_on_adr deregisters the object identified by its ***adr***.
rt_drg_on_name deregisters the object identified by its ***name***.

RETURN VALUES

rt_register returns a positive number on success, `0` on failure.
rt_get_adr returns the address associated to ***name*** on success, `0` on failure
rt_get_name returns the identifier pointed by the address ***adr*** on success, `0` on failure
rt_drg_on_adr returns a positive number on success, `0` on failure.
rt_drg_on_name returns a positive number on success, `0` on failure.

NOTES

The above functions can be used also for synchronizing on the existence or not of any implied object.

NAME

rt_make_hard_real_time
rt_make_soft_real_time

FUNCTION

Give a Linux process, or pthread, hard real time execution capabilities allowing full kernel preemption, or return it to the standard Linux behavior.

SYNOPSIS

```
#include "rtai_shm.h"

void rt_make_hard_real_time (void);
void rt_make_soft_real_time (void);
```

DESCRIPTION

rt_make_hard_real_time makes the soft Linux POSIX real time process, from which it is called, a hard real time **LXRT** process. It is important to remark that this function must be used only with soft Linux POSIX processes having their memory locked in memory. See Linux man pages.
rt_make_soft_real_time returns to soft Linux POSIX real time a process, from which it is called, that was made hard real time by a call to **rt_make_hard_real_time**.
Only the process itself can use these functions, it is not possible to impose the related transition from another process.
Note that processes made hard real time should avoid making any Linux System call that can lead to a task switch as Linux cannot run anymore processes that are made hard real time. To interact with Linux you should couple the process that was made hard real time with a Linux buddy server, either standard or POSIX soft real time. To communicate and synchronize with the buddy you can use the wealth of available **RTAI**, and its schedulers, services. After all it is pure nonsense to use a non hard real time Operating System, i.e. Linux, from within hard real time processes.

NAME

rt_allow_nonroot_hrt

FUNCTION

To allow a non root user to use the Linux POSIX soft real time process management and memory lock functions, and to allow it to do any input-output operation from user space.

SYNOPSIS

```
#include "rtai_shm.h"

void rt_allow_nonroot_hrt (void);
```

DESCRIPTION

Nothing to be added to the function description, except that it is not possible to impose the related transition from another process.

MINI_RTAI_LXRT module

The **MINI_RTAI_LXRT** tasklets module adds an interesting new feature along the line, pioneered by **RTAI**, of a symmetric usage of all its services inter-intra kernel and user space, both for soft and hard real time applications. In such a way you have opened a whole spectrum of development and implementation lanes, allowing maximum flexibility with uncompromized performances.

The new services provided can be useful when you have many tasks, both in kernel and user space, that must execute in soft/hard real time but do not need any **RTAI** scheduler service that could lead to a task block. Such tasks are here called tasklets and can be of two kinds: normal tasklets and timed tasklets (timers).

It must be noted that only timers should need to be made available both in user and kernel space. In fact normal tasklets in kernel space are nothing but standard functions that can be directly executed by calling them, so there would be no need for any special treatment. However to maintain full usage symmetry, and to ease any possible porting from one address space to the other, also normal tasklet functions can be used in whatever address space.

Note that if, at this point, you are reminded to similar Linux kernel services you are not totally wrong. They are not exactly the same, because of their symmetric availability in kernel and user space, but the basic idea behind them is clearly fairly similar.

Tasklets should be used whenever the standard hard real time tasks available with **RTAI** and **LXRT** schedulers can be a waist of resources and the execution of simple, possibly timed, functions could often be more than enough.

Instances of such applications are timed polling and simple Programmable Logic Controllers (PLC) like sequences of services. Obviously there are many others instances that can make it sufficient the use of tasklets, either normal or timers. In general such an approach can be a very useful complement to fully featured tasks in controlling complex machines and systems, both for basic and support services.

It is remarked that the implementation found here for timed tasklets rely on a server support task that executes the related timer functions, either in oneshot or periodic mode, on the base of their time deadline and according to their, user assigned, priority.

Instead, as told above, plain tasklets are just functions executed from kernel space; their execution needs no server and is simply triggered by calling a given service function at due time, either from a kernel task or interrupt handler requiring, or in charge of, their execution when they are needed.

Once more it is important to recall that all non blocking **RTAI** scheduler services can be used in any tasklet function. Blocking services must absolutely be avoided. They will deadlock the timers server task, executing task or interrupt handler, whichever applies, so that no more tasklet functions will be executed.

User and kernel space **MINI_RTAI_LXRT** applications can cooperate and synchronize by using shared memory.

It has been called **MINI_RTAI_LXRT** because it is a kind of light soft/hard real time server that can partially substitute **RTAI** and **LXRT** in simple applications, i.e. if the constraints hinted above are wholly satisfied. So **MINI_RTAI_LXRT** can be used in kernel and user space, with any **RTAI** scheduler.

Its implementations has been very easy, as it is nothing but what its name implies. **LXRT** made all the needed tools already available. In fact it duplicates a lot of **LXRT** so that its final production version will be fully integrated with it, ASAP. However, at the moment, it cannot work with **LXRT** yet.

Note that in user space you run within the memory of the process owning the tasklet function so you **MUST** lock all of your processes memory in core, by using **`mlockall`**, to prevent it being swapped out. Also abundantly pre grow your stack to the largest size needed during the execution of your application, see **`mlockall`** usage in Linux manuals.

The **RTAI** distribution contains many useful examples that demonstrate the use of most services, both in kernel and user space.

MINI_RTAI_LXRT service functions:

- rt_tasklet_init
- rt_tasklet_delete
- rt_insert_tasklet
- rt_remove_tasklet
- rt_find_tasklet_by_id
- rt_tasklet_exec
- rt_timer_init
- rt_timer_delete
- rt_insert_timer
- rt_remove_timer
- rt_set_timer_priority
- rt_set_timer_firing_time
- rt_set_timer_period
- rt_set_timer_handler
- rt_set_timer_data
- rt_tasklets_use_fpu

NAME

rt_tasklet_init
rt_tasklet_delete

FUNCTION

Init/delete, in kernel space, a tasklet structure to be used in user space.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

struct rt_tasklet_struct* rt_tasklet_init (void);
void rt_tasklet_delete (struct rt_tasklet_struct* tasklet);
```

DESCRIPTION

rt_tasklet_init allocate a tasklet structure (struct rt_tasklet_struct) in kernel space to be used for the management of a user space tasklet.
rt_tasklet_delete free a tasklet structure (struct rt_tasklet_struct) in kernel space that was allocated by **rt_tasklet_init** .
tasklet is the pointer to the tasklet structure (struct rt_tasklet_struct) returned by **rt_tasklet_init**.
As said above this functions are to be used only for user space tasklets. In kernel space they are just empty macros, as the user can, and must allocate the related structure directly, either statically or dynamically.

RETURN VALUE

rt_tasklet_init returns the pointer to the tasklet structure the user space application must use to access all its related services.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_insert_tasklet
rt_remove_tasklet

FUNCTION

Insert/remove a tasklet in the list of tasklets to be processed.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

int rt_insert_tasklet (struct rt_tasklet_struct* tasklet, void
      (*handler)(unsigned long), unsigned long data, unsigned long
      id, int pid)
void rt_remove_tasklet (struct rt_tasklet_struct* tasklet);
```

DESCRIPTION

rt_insert_tasklet insert a tasklet in the list of tasklets to be processed.
rt_remove_tasklet remove a tasklet from the list of tasklets to be processed.
tasklet is the pointer to the tasklet structure to be used to manage the tasklet at hand.
handler is the tasklet function to be executed.
data is an unsigned long to be passed to the handler. Clearly by an appropriate type casting one can pass a pointer to whatever data structure and type is needed.
id is a unique unsigned number to be used to identify the tasklet ***tasklet***. It is typically required by the kernel space service, interrupt handler ot task, in charge of executing a user space tasklet. The support functions **nam2num** and **num2nam** can be used for setting up ***id*** from a six character string.
pid is an integer that marks a tasklet either as being a kernel or user space one. Despite its name you need not to know the pid of the tasklet parent process in user space. Simple use 0 for kernel space and 1 for user space.

RETURN VALUE

`0` is returned on success, otherwise a negative number is returned as explained below.

ERRORS

EINVAL to indicate that an invalid ***handler*** address as been passed by **rt_insert_tasklet**, or an invalid ***tasklet*** is used in **rt_remove_tasklet**.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_find_tasklet_by_id
rt_tasklet_exec

FUNCTION

Find a tasklet identified by its id; exec a tasklet.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

struct rt_tasklet_struct* rt_find_tasklet_by_id (unsigned long id)
void rt_tasklet_exec (struct rt_tasklet_struct* tasklet);
```

DESCRIPTION

rt_find_tasklet_by_id insert a tasklet in the list of tasklets to be processed.
rt_tasklet_exec execute a tasklet from the list of tasklets to be processed.
id is the unique unsigned long to be used to identify the tasklet. The support functions **nam2num** and **num2nam** can be used for setting up ***id*** from a six character string.
tasklet is the pointer to the tasklet structure to be used to manage the tasklet ***tasklet***.
Kernel space tasklets addresses are usually available directly and can be easily be used in calling **rt_tasklet_exec**. In fact one can call the related handler directly without using such a support function, which is mainly supplied for symmetry and to ease the porting of applications from one space to the other,
User space tasklets instead must be first found within the tasklet list by calling **rt_find_tasklet_by_id** to get the tasklet address to be used in **rt_tasklet_exec**.

RETURN VALUE

On success **rt_find_tasklet_by_id** returns the pointer to tasklet handler, 0 is returned on failure.

ERRORS

0 to indicate that ***id*** is not a valid identifier so that the related tasklet was not found.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_timer_init
rt_timer_delete

FUNCTION

Init/delete, in kernel space, a timed tasklet, simply called timer, structure to be used in user space.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

struct rt_tasklet_struct* rt_timer_init (void);
void rt_timer_delete (struct rt_tasklet_struct* timer);
```

DESCRIPTION

rt_timer_init allocate a timer tasklet structure (`struct rt_tasklet_struct`) in kernel space to be used for the management of a user space timer.
rt_timer_delete free a timer tasklet structure (`struct rt_tasklet_struct`) in kernel space that was allocated by **rt_timer_init** .
timer is a pointer to a timer tasklet structure (`struct rt_tasklet_struct`).
As said above this functions are to be used only for user space timers. In kernel space they are just empty macros, as the user can, and must allocate the related structure directly, either statically or dynamically.

RETURN VALUE

rt_timer_init returns the pointer to the timer structure the user space application must use to access all its related services.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_insert_timer
rt_remove_timer

FUNCTION

Insert/remove a timer in the list of timers to be processed.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

int rt_insert_timer (struct rt_tasklet_struct* timer, int priority,
        RTIME firing_time, RTIME period, void (*handler)(unsigned
        long), unsigned long data, int pid)
void rt_remove_timer (struct rt_tasklet_struct* timer);
```

DESCRIPTION

rt_insert_timer insert a timer in the list of timers to be processed. Timers can be either periodic or oneshot. A periodic timer is reloaded at each expiration so that it executes with the assigned periodicity. A oneshot timer is fired just once and then removed from the timers list. Timers can be reinserted or modified within their handlers functions.
rt_remove_timer remove a timer from the list of the timers to be processed.
timer is the pointer to the timer structure to be used to manage the timer at hand.
priority is the priority to be used to execute timers handlers when more than one timer has to be fired at the same time.
It can be assigned any value such that: `0 <` ***priority*** `<` RT_LOWEST_PRIORITY.
firing_time is the time of the first timer expiration.
period is the period of a periodic timer. A periodic timer keeps calling its handler at "***firing_time*** `+ k*`***period*** " `k = 0, 1 …`
To define a oneshot timer simply use a null ***period***.
handler is the timer function to be executed at each timer expiration.
data is an unsigned long to be passed to the handler. Clearly by a appropriate type casting one can pass a pointer to whatever data structure and type is needed.
pid is an integer that marks a timer either as being a kernel or user space one. Despite its name you need not to know the pid of the timer parent process in user space. Simple use `0` for kernel space and 1 for user space.

RETURN VALUE

`0` is returned on success, otherwise a negative number is returned as explained below.

ERRORS

`EINVAL` to indicate that an invalid ***handler*** address as been passed by **rt_insert_timer.**

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_set_timer_priority

FUNCTION

Change the priority of an existing timer.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

void rt_set_timer_priority (struct rt_tasklet_struct* timer, int
        priority);
```

DESCRIPTION

rt_set_timer_priority change the priority of an existing timer.
timer is the pointer to the timer structure to be used to manage the timer at hand.
priority is the priority to be used to execute timers handlers when more than one timer has to be fired at the same time.
It can be assigned any value such that: 0 < ***priority*** < RT_LOWEST_PRIORITY.
This function can be used within the timer handler.

RETURN VALUE

None.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_set_timer_firing_time
rt_set_timer_period

FUNCTION

Change the firing time and period of a timer.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

int rt_set_timer_firing_time (struct rt_tasklet_struct* timer,
        RTIME firing_time)
int rt_set_timer_period (struct rt_tasklet_struct* timer, RTIME
        period)
#define rt_fast_set_timer_period (timer, period)
```

DESCRIPTION

rt_set_timer_firing_time changes the firing time of a periodic timer overloading any existing value, so that the timer next shoot will take place at the new firing time. Note that if a oneshot timer has its firing time changed after it has already expired this function has no effect. You should reinsert it in the timer list with the new firing time.
rt_set_timer_period changes the period of a periodic timer. Note that the new period will be used to pace the timer only after the expiration of the firing time already in place. Using this function with a period different from zero for a oneshot timer, that has not expired yet, will transform it into a periodic timer.
timer is the pointer to the timer structure to be used to manage the timer at hand.
firing_time is the new time of the first timer expiration.
period is the new period of a periodic timer.
The macro **rt_fast_set_timer_period** can substitute the corresponding function in kernel space if both the existing timer period and the new one fit into an 32 bits integer.
These functions and macro can be used within the timer handler.

RETURN VALUE

`0` is returned on success.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_set_timer_handler

FUNCTION

Change the timer handler.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

int rt_set_timer_handler (struct rt_tasklet_struct* timer, void
        (*handler)(unsigned long))
#define rt_fast_set_timer_handler (timer, handler)
```

DESCRIPTION

rt_set_timer_handler changes the timer handler function overloading any existing value, so that at the next timer firing the new handler will be used. Note that if a oneshot timer has its handler changed after it has already expired this function has no effect. You should reinsert it in the timer list with the new handler.
timer is the pointer to the timer structure to be used to manage the timer at hand.
handler is the new handler.
The macro **rt_fast_set_timer_handler** can safely be used to substitute the corresponding function in kernel space.
This function and macro can be used within the timer handler.

RETURN VALUE

0 is returned on success.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_set_timer_data

FUNCTION

Change the data passed to a timer.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

int rt_set_timer_data (struct rt_tasklet_struct* timer, unsigned
          long data)
#define rt_fast_set_timer_data (timer, data)
```

DESCRIPTION

rt_set_timer_data changes the timer data overloading any existing value, so that at the next timer firing the new data will be used. Note that if a oneshot timer has its data changed after it is already expired this function has no effect. You should reinsert it in the timer list with the new data.
timer is the pointer to the timer structure to be used to manage the timer at hand.
data is the new data.
The macro **rt_fast_set_timer_data** can safely be used substitute the corresponding function in kernel space.
This function and macro can be used within the timer handler.

RETURN VALUE

`0` is returned on success.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

NAME

rt_tasklets_use_fpu

FUNCTION

Notify the use of floating point operations within any tasklet/timer.

SYNOPSIS

```
#include "mini_rtai_lxrt.h"

void rt_tasklets_use_fpu (int use_fpu_flag)
```

DESCRIPTION

rt_tasklets_use_fpu notifies that there is at least one tasklet/timer using floating point calculations within its handler function.

use_fpu_flag set/resets the use of floating point calculations. A value different from `0` sets the use of floating point calculations. A `0` value resets the no floating calculations state.

Note that the use of floating calculations is assigned once for all and is valid for all tasklets/timers. If just one handler needs it all of them will have floating point support. An optimized floating point support, i.e. on a per tasklet/timer base will add an unnoticeable performance improvement on most CPUs. However such an optimization is not rule out a priori, if anybody can prove it is really important.

This function and macro can be used within the timer handler.

RETURN VALUE

None.

ERRORS

None.

NOTES

To be used only with **RTAI** 24.x.xx**.**

Functions provided by rtai_fifos module

See Appendix C for a quick general overview of **RTAI** fifos.

RTAI FIFO communication functions:

RTAI fifos maintain full compatibility with those available in NMT_RTLinux while adding many other useful services that avoid the clumsiness of Unix/Linux calls. So if you need portability you should bent yourself to the use of select for timing out IO operations, while if you have not to satisfy such constraints use the available simpler, and more direct, **RTAI** fifos specific services.
In the table below the standard Unix/Linux services in user space are enclosed in []. See standard Linux man pages if you want to use them, they need not be explained here.

Called from RT task	**Called from Linux process**
rtf_create	rtf_open_sized
	[open]
rtf_destroy	[close]
rtf_reset	rtf_reset
rtf_resize	rtf_resize
rtf_put	[write]
	rtf_write_timed
rtf_get	[read]
	rtf_read_timed
	rtf_read_all_at_once
rtf_create_handler	
	rtf_suspend_timed
	rtf_set_async_sig

In Linux fifos have to be created by `mknod /dev/rtf<x> c 150 <x>` where `<x>` is the minor device number, from 0 to 63; thus on the Linux side RTL fifos can be used as standard character devices. As it was said above to use standard IO operations on such devices there is no need to explain anything, go directly to Linux man pages. **RTAI** fifos specific services available in kernel and user space are instead explained here.
What is important to remember is that in the user space side you address fifos through the file descriptor you get at fifo device opening while in kernel space you directly address them by their minor number. So you will mate the *fd* you get in user space by using *open(/dev/rtfxx,…)* to the integer *xx* you'll use in kernel space.

IMPORTANT NOTE:

RTAI fifos should be used just with applications that use only real time interrupt handlers, so that no **RTAI** scheduler is installed, or if you need compatibility with NMT RTL. If you are working with any **RTAI** scheduler already installed you are strongly invited to think about avoiding them, use **LXRT** instead.
It is far better and flexible, and if you really like it the fifos way mailboxes are a one to one, more effective, substitute. After all **RTAI** fifos are implemented on top of them.

NAME

rtf_create
rtf_open_sized

FUNCTION

Create a real-time FIFO

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_create (unsigned int fifo, int size);
int rtf_open_sized (const char *device, int permission, int size);
```

DESCRIPTION

rtf_create creates a real-time fifo (RT-FIFO) of initial ***size*** and assigns it the identifier ***fifo***. It must be used only in kernel space.
fifo is a positive integer that identifies the fifo on further operations. It has to be less than RTF_NO.
fifo may refer to an existing RT-FIFO. In this case the size is adjusted if necessary.
The RT-FIFO is a character based mechanism to communicate among real-time tasks and ordinary Linux processes. The **rtf_*** functions are used by the real-time tasks; Linux processes use standard character device access functions such as read, write, and select.
rtf_open_sized is the equivalent of **rtf_create** in user space. If any of these functions finds an existing fifo of lower size it resizes it to the larger new size. Note that the same condition apply to the standard Linux device open, except that when it does not find any already existing fifo it creates it with a default size of 1K bytes.
device and ***permission*** are the standard Linux open parameters.
It must be remarked that practically any fifo size can be asked for. In fact if ***size*** is within the constraint allowed by kmalloc such a function is used, otherwise vmalloc is called, thus allowing any size that can fit into the available core memory.
Multiple calls of the above functions are allowed, a counter is kept internally to track their number, and avoid destroying/closing a fifo that is still used.

RETURN VALUE

On success **rtf_create** returns ***size***, instead **rtf_open_sized** return the usual Unix file descriptor to be use in standard reads and writes.
On failure, a negative value is returned as described below.

ERRORS

ENODEV ***fifo*** is greater than or equal to RTF_NO.
ENOMEM ***size*** bytes could not be allocated for the RT-FIFO.

NOTES

In user space the standard UNIX **open** acts like **rtf_open_sized** with a default 1K ***size.***

NAME

rtf_destroy

FUNCTION

Close a real-time FIFO

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_destroy (unsigned int fifo);
```

DESCRIPTION

rtf_destroy closes, in kernel space, a real-time fifo previously created/reopened with **rtf_create** or **rtf_open_sized**. An internal mechanism counts how many times a fifo was opened. Opens and closes must be in pair. **rtf_destroy** should be called as many times as **rtf_create** was. After the last close the fifo is really destroyed.
No need for any particular function for the same service in user space, simply use the standard Unix close.

RETURN VALUE

On success, a non-negative number is returned. Actually it is the open counter, that means how many times **rtf_destroy** should be called yet to destroy the fifo.
On failure, a negative value is returned as described below.

ERRORS

ENODEV ***fifo*** is greater than or equal to RTF_NO.
EINVAL ***fifo*** refers to a not opened fifo.

NOTES

The equivalent of **rtf_destroy** in user space is the standard UNIX **close**.

NAME

rtf_reset

FUNCTION

Reset a real-time FIFO

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_reset (unsigned int fd_fifo);
```

DESCRIPTION

rtf_reset resets RT-FIFO ***fd_fifo*** by setting its buffer pointers to zero, so that any existing data is discarded and the fifo started anew like at its creations. It can be used both in kernel and user space.

fd_fifo is a file descriptor returned by standard UNIX open in user space while it is directly the chosen fifo number in kernel space.

RETURN VALUE

On success, 0 is returned.
On failure, a negative value is returned.

ERRORS

ENODEV	***fd_fifo*** is greater than or equal to RTF_NO.
EINVAL	***fd_fifo*** refers to a not opened fifo.
EFAULT	Operation was unsuccessful.

NAME

rtf_resize

FUNCTION

Resize a real-time FIFO

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_resize (unsigned int fifo, int size);
```

DESCRIPTION

rtf_resize modifies the real-time fifo ***fifo***, previously created with , **rtf_create**, to have a new size of ***size***. Any data in the fifo is discarded.
fd_fifo is a file descriptor returned by standard UNIX open in user space while it is directly the chosen fifo number in kernel space.

RETURN VALUE

On success, ***size*** is returned.
On failure, a negative value is returned.

ERRORS

ENODEV	***fifo*** is greater than or equal to RTF_NO.
EINVAL	***fifo*** refers to a not opened fifo.
ENOMEM	***size*** bytes could not be allocated for the RT-FIFO. Fifo size is unchanged.

NAME

rtf_put

FUNCTION

Write data to FIFO

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_put (unsigned int fifo, void *buf, int count);
```

DESCRIPTION

rtf_put tries to write a block of data to a real-time fifo previously created with **rtf_create**.
fifo is the ID with which the RT-FIFO was created.
buf points the block of data to be written.
count is the size of the block in bytes.
This mechanism is available only in kernel space, i.e. either in real-time tasks or handlers; Linux processes use a write to the corresponding /dev/fifo<*n*> device to enqueue data to a fifo. Similarly, Linux processes use read or similar functions to read the data previously written via **rtf_put** by a real-time task.

RETURN VALUE

On success, the number of bytes written is returned. Note that this value may be less than ***count*** if ***count*** bytes of free space is not available in the fifo.
On failure, a negative value is returned.

ERRORS

ENODEV	***fifo*** is greater than or equal to RTF_NO.
EINVAL	***fifo*** refers to a not opened fifo.

NOTES

The equivalent of **rtf_put** in user space is the standard UNIX **write**, which can be either blocking or nonblocking according to how you opened the related device.

NAME

rtf_write_timed

FUNCTION

Write data to FIFO in user space, with timeout

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_write_timed (int fd, char *buf, int count, int delay);
```

DESCRIPTION

rtf_write_timed writes a block of data to a real-time fifo identified by the file descriptor ***fd*** waiting at most ***delay*** milliseconds to complete the operation.
fd is the file descriptor returned at fifo open.
buf points the block of data to be written.
count is the size of the block in bytes.
delay is the timeout time in milliseconds.

RETURN VALUE

On success or timeout, the number of bytes written is returned. Note that this value may be less than ***count*** if ***count*** bytes of free space is not available in the fifo or a timeout occured.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd*** refers to a not opened fifo.

NOTES

The standard, clumsy, Unix way to achieve the same result is to use **select**.

NAME

rtf_get

FUNCTION

Read data from FIFO

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_get (unsigned int fifo, void *buf, int count);
```

DESCRIPTION

rtf_get tries to read a block of data from a real-time fifo previously created with a call to **rtf_create**.
fifo is the ID with which the RT-FIFO was created.
buf points a buffer of ***count*** bytes size provided by the caller. This mechanism is available only to real-time tasks; Linux processes use a read from the corresponding fifo device to dequeue data from a fifo. Similarly, Linux processes use write or similar functions to write the data to be read via **rtf_put** by a real-time task.
rtf_get is often used in conjunction with **rtf_create_handler** to process data received asynchronously from a Linux process. A handler is installed via **rtf_create_handler**; this handler calls **rtf_get** to receive any data present in the RT-FIFO as it becomes available. In this way, polling is not necessary; the handler is called only when data is present in the fifo.

RETURN VALUE

On success, the size of the received data block is returned. Note that this value may be less than ***count*** if ***count*** bytes of data is not available in the fifo.
On failure, a negative value is returned.

ERRORS

ENODEV ***fifo*** is greater than or equal to RTF_NO.
EINVAL ***fifo*** refers to a not opened fifo.

NOTES

The equivalent of **rtf_get** in user space is the standard UNIX **read**, which can be either blocking or nonblocking according to how you opened the related device.

NAME

rtf_read_timed

FUNCTION

Read data from FIFO in user space, with timeout

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_read_timed (int fd, char *buf, int count, int delay);
```

DESCRIPTION

rtf_read_timed reads a block of data from a real-time fifo identified by the file descriptor ***fd*** waiting at most ***delay*** milliseconds to complete the operation.
fd is the file descriptor returned at fifo open.
buf points the block of data to be written.
count is the size of the block in bytes.
delay is the timeout time in milliseconds.

RETURN VALUE

On success or timeout, the number of bytes read is returned. Note that this value may be less than ***count*** if ***count*** bytes of free space is not available in the fifo or a timeout occured.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd*** refers to a not opened fifo.

NOTES

The standard, clumsy, Unix way to achieve the same result is to use **select**.

NAME

rtf_read_all_at_once

FUNCTION

Read data from FIFO in user space, waiting for all of them

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_read_all_at_once (int fd, char *buf, int count);
```

DESCRIPTION

rtf_read_all_at_once reads a block of data from a real-time fifo identified by the file descriptor ***fd*** blocking till all waiting at most ***dcount*** bytes are available, whichever option was used at the related device opening.
fd is the file descriptor returned at fifo open.
buf points the block of data to be written.

RETURN VALUE

On success, the number of bytes read is returned.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd*** refers to a not opened fifo.

NAME

rtf_create_handler

FUNCTION

Install a FIFO handler function

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_create_handler (unsigned int fifo, int (*handler)(unsigned
       int fifo));
int rtf_create_handler (unsigned int fifo, X_FIFO_HANDLER(handler
       ));
```

DESCRIPTION

rtf_create_handler installs a handler which is executed when data is written to or read from a real-time fifo.
fifo is an RT-FIFO that must have previously been created with a call to **rtf_create**.
The function pointed by ***handler*** is called whenever a Linux process accesses that fifo.
rtf_create_handler is often used in conjunction with **rtf_get** to process data acquired asynchronously from a Linux process. The installed handler calls **rtf_get** when data is present. Because the handler is only executed when there is activity on the fifo, polling is not necessary.
The form with X_FIFO_HANDLER(handler) allows to install an extended handler, i.e. one prototyped as:

```
int (*handler)(unsigned int fifo, int rw);
```

to allow the user to easily understand if the handler was called at fifo read, ***rw*** = *'r'*, or write, , ***rw*** = *'w'*.

RETURN VALUE

On success, `0` is returned.
On failure, a negative value is returned.

ERRORS

`EINVAL` ***fifo*** is greater than or equal to `RTF_NO`, or ***handler*** is `NULL`.

NOTES

rtf_create_handler does not check if FIFO referred by ***fifo*** is open or not. The next call of **rtf_create** will uninstall the handler just "installed".

NAME

rtf_suspend_timed

FUNCTION

Suspend a process for some time

SYNOPSIS

```
#include "rtai_fifos.h"

void rtf_suspend_timed (int fd, int delay);
```

DESCRIPTION

rtf_suspend_timed suspends a Linux process according to ***delay***.
fd is the file descriptor returned at fifo open, **rtf_suspend_timed** needs a fifo support.
delay is the timeout time in milliseconds.

NOTES

The standard, clumsy, way to achieve the same result is to use **select** with null file arguments, for long sleeps, with seconds resolution, **sleep** is also available.

NAME

rtf_set_async_sig

FUNCTION

Activate asynchronous notification of data availability

SYNOPSIS

```
#include "rtai_fifos.h"

void rtf_set_async_sig(int fd, int signum);
```

ERRORS

EINVAL ***fd*** refers to a not opened fifo.

DESCRIPTION

rtf_set_async_sig activate an asynchronous signals to notify data availability by catching a user set signal ***signum***.
signum is a user chosen signal number to be used, default is SIGIO.

RTAI FIFO semaphore functions:

Fifos have an embedded synchronization capability, however using them only for such a purpose can be clumsy. So **RTAI** fifos have binary semaphores for that purpose. Note that, as for put and get fifos functions, only nonblocking functions are available in kernel space.

Called from RT task	**Called from Linux process**
rtf_sem_init	rtf_sem_init
rtf_sem_post	rtf_sem_post
	rtf_sem_wait
rtf_sem_trywait	rtf_sem_trywait
	rtf_sem_timed_wait
rtf_sem_destroy	rtf_sem_destroy

To add a bit of confusion (☺), with respect to RTAI schedulers semaphore functions, fifos semaphore functions names follow the POSIX mnemonics.
It should be noted that semaphores are associated to a fifo for identification purposes. So it is once more important to remember is that in the user space side you address fifos through the file descriptor you get at fifo device opening while in kernel space you directly address them by their minor number. So you will mate the *fd* you get in user space by *open(/dev/rtfxx,…)* to the integer *xx* you'll use in kernel space.

NAME

rtf_sem_init

FUNCTION

Initialize a binary semaphore

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_sem_init (int fd_fifo, int value);
```

DESCRIPTION

rtf_sem_init initializes a semaphore identified by the file descriptor or fifo number ***fd_fifo***. A fifo semaphore can be used for communication and synchronization between kernel and user space.
fd_fifo is a file descriptor returned by standard UNIX open in user space while it is directly the chosen fifo number in kernel space. In fact fifos semaphores must be associated to a fifo for identification purposes.
value is the initial value of the semaphore, it must be either 0 or 1.
rt_sem_init can be used both in kernel and user space.

RETURN VALUE

On success, 0 is returned.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd_fifo*** refers to an invalid file descriptor or fifo.

NAME

rtf_sem_destroy

FUNCTION

Delete a semaphore

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_sem_destroy (int fd_fifo);
```

DESCRIPTION

rtf_sem_destroy deletes a semaphore previously created with **rtf_sem_init**.
fd_fifo is a file descriptor returned by standard UNIX open in user space while it is directly the chosen fifo number in kernel space. In fact fifos semaphores must be associated to a fifo for identification purposes.
Any tasks blocked on this semaphore is returned in error and allowed to run when semaphore is destroyed.
rtf_sem_destroy can be used both in kernel and user space.

RETURN VALUE

On success, 0 is returned.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd_fifo*** refers to an invalid file descriptor or fifo.

NAME

rtf_sem_post

FUNCTION

Posting (signaling) a semaphore

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_sem_signal (int fd_fifo);
```

DESCRIPTION

rtf_sem_post signal an event to a semaphore. The semaphore value is set to one and the first process, if any, in semaphore's waiting queue is allowed to run.
fd_fifo is a file descriptor returned by standard UNIX open in user space while it is directly the chosen fifo number in kernel space. In fact fifos semaphores must be associated to a fifo for identification purposes.
Since it is not blocking **rtf_sem_post** can be used both in kernel and user space.

RETURN VALUE

On success, 0 is returned.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd_fifo*** refers to an invalid file descriptor or fifo.

NAME

rtf_sem_wait

FUNCTION

Take a semaphore

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_sem_wait (int fd);
```

DESCRIPTION

rtf_sem_wait waits for a event to be posted (signaled) to a semaphore. The semaphore value is set to tested and set to zero. If it was one **rtf_sem_wait** returns immediately. Otherwise the caller process is blocked and queued up in a priority order based on is POSIX real time priority.
A process blocked on a semaphore returns when:

- The caller task is in the first place of the waiting queue and somebody issues a **rtf_sem_post**;
- An error occurs (e.g. the semaphore is destroyed);

fd is the file descriptor returned by standard UNIX open in user space
Since it is blocking **rtf_sem_wait** cannot be used both in kernel and user space.

RETURN VALUE

On success, 0 is returned.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd_fifo*** refers to an invalid file descriptor or fifo.

NAME

rtf_sem_trywait

FUNCTION

Take a semaphore, only if the calling task is not blocked

SYNOPSIS

```
#include "rtai_sched.h"

int rtf_sem_trywait (int fd_fifo);
```

DESCRIPTION

rtf_sem_trywait is a version of the semaphore wait operation is similar to **rtf_sem_wait** but it is never blocks the caller. If the semaphore is not free, **rtf_sem_trywait** returns immediately and the semaphore value remains unchanged.
fd_fifo is a file descriptor returned by standard UNIX open in user space while it is directly the chosen fifo number in kernel space. In fact fifos semaphores must be associated to a fifo for identification purposes.
Since it is not blocking **rtf_sem_trywait** can be used both in kernel and user space.

RETURN VALUE

On success, 0 is returned.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd_fifo*** refers to an invalid file descriptor or fifo.

NAME

rtf_sem_timed_wait

FUNCTION

Wait a semaphore with timeout

SYNOPSIS

```
#include "rtai_fifos.h"

int rtf_sem_timed_wait (int fd, int delay);
```

DESCRIPTION

rtf_sem_timed_wait are timed version of the standard semaphore wait call. The semaphore value is tested and set to zero. If it was one **rtf_sem_timed_wait** returns immediately. Otherwise the caller process is blocked and queued up in a priority order based on is POSIX real time priority. A process blocked on a semaphore returns when:

- The caller task is in the first place of the waiting queue and somebody issues a **rtf_sem_post**;
- Timeout occurs;
- An error occurs (e.g. the semaphore is destroyed);

fd is the file descriptor returned by standard UNIX open in user space.
In case of timeout the semaphore value is set to one before return.
delay is in milliseconds and is relative to the Linux current time.
Since it is blocking **rtf_sem_timed_wait** cannot be used both in kernel and user space.

RETURN VALUE

On success, 0 is returned.
On failure, a negative value is returned.

ERRORS

EINVAL ***fd_fifo*** refers to an invalid file descriptor or fifo.

APPENDIX A

An overview of RTAI schedulers.

RTAI has a UniProcessor (UP) specific scheduler and two for MultiProcessors (MP). In the latter case you can chose between a symmetricMultiProcessor (SMP) and a MultiUniProcessor (MUP) scheduler.

The UP scheduler can be timed only by the 8254 timer and cannot be used with MPs.

The SMP scheduler can be timed either by the 8254 or by a local APIC timer. In SMP/8254 tasks are defaulted to work on any CPU but you can assign them to any subset, or to a single CPU, by using the function "rt_set_runnable_on_cpus".
It is also possible to assign any real time interrupt service to a specific cpu by using "rt_assign_irq_to_cpu" and "rt_reset_irq_to_sym_mode".
Thus a user can statically optimize his/her application if he/she believes that it can be better done than using a symmetric load distribution.
The possibility of forcing any interrupts to a specific CPU is clearly not related to the SMP scheduler and can be used also with interrupt handlers alone.
Note that only the real time interrupt handling is forced to a specific CPU. That means that if you check this feature by using "cat /proc/interrupts" for a real time interrupt that is chained to Linux, e.g. the timer when rtai_sched is installed, you can still see some interrupts distributed to all the CPUs, even if they are mostly on the assigned one. That is because Linux interrupts are kept symmetric by the RTAI dispatcher of Linux irqs.
For the SMP/APIC based scheduler if you want to statically optimize the load distribution by binding tasks to specific CPUs it can be useful to use "rt_get_timer_cpu()" just after having installed the timer, to know which CPU is using its local APIC timer to pace the scheduler. Note that for the oneshot case that will be the main timing CPU but not the only one. In fact which local APIC is shot depends on the task scheduling out, as that will determine the next shooting.

SMP schedulers allow to chose between a periodic and a oneshot timer, not to be used together. The periodic ticking is less flexible but, with the usual PC hardware much more efficient. So it is up to you which one to choose in relation to the applications at hand.
It should be noted that in the oneshot mode the time is measured on the base of the CPU time stamp clock (TSC) and neither on the 8254 chip nor on the local APIC timer, which are used only to generate oneshot interrupts. The periodic mode is instead timed by either the 8254 or the local APIC timers.
If the 8254 is used slow I/Os to the ISA bus are limited as much as possible with a sizable gain in efficiency. The oneshot mode has just about 15-20% more overhead than the periodic one. The use of the local APIC timers leads to a further improvement and substantially less jitter.
Remember that local APICs are hard disabled on UPs, unless you are using just one CPU on an MP motherboard. Experience with local APIC timers shows that there is no performance improvement for a periodic scheduling, except for a marginal reduced jitter, while the oneshot case gain is the sizable 10-15% mentioned above.
In fact by using the TSC just two outb are required to reprogram the 8254, i.e. approximately 3 us, against almost nothing for the APIC timer.
However you have to broadcast a message to all the CPUs in any case, and that is at least about more than 3 us, the APIC bus is an open drain 2 wires one and is not lightning like.
Note that the performance loss of the 8254 is just a fraction of the overall task switching procedure, which is always substantially heavier in the oneshot case than in periodic mode.
No doubt however that if you have an SMP motherboard, or a local APIC enabled anyhow, you should use the APIC SMP scheduler. Note however that in this case we have chosen not to bound the timer to a specific CPU. Nonetheless, as recalled above, you can still optimise the static binding of your task by using the function "rt_get_timer_cpu()" which allows you to know which local APIC is timing your application so that you can "rt_set_runnable_on_cpus" any task accordingly.
See README in "smpscheduler".

Since the TSC is not available on 486 machines for them we use a form of emulation of the "read time stamp clock" (rdtsc) assembler instruction based on counter2 of the 8254. So you can use RTAI also on such machines. Be warned that the oneshot timer on 486 is a performance overkill because of the need of reading the tsc, i.e. 8254 counter2 in this case, 2/3 times. That can take 6-8 us, i.e. more than it takes for a full switch among many tasks while using a periodic timer. Thus only a few khz period is viable, at most, for real time tasks if you want to keep Linux alive.
No similar problems exist for the periodic timer that needs not to use any TSC at all. So, compared to the 20% cited above, the real time performance ratio of the oneshot/periodic timer efficiency ratio can be very low on 486 machines.
Moreover it will produce far worse jitters than those caused on Pentiums and upward machines. If you really need a oneshot timer buy at least a Pentium. Instead, for a periodic timing 486s can still be more than adequate for many applications.

The MUP scheduler instead derives its name by the fact that real time tasks MUST be bound to a single CPU at the very task initialization. They can be afterward moved by using functions “rt_set_runnable_on_cpus” and “rt_set_runnable_on_cpuid”. The MUP scheduler can however use inter CPUs services related to semaphores, messages and mailboxes. The advantage of using the MUP scheduler comes mainly from the possibility of using mixed timers simultaneously, i.e. periodic and oneshot, where periodic timers can be based on different periods, and of possibly forcing critical task on the CPU cache.
With dual SMP machines we cannot say that there is a noticeable difference in efficiency.
MUP has been developed primarily for our, not so fast, a few khz, PWM actuators, BANG-BANG air jet thrusters, coupled to a periodic scheduler.
All the functions of UP and SMP schedulers are available in the MUP scheduler.

APPENDIX B0

Pierre Cloutier's: How does LXRT works?

This one pager is an attempt to explain conceptually how **LXRT** works. It does not try to get into the nifty gritty details of the implementation but it tries to explain how the context of execution switches between Linux and **RTAI**.

But first, what are we trying to do?

LXRT provides a family of real time scheduler services that can be used by both real time RTAI tasks and Linux tasks. To keep things simple for the programmer the implementation is fully symmetric. In other words, the same function calls are used in both the kernel and user space.

What are those real time scheduler services?

RTAI provides the standard services like resume, yield, suspend, make periodic, wait until etc. You will also find semaphores, mail boxes, remote procedure calls, send and receive primitives integrated into the state machine of the real time scheduler. Typically, the IPC function calls support:

. Blocking until the transaction occurs.
. Returning immediately if the other end is not ready.
. Blocking for the transaction until a timeout occurs.

How do I setup my Linux program for LXRT?

You call rt_task_init(name, ...). The call differs from the real time counterpart (there are a few exceptions to the symmetry rule) in that, among other things, you provide a name for your program. The name must be unique and is registered by LXRT. Thus, other programs, real time or not, can find the task pointer of your program and communicate with it.

LXRT creates a real time task who becomes the "angel" of your program. The angel's job is to execute the real time services for you. For example, if you call rt_sleep(...), LXRT will get your angel to execute the real rt_sleep() function in the real time scheduler. Control will return to your program when the angel returns from rt_sleep().

With LXRT, can a Linux task send a message to a real time task?

Yes. You simply use the rt_send(...) primitive that you would normally use in the code of a kernel program. LXRT gets your angel to execute rt_send(...). Control returns to your program when the target task completes the corresponding rt_receive(...) call.

What happens when I send a message to another user space program?

Well, pretty much the same thing except that you now have two angels talking to each other...

Can a real time task also register a name with LXRT?

Yes. The call rt_register(name, ...) does that. Thus, other programs, real time or not, can find the task pointer of your program and communicate with it.

Where do I put the code for the "angels"?

There is not any code required for the real time component of your Linux task. LXRT uses the standard RTAI scheduler functions for that. In the QNX world, the "angel" is called a virtual circuit.

How does it work from the point of view of a user space program?

The inline functions declared in rtai_lxrt.h all do a software interrupt (int 0xFC). Linux system calls use the software int 0x80. Hence the approach is similar to a system call. LXRT sets the interrupt vector to call rtai_lxrt_handler(void), a function that saves everything on the stack, changes ds and es to __KERNEL_DS and then calls lxrt_handler, the function that does the work.

lxrt_handler(...) extracts the first argument from user space and decides what to do from the service request number srq. For real time services, lxrt_resume(...) is called with the scheduler function address pointer fun, the number of remaining arguments, a pointer to the next argument, a service type argument, and the real time task pointer. lxrt_resume(...) will do what is necessary to change the context of execution to RTAI and transfer execution to the specified function address in the real time scheduler.

lxrt_resume(...) first copies the other arguments on the stack of the real time task. Any required data is also extracted from user space and copied into rt_task->msg_buf. At this point, the addresses of three functions are stored above stack_top (LXRT made sure this wizardry would be possible when it first created the real time task):

top-1 lxrt_suspend(...)
top-2 fun(...)
top-3 lxrt_global_sti(...)

The stack is changed to point to top-3, global interrupts are disabled and and the context of execution is switched to RTAI with the call to LXRT_RESUME(rt_task). RTAI executes lxrt_global_sti(...), fun(...), and eventually lxrt_suspend(...). Remember that fun(...) is a RTAI scheduler function like, for example, rt_rpc(...). At this point, fun(...) may or may not complete.

The easy way back to user space - fun(...) completes immediately:

RTAI enters function lxrt_suspend(...) that sets the real time task status to 0 and calls rt_schedule(). The context of execution is eventually switched back to Linux and the system call resumes after LXRT_RESUME(rt_task). Data for mail boxes is copied to user space and a jump to ret_from_intr() is made to complete the system call.

The long way back to user space - fun(...) cannot completed immediately:

RTAI schedules Linux to run again and the state of the real time task is non zero, indicating it is held. Therefore, the system call cannot return to user space and must wait. So it sets itself TASK_INTERRUPTIBLE and calls the Linux scheduler.

Eventually fun(...) completes and RTAI enters function lxrt_suspend(...) that notices the system call is held. So RTAI pends a system call request to instruct Linux to execute another system call whose handler is function lxrt_srq_handler(void). When Linux calls lxrt_srq_handler(), the original system call is re-scheduled for execution and returns to user space as explained above.

What happens to the registered resources if the Linux task crashes?

The "informed" version of LXRT has setup a pointer to a callback function in the do_exit() code of the Linux kernel. The callback is used to free the resources that where registered by the real time task. It also deletes the real time task and unblocks any other task that may have been SEND, RPC, RETURN or SEM blocked on the real time task.

What about mail boxes?

The mail box IPC approach is connection less. In other words, it is not possible for a zombie real time task to detect that another task is MBX blocked specifically for a message from him. The solution here is to use the rt_mbx_receive_timed() with a timeout value and verify the return value to detect the error.

What about performance?

Intertask communications with LXRT are about 36% faster than with old FIFO's. Testing Linux ⇔ Linux communications with int size msg and rep's on a P233 I got these numbers:

LXRT	12,000 cycles RTAI-0.9x :-)
LXRT	13,000 cycles RTAI-0.8
Fifo	19,000 cycles RTAI-0.8
Fifo new	22,300 cycles RTAI-0.8 10% more cycles, a lot more utilities (that cause some overhead)
SRR	14,200 cycles QNX 4 Send/Receive/Reply implemented with a Linux module without a real time executive.

APPENDIX B1

LXRT and hard real time in user space.

We provide hard real time services in user space, also for normal, i.e. non root, users. We think that it will not be as good a performer as kernel space hard real time task modules, but a few microseconds more latency can be acceptable for many applications, especially on most recent almost Ghz CPUs.
Many users will be glad with it for itself. At the very least, it will be useful in easing development and many other things: training, teaching and so on. It is wholly along the basic LXRT concept so we have seen it as an extension to LXRT.

To get hard real time in user space you need a fully preemptable kernel. The question, within RTAI "philosophy", is how to get full preemption with minimum changes, possibly none, to the kernel source.

The solution calls for a compromise. We propose to accept that a hard real time process does no Linux context kernel operations leading to a task switch. In that sense, it is better to speak of a "user space kernel module", and we will use the two terms interchangeably.

The approach is similar to what one of us did when he was using QNX: he always mated a hard real time tasks with a buddy for any I/O operation that could lead to excessive delays. In fact, even within such a good and fully preemptable kernel, I/Os could lead to deadlines misses under heavy hard real time I/O load from many hard real time tasks. Many examples in this distribution, i.e.: clocks, latency calibration and sound, show you a clear picture of how easy it is to use kernel services by mating to a buddy server process, without any problem.
So, at least on the base of our modest experience, that is not an unbearable constraint. Since RTAI has many good intertask services, we do not see any problem in using the same approach again, especially in view of with what Pierre has done, is doing and will do, to make it the "informed" way.

It is nonetheless possible that such a constraint will be somewhat lifted as development proceed. Moreover the user space approach does not forbid you to do it in kernel space, if it is eventually needed. In fact it is not seen as a complete alternative to doing it in the kernel, but simply as a way of giving you more opportunities, at least during the development phase. However the more we use it the more we tend to avoid kernel space, as far as possible.

Taking into account that the present solution is somewhat still at the beginning of its development, we see a lot of space for making it better. For sure it is simpler than the exhausting search of safe scattered kernel pre-emption points many experts are looking at.

How is that possible?

We think that what you'll in RTAI-LXRT shows that it can work, even if it can be improved. The idea is to keep soft interrupts disabled for hard real time user space modules. This way, kernel module hard real time tasks and hard real time interrupts can preempt user space modules, but user space modules cannot be preempted neither by Linux hard interrupt nor by Linux processes.
Linux hardware interrupt are pended as usual for service when RTAI's real time tasks (both in the kernel and user space) are idle.

How does it work?

Hard real time user space modules are just normal Linux processes that mate to a special buddy hard real time kernel task module, as done under LXRT already. They must be POSIX real time Linux processes locked into memory using SCHED_FIFO. Thus their memory must be pre grown to its maximum extension and completely locked in memory. See Linux man pages for mlockall/munlockall.

To distinguish them from usual LXRT firm real time processes the user simply calls rt_make_hard_real_time(), whereas by using rt_make_soft_real_time() he/she can return to standard Linux task switching.
Note that some of the required features, e.g POSIX real time under Linux, require root permission. However by using the function rt_allow_nonroot_hrt() you are allowed to: make a process POSIX real time, lock the memory and do IO operations, as a normal non root user. It is nonetheless necessary that the superuser "insmod"s the required modules (rtai, rtai_sched and lxrt).

The call to rt_make_hard_real_time allows to take a normal process out from the Linux running queue by calling schedule() after having queued the task to a bottom handler. When the bottom handler runs, the task is scheduled as a hard real time module by lxrt_schedule(), and Linux will not know of it, while having it still in its process list.

Lxrt_schedule() is also set as the signal function to be called when returning to the Linux context from a hard real time kernel space schedule, thus ensuring preemption in any case.

Lxrt_schedule() clear the soft interrupt flags and mimics the Linux schedule() function, with scheduling policy SCHED_FIFO, even from within interrupts.

To return to soft real time, rt_make_soft_real_time() does the opposite.

What it currently does:

There are some (not so) simple test processes that runs periodically and on which scheduling latency is measured. No doubt that it does something different as by running the same tasks under the same load with plain LXRT the latency goes as high as Linux 10 ms tick, compared to a 10/20 microseconds under user space modules (preliminary rough measures) and load. Note that within this new context it is likely that you can use also Linux pthreads both for soft and hard real time.
In fact pthreads are normal user processes in disguise, Xavier made a choice, i.e. pthreads as cloned processes, that is good also for LXRT.
Other examples show interacting tasks at work, while the sound task gives an idea of IOs from user space.

The experience gathered so far indicates that, despite the availability of more processing power, under SMP the latency for the same background load can be double/tripled with respect to UP. That is likely due to cache trashing caused by process switches and seems not to depend on the RTAI MP scheduler you are using. So it makes a larger jitter difference, with respect to working in kernel space, using hard real time processes under SMP than under UP.
In fact under UP the jitter is roughly the same weather you are using user or kernel space modules.

What it currently does not very quickly:

Lxrt_schedule() can schedule in and out plain Linux processes, but to do it safely that must happen within Linux idle tasks. Clearly when one tests under heavy load the starting and ending of hard real time mode can be somewhat sluggish. In any case problems are just in starting and ending, once user space modules are in place they are fine.
The matter has been somewhat improved by forcing the scheduling weight of the idle task, just four lines added/modified within the kernel.
We know that there can be other ways of doing it, but all what we could conceive is likely to require heavy kernel modifications. Once more we recall that all our "philosophy" is to deplete the kernel with the slightest changes possible to it, better if none.

Note that within lxrt.c we trapped the kernel sys call and interrupt enabling to be sure that they are not called within hard real time user space modules.
Pierre has conceived the same thing as possible to be done directly in rtai.c, as it can already trap all the reserved Linux traps, but no alternative handler has been implemented yet.

The new additions to lxrt:

- changed rtai_lxrt_handler to avoid ret_from_intr if returning from within a hard real time process;
- added macros my_switch_to(prev,next,last), loaddebug and function switch_to, all copied from Linux;
- added lxrt_schedule to schedule hard real time user space tasks among themselves and to and from the Linux context, with soft flags disabled (cli()), a lot of new data needed are found just above it, the name should self explain them;
- added function lxrt_do_steal to be run from the bh timer to schedule a new hard real time process;
- added the pointer rthal_enint to save the trapped trap rtahl.enint in order to diagnose enable from within rt user space modules;
- added lxrt_enint to actually do the above trapping;
- added lxrt_sigfun to lxrt_schedule when getting back to Linux from the rtai schedulers;
- added steal_from_linux to make a Linux process a user space hard real time module;
- added give_back_to_linux to return a user space module to the Linux processes;
- added linux_syscall_handler to save the trapped Linux sys handler;
- added lxrt_linux_syscall_handler to diagnose calls to sys from hard real time processes;
- print_to_screen to allow a safe printing of diagnosting messages from within user space modules working in hard real time mode.

User functions:

- print_to_screen(const char *format, ...): to safely print information and diagnostic messages in hard real time user space modules;
- void rt_make_soft_real_time(void): to return a hard real time user space process to soft Linux POSIX real time;
- void rt_make_hard_real_time(void): to make a soft Linux POSIX real time process a hard real time LXRT process;
- rt_allow_nonroot_hrt(void): to allow a non root user the make a process Linux POSIX real time, lock process memory in ram and carry out IO operations from user space.

Tests:

There is a wealth of examples to show extended lxrt operations, both in soft and hard real time mode. They can be useful also in giving you some clues for your applications.

Tests list:

- single task (directory one);
- two tasks (directory two);

- many tasks (directory many);
- many tasks (directory forked);
- many pthreads (directory threads);
- latency calibration (directory latency_calibration);
- sound test (directory sound);
- digital clock with semaphores (directory sem_clock);
- digital clock with messages (directory msg_clock).
- task resumed from an interrupt handler (directory resumefromintr).
- press test (directory pressa);
- resume in user space directly from the timer interrupt handler(directory resumefromint).

The possibility of using pthread_create to generate Linux processes is very useful since it allows a task layout that is close to the structure of modules. That could make it easier the translation to kernel modules for maximum performances. Also to be remarked is the possibility of resuming user space modules directly from interrupt handlers, see example reseumefromint.

If you want to check the jitter while one of the clocks or the sound example are running, you should enter the latency_calibration directory under another screen and type "./rt_process 1 &" followed by "./check". Try it varying Linux load. Be carefull, you must end it before closing the clocks/sound tests, see a more detailed comment within README in latency_calibration directory.

Have a look at the README files in each directory for more information.

It is important to remark that what is found under this directory can be used for any application but it is intended mainly for development work. It will be soon ported to lxrt-informed for a safer production use.
Thus it is remarked that you must install a SIGINT handler if you want to safely terminate your LXRT processes, cleaning up any RTAI resource they use, after Ctrl-C. Some examples show how it can be done.
We remind once more that what you find in directory lxrt is the final development version, the related production version is in lxrt-informed.
It may happen that under this directory you can find features not yet ported in lxrt-informed. It will likely be so for a very short time. So take care of abnormal terminations yourself or wait for help from lxrt-informed.

APPENDIX C

A general overview of RTAI fifos.

The new fifo implementation for RTAI maintains full compatibility with the basic services provided by its original NMT-RTL counterpart while adding many more.

It is important to remark that even if RTAI fifo APIs appears as before the implementation behind them is based on the mailbox concepts, already available in RTAI and symmetrically usable from kernel modules and Linux processes. The only notable difference, apart from the file style API functions to be used in Linux processes, is that on the module side you always have only non blocking put/get, so that any different policy should be enforced by using appropriate user handler functions.

With regard to fifo handlers it is now possible to install also one with a read/write argument (read 'r', write 'w'). In this way you have a handler that can know what it has been called for. It is useful when you open read-write fifos or to check against miscalls.
For that you can have a handler prototyped as:

```
int x_handler(unsigned int fifo, int rw);
```

that can be installed by using:

```
rtf_create_handler(fifo_numver, X_FIFO_HANDLER(x_handler).
```

see rtai_fifos.h for the X_FIFO_HANDLER macro definition.
The handler code is likely to be a kind of:

```
int x_handler(unsigned int fifo, int rw);
{
        if (rw == 'r') {
                // do stuff for a call from read and return appropriate value.
        } else {
                // do stuff for a call from write and return appropriate value.
        }
}
```

Even if fifos are strictly no more required in RTAI, because of the availability of LXRT and LXRT-INFORMED, they are kept both for compatibility reasons and because they are very useful tools to be used to communicate with interrupt handlers, since they do not require any scheduler to be installed.
In this sense you can see this new implementation of fifos as a kind of universal form of device drivers, since once you have your interrupt handler installed you can use fifo services to do all the rest.

However the new implementation made it easy to add some new services. One of these is the possibility of using asyncronous signals to notify data availability by catching a user set signal. It is implemented in a standard way, see the function:

```
rtf_set_async_sig(int fd, int signum) (default signum is SIGIO);
```

and standard Linux man for fcntl and signal/sigaction, while the others are specific to this implementation.

A complete picture of what is available can be obtained from a look at rtai_fifos.h prototypes.

It is important to remark that now fifos allows multiple readers/writers so the select/poll mechanism to synchronize with in/out data can lead to unexpected blocks for such cases. For example: you poll and get

that there are data available, then read/write them sure not to be blocked, meanwhile another user gets into and stoles all of your data, when you ask for them you get blocked.

To avoid such problems you have available the functions:

```
rtf_read_all_at_once(fd, buf, count);
```

that blocks till all count bytes are available;

```
rtf_read_timed(fd, buf, count, ms_delay);

rtf_write_timed(fd, buf, count, ms_delay);
```

that block just for the specified delay in milliseconds but are queued in real time Linux process priority order. If ms_delay is zero they return immediately with all the data they could get, even if you did not set O_NONBLOCK at fifo opening.
So by mixing normal read/writes with their friends above you can easily implement blocking, non blocking and timed IOs. They are not standard and so not portable, but far easy to use then the select/poll mechanism.
The standard llseek is also available but it is equivalent to calling rtf_reset, whatever fifo place you point at in the call.

For an easier timing you have available also:

```
rtf_suspend_timed(fd, ms_delay).
```

To make them easier to use, fifos can now be created by the user at open time. If a fifo that does not exist already is opened, it is created with a 1K buffer. Any following creation on modules side resizes it without any loss of data. Again if you want to create a fifo from the user side with a desired buffer size you can use:

```
rtf_open_sized(const char *dev, perm, size).
```

Since they had to be there already to implement our mailboxes we have made available also binary semaphores. They can be used for many things, e.g. to synchronize shared memory access without any scheduler installed and in place of using blocking fifos read/writes with dummy data, just to synchronize.
The semaphore services available are:

```
rtf_sem_init(fd, init_val);

rtf_sem_wait(fd);

rtf_sem_trywait(fd);

rtf_sem_timed_wait(fd, ms_delay);

rtf_sem_post(fd);

rtf_sem_destroy(fd);
```

Note that fd is the file descriptor, a semaphore is always associated to a fifo and you must get a file descriptor by opening the corresponding fifo.

Naturally the above functions are symmetrically available in kernel space but, except for init and create, only for the nonblocking services, i.e: trywait and post.

INDEX

S