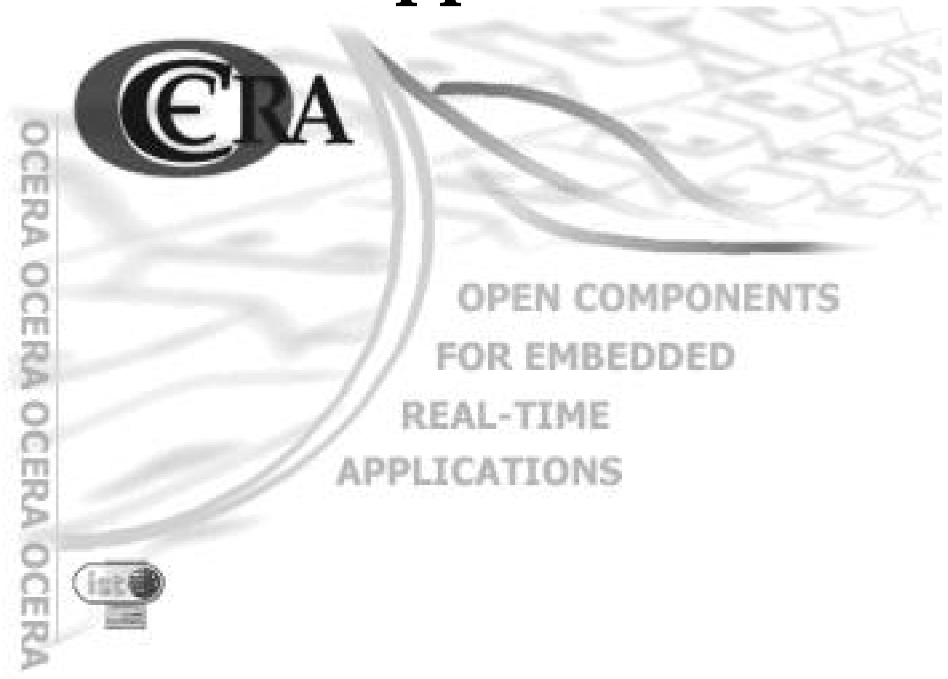
WP10 - Training and Technical support



Deliverable D10.4 - Programmer’s Guide

WP10 - Training and Technical support : Deliverable D10.4 - Programmer's Guide

by Pierre Morel, Jorge Real-Sáez, Ismael Ripoll, Patricia Balbastre, and Miguel Masmano-Tello

Published June 2003

Table of Contents

Document presentation ..... **i**

I. Overview ..... **i**

- 1. Overview 1
- 2. Introduction 2
  - 2.1. Ocera Overview 2
    - 2.1.1. historical choices 2
    - 2.1.2. License 2
      - 2.1.2.1. Linux Licensing 2
      - 2.1.2.2. RTLinux Licensing 2
      - 2.1.2.3. Ocera Licensing 2
    - 2.1.3. Hard versus Soft real time 2
    - 2.1.4. components 2
      - 2.1.4.1. Scheduling 2
      - 2.1.4.2. Fault Tolerance 3
      - 2.1.4.3. Quality Of Service 3
      - 2.1.4.4. Communication 3
  - 2.2. Internals 3
    - 2.2.1. what we will see in this chapter 3
    - 2.2.2. Booting 3
    - 2.2.3. Taking control 3
    - 2.2.4. Interrupt handling 4
    - 2.2.5. Scheduling 5
    - 2.2.6. Quality of service 6
    - 2.2.7. Fault tolerance 6
    - 2.2.8. SMP support 6

II. Development tools **7**

- 3. Hardware 8
  - 3.1. Supported Hardware 8
    - 3.1.1. CPU 8
      - 3.1.1.1. ix86 8
      - 3.1.1.2. PowerPC 8
      - 3.1.1.3. StrongARM 8
    - 3.1.2. System BUS 8
    - 3.1.3. micro controllers 8
  - 3.2. Hard Real Time supported hardware 8
  - 3.3. Soft Real Time supported hardware 8
    - 3.3.1. Standard Hardware 8
    - 3.3.2. Incompatibilities 8
- 4. Development tools 9
  - 4.1. Development environment 9
  - 4.2. Compilers 9
    - 4.2.1. Native Compilers 9
    - 4.2.2. Cross Compilers 9
  - 4.3. Libraries 9
  - 4.4. Debugger 9
  - 4.5. Analyzers 9
  - 4.6. Supervision tools 9
    - 4.6.1. The /proc Filesystem entries 9
    - 4.6.2. Setting POSIX traces 9
    - 4.6.3. Integrated Development environment 9

III. Hard Real Time Components **10**

- 5. Accessing Hard Real time functionalities 11

5.1. What is an OCERA Hard Realtime application 11
5.2. Building an application 11
5.2.1. Makefile 11
5.2.2. Compilers 11
5.2.3. Libraries 11
5.2.4. Choosing the components 11
5.2.4.1. Boot 11
5.2.4.2. Filesystem 11
5.2.4.3. Scheduler 11
5.2.4.4. QOS 11
5.2.4.5. IPC 11
5.2.4.6. Streams 11
5.2.4.7. Network 12
5.2.4.8. RT Network 12
5.2.4.9. Disk access 12
5.2.4.10. User space 12
5.2.4.11. CAN driver 12
5.2.4.12. Other drivers 12
5.2.5. Developing a Real Time task 12
5.2.5.1. General consideration 12
5.2.5.2. Hard Realtime API 12
5.2.6. Developing drivers 12
5.2.6.1. Streams drivers 12
5.2.6.2. Streams modules and heads 13
5.2.6.3. Driver's utilities 14
5.2.6.4. Limitations on drivers implementation 14
5.3. Downloading 14
5.3.1. Embedded systems 14
5.3.2. Hosts systems 14
6. Scheduling components contribution 15
6.1. Dynamic memory allocator 15
6.1.1. Description 15
6.1.2. Usage 15
6.1.3. Programming interface (API) 15
6.1.4. Example 16
6.2. POSIX Signals 16
6.2.1. Description 16
6.2.2. Usage 17
6.2.3. Programming interface (API) 17
6.2.4. Example 18
6.3. POSIX Timers 19
6.3.1. Description 19
6.3.2. Usage 19
6.3.3. Programming interface (API) 19
6.3.4. Example 20
6.4. POSIX tracing 21
6.4.1. Description 21
6.4.2. Main concepts 21
6.4.3. Data types 21
6.4.3.1. Trace Event 22
6.4.3.2. Trace Stream 22
6.4.3.3. Processes Involved in the Tracing Activity 24
6.4.3.4. Trace Controller Process 24
6.4.3.5. The Traced or Target Process 26
6.4.3.6. Trace Analyser Process 27

6.4.4. Additional information 27
6.4.5. Example 28
6.5. POSIX message queues 31
6.5.1. Description 31
6.5.2. Usage 31
6.5.3. Programming interface (API) 32
6.6. Ada Support 32
6.6.1. Description 32
6.6.2. Usage 32
6.6.3. Example 33
6.7. Posix Barriers 34
6.7.1. Description 34
6.7.2. Usage 34
6.7.3. Programming interface (API) 35
6.7.4. Example 35
6.8. Application-Defined Scheduler 37
6.8.1. Description 37
6.8.2. Usage 37
6.8.3. Programming interface (API) 37
6.8.4. Example 38
6.9. Application-Defined Scheduler 45
6.9.1. Description 45
6.9.2. Usage 45
6.9.3. Programming interface (API) 45
6.9.4. Example 46
IV. Soft Real Time Components 54
7. Accessing Soft Real time functionalities 55
7.1. What is an OCERA Soft Realtime application 55
7.2. Underlying Hard Real Time system 55
7.2.1. Overview of functionalities 55
7.2.2. Dependencies 55
7.3. Building an application 56
7.3.1. Makefile 56
7.3.2. Compilers 56
7.3.3. Libraries 56
7.3.4. Choosing the components 56
7.3.4.1. Boot 56
7.3.4.2. Filesystem 56
7.3.4.3. Scheduler 56
7.3.4.4. QOS 56
7.3.4.5. IPC 56
7.3.4.6. Streams 56
7.3.4.7. Network 56
7.3.4.8. RT Network 56
7.3.4.9. Disk access 56
7.3.4.10. CAN driver 56
7.3.4.11. Other drivers 56
7.4. Downloading 56
7.4.1. Embedded systems 56
7.4.2. Hosts systems 56
8. Soft Real time API 57
V. Using new hardware 58
9. Driver Framework 59
9.1. control command with CAN 59
9.2. control command with CAN 59

9.3. real time Ethernet 59
9.4. two real time level application 59
9.5. streaming video 59
10. Porting to new hardware 60
10.1. Supporting new processors 60
10.1.1. The port of Linux and RTLinux exists 60
10.1.2. The port of Linux exists, but not RTLinux 60
10.2. Supporting new Cards 60
10.2.1. Interrupt processing 60
10.2.2. patch process 60
VI. Man pages 62
11. Manuel pages for components 63
12. Manuel pages RTLinux 64
ANNEX 1: Component Manages 65
pthread_getappschedparam 66
posix_appsched_actions_addactivate 66
posix_appsched_execute_actions 67
posix_appsched_invoke_scheduler 68
pthread_appschedattr_seteventmask 69
pthread_attr_setappscheduler 70
pthread_attr_setappschedulerstate 70
pthread_getappschedparam 71
pthread_setspecific_for 72
pthread_sigmask 73
sigpending 74
sigsuspend 74
timer_create 75
timer_delete 76
timer_settime 77
make_linux_task_cbs_server 78
pthread_attr_setinitbudget_np 79
pthread_initcbs_np 80
pthread_setinitbudget_np 80

List of Tables

1. Project Co-ordinator i
2. Participant List i

Document presentation

Table 1. Project Co-ordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14. CP: 46022, Valencia, Spain
Phone:	+34 9877576
Fax:	+34 9877579
E-mail:	alfons@disca.upv.es

Table 2. Participant List

Role	Id.	Name	Acronym	Country
CO	1	Universidad Politécnica de Valencia	UPVLC	E
CR	2	Scuola Superiore S. Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA	CEA	FR
CR	5	UNICONTROLS	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	VISUAL TOOLS S.A.	VT	E

I. Overview

Table of Contents

1. Overview 1
2. Introduction 2

State of this part

This part is under development and must be considered as a draft.
Some chapter are left blank.

Chapter 1. Overview

This is the OCERA Programmer's Guide. We will try through this document to help you to develop hard or soft real time applications.

First we will make a brief presentation of the **targets** you may want to develop using OCERA components for. Like Host Systems or and Embedded System.

We will then focus on the **Development System** with Compiler, Cross compiling, Debugger, Analyzers, and Supervision tools

After this is done, we can present **the Hard real time API**, **the Soft real time API**, the way to develop **drivers** like streams drivers and modules or more *basics* drivers.

The last chapter will be an overview of the ways of **porting to new hardware**.

This document is subject to change along the project, to reflect the up-to-date project.

Chapter 2. Introduction

2.1. Ocera Overview

2.1.1. historical choices

2.1.2. License

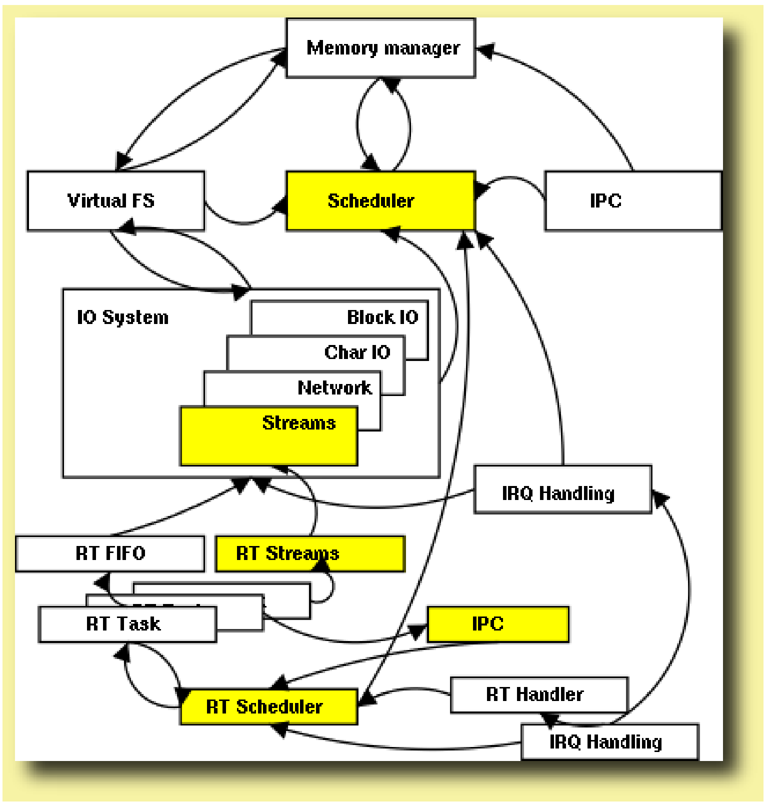
2.1.2.1. Linux Licensing

2.1.2.2. RTLinux Licensing

2.1.2.3. Ocera Licensing

2.1.3. Hard versus Soft real time

image 1



2.1.4. components

2.1.4.1. Scheduling

image 2

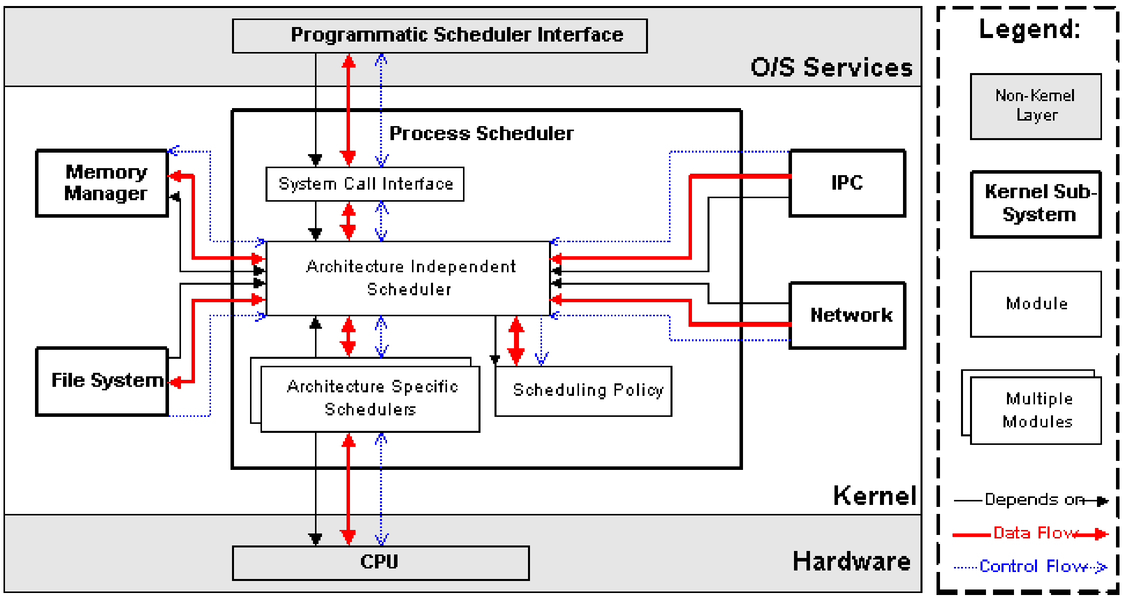


Image 3 ci dessous

2.1.4.2. Fault Tolerance

2.1.4.3. Quality Of Service

2.1.4.4. Communication

2.2. Internals

For a detailed architecture presentation, you may also refer to the **OCERA Architecture** document and to the. **OCERA User's Guide**.

2.2.1. what we will see in this chapter

2.2.2. Booting

2.2.3. Taking control

RTLinux takes control over the Linux kernel as the rtlinux.o module is loaded. The __init routine of the rtlinux.o module calls the "self explained" function *arch_take_over*. A light version of this function is shown here under. In particular we do not detail SMP architecture initialization or *#define* preprocessing here to simplify the presentation. Of course, you can browse the source to see the details of the routine.

```
arch_takeover
        rtl_hard_cli
        rtl_global.flags = g_initialized
        rtl_local.flags = l_ienable | l_idle
        rtl_reschedule_handlers = default_reschedule_handler
        patch_kernel
rtl_hard_sti
rtl_soft_sti
```

As one can see, after clearing interrupts and doing some initialization, the routine initialize the *reschedule_handler* and then calls the once again "self explained" function *patch kernel* routine.

```
patch_kernel
        xdo_IRQ = pfunc[pf_do_IRQ].address (pfunc is a table of functions)
        local_ret_from_intr = pfunc[pf_ret_from_intr].address
        p=find_patch(pfunc[pf_do_IRQ].address
        save_jump(p,pf_do_IRQ)
        patch_jump(p,rtl_intercept)
        pfunc[pf_rtl_emulate_iret] = rtl_soft_sti
        IF LOCAL_APIC
                save_jump(LOCALS_PATCHS)
                zap_ack_apic
                init_local_code
        pre_patch_control=irq_control
        irq_control.do_save_flags = rtl_soft_save_flags
        irq_control.do_restore_flags = rtl_soft_restore_flags
        ...etc replace cli/sti local_irq_save,restore,disable and enable
        ...
        for i < NB_IRQS
                save_linux_irq_desc = h.handler
                h.handler = rtl_generic_type
```

This functions setup the interrupt routine local_ret_from_intr to the address of *rtl_intercept* it then patches the APIC subroutines if a APIC exists by calling *zap_ack_apic* and *init_local_code* and initialize the *irq_control* which contains the address of the routines that will replace the standard Linux routines:

- *do_save_flags*
- *do_restore_flags*
- *cli*
- *sti*
- *local_irq_save*
- *local_irq_restore*
- *local_irq_enable*
- *local_irq_disable*

2.2.4. Interrupt handling

The interrupts hare processed in 4 levels. These three levels are called whenever an interrupt is called this is:

- *rtl_intercept* the real interrupt routine called when the interrupt arrives and responsible for the APIC handling and interrupt acknowledge.
- *dispatch_rtl_handler* dispatch the interrupts to the Real Time handlers.
- *dispatch_linux_irq* dispatch the interrupts to the Linux interrupt handlers.

The main interrupt routine is detailed here under:

```
rtl_intercept
        rtl_spin_lock(rtl_global.har_irq_controller_lock)
        if rtl_irq_controller_get_irq
                rtl_irq_controller_ack
                if G_TEST_RTH (test if IRQ is for RTLinux)
                        rtl_spin_unlock
                        dispatch_rtl_handler
                        rtl_spin_lock
                else
                        G_PEND (set IRQ as pending)
                        G_SET(g_pend_since_sti) (set flags IRQ pending)
                if RUN_LINUX_HANDLER (irq enabled and RT not busy)
                        G_UNPENd
                        rtl_soft_cli
                        G_DISABLE
                        rtl_spin_unlock
                        rtl_hard_sti
                        dispatch_linux_irq
                RETURN_FROM_INTERRUPT_LINUX (simple return)
        rtl_spin_unlock
        RETURN_FROM_INTERRUPT (pop all and IRET)
```

The fourth level is the soft_irq level for linux. This is called whenever the RTLinux scheduler has finished to dispatch the real time tasks. See the details on the real time scheduling in the next section.

```
global flags:
g_rtl_started
g_pend_since_sti
g_initializing
g_initialized

Local flags:
l_busy 1 if RTLinux is scheduling a RT task
l_ienable 1 if soft sti emulation (cli/sti)
l_pend_since_sti 1 if irq pending since last sti
l_psc_active old flag for memory protection PSC module.

Macro:
G_PEND,G_UNPEND,G_ISPEND: 1 if global irq pending
G_ENABLE,G_DISABLE,G_ISENABLE: 1 if irq is globally soft enabled
G_SET_RTH,G_CLEAR_RTH,G_TEST_RTH: 1 if RT Handler set for irq
G_SET,G_CLEAR,G_TEST: 1 if global flag set

L_PEND,L_UNPEND,L_ISPEND: local version
L_SET,L_CLEAR,_L_TEST: local version
L_SET_RTH,L_CLEAR_RTH,L_TEST_RTH: local version


Structures:

rtl_global_handlers[irq] : table for RT Handlers
set: rtl_request_global_irq
clear: rtl_free_global_irq
used: rtl_intercept
activate: G_PEND(irq) and G_SET(g_pend_since_sti)

irqaction *
set: rtl_get_soft_irq (handler)
-> request_irq
-> setup_irq

activate: rtl_global_pend_irq: G_PEND and G_SET(g_pend_since_sti)

LINUX:
do_IRQ
-> handle_IRQ_event
-> action->handler()
-> desc->handler->end()
-> do_softirq()

The way RT_FIFO do it:

1) RTL2LIN:
a) setup a linux handler by calling rtl_get_soft_irq
b) set irq pending with: rtl_global_pend_irq
c) wait to be called by rtl_intercept....
d) call wakeup_interruptible from a safe place
2) LIN2RTL
a) setup a handler with RTF_HANDLER (call in fifo struct.)
to be called on synchronization with linux
on read/write
b) setup a RT handler with RT_RTF_HANDLER to be called
on synchronization with RTLinux tasks on put/get
```

2.2.5. Scheduling

Every time a call is done to irq_control.do_sti (which replace the sti call), the function do_soft_sti is called this function calls rtl_process_pending before to call the rtl_soft_sti_no_emulation function to setup the local ienable.

```
rtl_process_pending
        rtl_soft_cli
        do
                while get_lpended_irq
                        soft_dispatch_local
                while get_gpending_irq
                        soft_dispatch_global
        while G_TEST(g_pend_since_sti | l_pend_since_sti
        if softirq_active(cpu_id)
                do_softirq      /*kernel/softirq*/G
```

2.2.6. Quality of service

2.2.7. Fault tolerance

2.2.8. SMP support

II. Development tools

Table of Contents

3. Hardware 8
4. Development tools 9

State of this part

This part is under development and must be considered as a draft.

Some chapter are left blank.

Chapter 3. Hardware

3.1. Supported Hardware

3.1.1. CPU

3.1.1.1. ix86

3.1.1.2. PowerPC

3.1.1.3. StrongARM

3.1.2. System BUS

3.1.3. micro controllers

3.2. Hard Real Time supported hardware

3.3. Soft Real Time supported hardware

3.3.1. Standard Hardware

3.3.2. Incompatibilities

Chapter 4. Development tools

4.1. Development environment

4.2. Compilers

4.2.1. Native Compilers

4.2.2. Cross Compilers

4.3. Libraries

4.4. Debugger

4.5. Analyzers

4.6. Supervision tools

4.6.1. The /proc Filesystem entries

4.6.2. Setting POSIX traces

4.6.3. Integrated Development environment

III. Hard Real Time Components

Table of Contents

5. Accessing Hard Real time functionalities **11**
6. Scheduling components contribution **15**

State of this part

This part is under development and must be considered as a draft.

Some chapter are left blank.

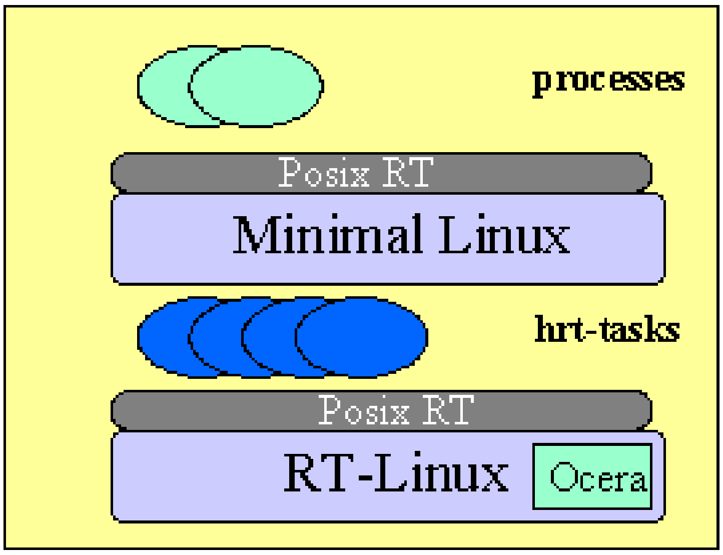
A lot of the work, all the presentation of the scheduling components, has be done by Patricia Balbastre from UPV.

Chapter 5. Accessing Hard Real time functionalities

This chapter is intended for people who need to integrate a Hard Real Time application in the OCERA framework.

Remember that OCERA propose a Framework for embedded Real Time applications. We will see some more detailed aspects of how you can use this framework, but you will need to refer to the *Programmer's guide* if you need details on each functions.

5.1. What is an OCERA Hard Realtime application



5.2. Building an application

5.2.1. Makefile

5.2.2. Compilers

5.2.3. Libraries

5.2.4. Choosing the components

5.2.4.1. Boot

5.2.4.2. Filesystem

5.2.4.3. Scheduler

5.2.4.4. QOS

5.2.4.5. IPC

5.2.4.6. Streams

5.2.4.7. Network

5.2.4.8. RT Network

5.2.4.9. Disk access

5.2.4.10. User space

5.2.4.11. CAN driver

5.2.4.12. Other drivers

5.2.5. Developing a Real Time task

5.2.5.1. General consideration

An important thing by developing Hard real time task is that you may never access data structures from another realtime task without being sure that the other task protect its data structures against possible context switch.

This is particularly true for Linux, which is a task from the point of view of RTLinux. RTLinux is a dirty patch that work with all Linux drivers and modules and as this does not protect every Linux data structures from real time context switch. To overcome this, you will have to split your application in two part, one in the RTLinux context, making a soft IRQ to start the second one in the Linux context.

5.2.5.2. Hard Realtime API

You may want to develop a Real Time Task to handle Hard Real Time Events. Typically, events needing an answer better than 1ms need to be handle in a RTLinux thread or even in a RTLinux driver. We will first see the primitives one can use by developing a RTLinux thread.

- pthread_attr_init
- pthread_attr_setcpu_np
- pthread_attr_setschedparam
- pthread_create

5.2.6. Developing drivers

You can develop drivers as *non standard* drivers or as standard streams drivers.

The streams environment provides a framework to develop drivers and the interfaces to hard real time tasks or to soft real time tasks within Linux Context.

One other important thing with drivers is to define what kind of drivers we develop. In this document we will focus on RTLinux drivers, as oposition to Linux drivers.

The main difference between the two kind of drivers is that we cannot use the same primitive to :

- reserve an interrupt vector
- wakeup associated threads or tasks
- protect our critical code or structures

We also have different duties relative to performance and quality of service.

5.2.6.1. Streams drivers

As said above, we are working within Real Time Linux context. So we must use the following primitives:

- rtl_request_irq
- rtl_free_irq
- rtl_get_soft_irq
- rtl_free_soft_irq
- rtl_hard_enable_irq
- rtl_global_pend_irq
- rtl_critical
- rtl_end_critical

5.2.6.2. Streams modules and heads

List of primitive to use:

- *ocs_module_init*

  This call is needed to obtain a stream structure pointer pointing to a new initialized stream.

- *ocs_module_cleanup*

  This call is needed to free a stream structure .

- *ocs_bind*

  This call is needed to bind two streams structures together. This is called by the *push* system call for example.

- *ocs_unbind*

  This call is needed to unbind two streams structures together. This is called by the *pop* system call for example.

- *ocs_free_mbuf*

  This call is needed to free a message.

- *ocs_free_dbuf*

  This call is needed to free a message.

- *ocs_get_mbuf*

  This call is needed to free a message.

- *ocs_get_dbuf*

  This call is needed to free a message.

- *ocs_get_q*

  This call is needed to free a message.

- *ocs_put_q*

  This call is needed to free a message.

- *ocs_buf_count*

  This call is needed to free a message.

5.2.6.3. Driver's utilities

5.2.6.4. Limitations on drivers implementation

5.3. Downloading

5.3.1. Embedded systems

5.3.2. Hosts systems

Chapter 6. Scheduling components contribution

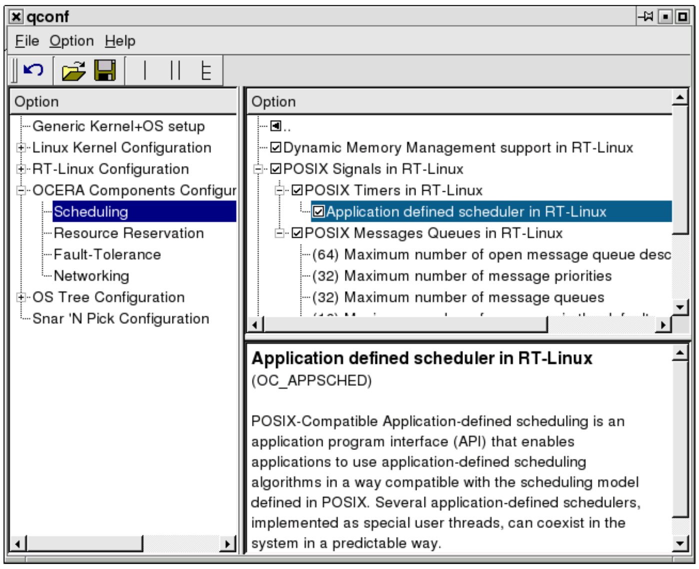
6.1. Dynamic memory allocator

6.1.1. Description

This component provides standard dynamic memory allocation, malloc and free functions, with real-time performance.

6.1.2. Usage

The component is designed to work in three different targets: 1) as a Linux user level library, 2) in the Linux kernel, and 3) in RTLinux.



```
Generic Kernel+OS setup
Linux Kernel Configuration
RT-Linux Configuration
OCERA Components Configur
    Scheduling
    Resource Reservation
    Fault-Tolerance
    Networking
OS Tree Configuration
Snar 'N Pick Configuration
```

```
..
☑Dynamic Memory Management support in RT-Linux
☑POSIX Signals in RT-Linux
  ☑POSIX Timers in RT-Linux
    ☑Application defined scheduler in RT-Linux
  ☑POSIX Messages Queues in RT-Linux
    (64) Maximum number of open message queue desc
    (32) Maximum number of message priorities
    (32) Maximum number of message queues
```

Application defined scheduler in RT-Linux
(OC_APPSCHED)

POSIX-Compatible Application-defined scheduling is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way.

The target dependent code is surrounded by conditional directives that automatically compiles the final object file to the correct target depending on the set of defined macros: __RTL__, __KERNEL__, etc.

When the bigphysarea patch is available in the kernel, the allocator will use this facility by default. Therefore the maximum memory pool will be limited by the memory initially (at boot time with the kernel parameter "mem") allocated by bigphysarea.

When the allocator is compiled as a kernel module (to by used from the Linux kernel or by RTLinux applications), the name of the module is rtl_malloc.o; it can be loaded using the rtlinux script:

```
# rtlinux start
Scheme: (-) not loaded, (+) loaded
  (+) mbuff
  (+) rtl
  (+) rtl_fifo
  (+) rtl_malloc
  (+) rtl_posixio
  (+) rtl_sched
  (+) rtl_time
```

The module accepts the parameter "max_size" which is the size of the initial memory pool in Kbytes. If the parameter is not passed to the module then the default initial memory pool size is 1Mbyte. In order to use more than 1Mb you have to manually load the module.

6.1.3. Programming interface (API)

In order to avoid naming conflicts, the API provided by DIDMA is non POSIX, it looks like the API given by the ANSI "C" standard adding a rt_ prefix.

```
void *rt_malloc(size_t *size);
void rt_free(void *ptr);
void *rt_calloc(size_t nsize, size_t elem_size);
void *rt_realloc(void *p, size_t new_len);
And it also provides several macros which are equal than ANSI-C functions interface for dynamic memory allo
void *malloc(size_t *size);
void free(void *ptr);
void *calloc(size_t nsize, size_t elem_size);
void *realloc(void *p, size_t new_len);
```

6.1.4. Example

```
#include \<rtl_malloc.h\>
#include \<rtl.h\>
#include \<pthread.h\>


pthread_t thread;

void * start_routine(void *arg){
  char *string;
  char hello_world [] = "Hello world";

  rtl_printf("Calling malloc... ");
  // DIDMA malloc
  string = (char *) malloc (sizeof (char) * (strlen (hello_world) + 1));

  if (string == NULL) {
    rtl_printf("WRONG\n");
    return (void *) 0;
  }
  rtl_printf("Malloc OK\n");
  strcpy (string, hello_world);
  rtl_printf ("HELLO_WORLD: %s\n", hello_world);
  rtl_printf ("HELLO_WORLD copy: %s\n", string);
  rtl_printf("Calling free... ");

  // DIDMA free
  free (string);

  rtl_printf("DONE\n");
  return (void *)0;
}

int init_module(void){
  return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void){
  pthread_delete_np (thread);
}
```

6.2. POSIX Signals

6.2.1. Description

This component extends the signaling subsystem of RTLinux to provide user-defined signals and the user signal handlers.

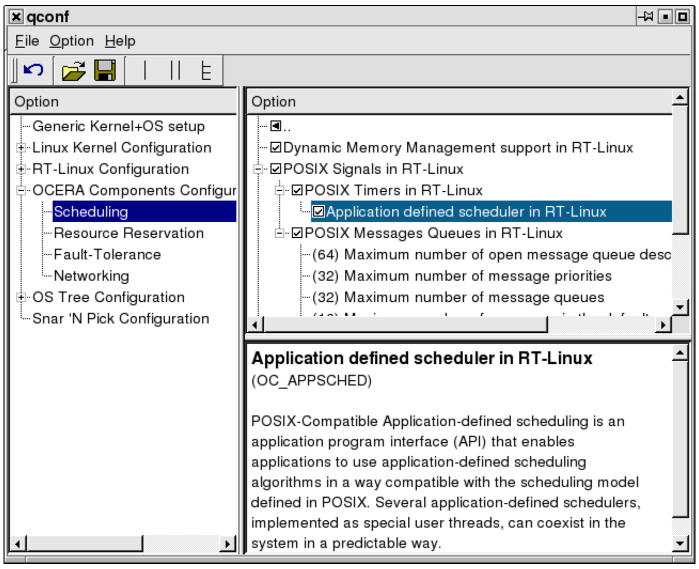
Signals are an integral part of multitasking in the UNIX/POSIX environment. Signals are used for many purposes, including exception handling (bad pointer accesses, divide by zero, etc.), process notification of asynchronous event occurrence (timer expiration, I/O completion, etc.), emulation of multitasking and interprocess communication.

A POSIX signal is the software equivalent of an interrupt or exception occurrence. When a task receives a signal, it means that something has happened which requires the task s attention. Because a thread can send a signal to another thread, signals can be used for interprocess communication. Signals are not always the best interprocess communication mechanism; they are limited and can asynchronously interrupt a thread in ways that require clumsy coding to deal with. Signals are mostly used for other purposes, like the timer expiration and asynchronous I/O completion. There are legitimate reasons for using signals to communicate between processes. First, signals are frequently used in UNIX systems. Another reason is that signals offer an advantage that other communication mechanisms do no support: signals are asynchronous. That is, a signal can be delivered to a thread while the thread is doing something else. The advantages of asynchrony is the immediacy of the notification and the concurrence.

This facility provides only regular UNIX(r) signaling infrastructure. Although realtime POSIX extensions defines an advanced and powerful signal system, its complexity make the implementation more complex and less predictable (since the standard requires that signals can not lost and also delivered in the same order it were generated, then signals can not be internally implemented as bitmaps but lit must be handled as message queues) .

6.2.2. Usage

This facility is optional and has to be selected in the configuration tool. This component is integrated into the RTLinux scheduler module. The functionality is available once the rtl_sched.o module is loaded.



```
Generic Kernel+OS setup
+ Linux Kernel Configuration
+ RT-Linux Configuration
- OCERA Components Configur
    Scheduling
    Resource Reservation
    Fault-Tolerance
    Networking
+ OS Tree Configuration
  Snar 'N Pick Configuration
```

```
..
☑Dynamic Memory Management support in RT-Linux
☑POSIX Signals in RT-Linux
  ☑POSIX Timers in RT-Linux
    ☑Application defined scheduler in RT-Linux
  ☑POSIX Messages Queues in RT-Linux
    (64) Maximum number of open message queue desc
    (32) Maximum number of message priorities
    (32) Maximum number of message queues
```

Application defined scheduler in RT-Linux
(OC_APPSCHED)

POSIX-Compatible Application-defined scheduling is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way.

6.2.3. Programming interface (API)

```
struct rtl_sigaction {
  union {
    void (*_sa_handler)(int);
    void (*_sa_sigaction)(int, struct rtl_siginfo *, void *);
  } _u;
  int sa_flags;
  unsigned long sa_focus;
  rtl_sigset_t sa_mask;
};
/* Macros to manupulate POSIX signal sets.*/
rtl_sigaddset(sigset_t *set, sig);
rtl_sigdelset(sigset_t *set, sig);
rtl_sigismember(sigset_t *set, sig);
```

```
rtl_sigemptyset(sigset_t *set);
rtl_sigfillset(sigset_t *set);
/* Programing actions for signals ocurrences*/
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
/* Set the process s signal blockage mask */
int sigprocmask(int how, const rtl_sigset_t *set, rtl_sigset_t *oset);
int pthread_sigmask(int how, const rtl_sigset_t *set, rtl_sigset_t *oset);
/* Wait for a signal to arrive, setting the given mask */
int sigsuspend(const rtl_sigset_t *sigmask); int sigpending(rtl_sigset_t *set);
/* Send a signall to a thread */
int pthread_kill(pthread_t thread, int sig);
```

This is the standar POSIX API and it is used as the standard defines. Documentation about how signals are programamed can be found in any book about UNIX programming.

6.2.4. Example

```
/*
 * POSIX.1 Signals test program
 *
 */

#include <rtl.h>
#include <rtl_sched.h>

#define MAX_TASKS 2
#define MY_SIGNAL RTL_SIGUSR2
static pthread_t thread[MAX_TASKS];

static void signal_handler(int sig){
  rtl_printf(">------------------------------------->\n");
  rtl_printf("Hello world! Signal handler called for signal:%d\n",sig);
}

static void * start_routine(void *arg) {
 int i=0,err=0,signal;
 struct sched_param p;
 struct sigaction sa;
 rtl_sigset_t set;

 p.sched_priority = 1;
 pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

 signal=MY_SIGNAL+(unsigned) arg;
 rtl_sigfillset(&set);
 rtl_sigdelset(&set,signal);
 pthread_sigmask(SIG_SETMASK,&set,NULL);

 sa.sa_handler=signal_handler;
 sa.sa_mask=0;
 sa.sa_flags=0;
 sa.sa_focus=0;

 if ((err=sigaction(signal,&sa,NULL))<0 )
   rtl_printf("sigaction(%d,&sa,NULL) FAILING, err:%d, errno:%d.\n",
      signal,err,errno);

 pthread_make_periodic_np (pthread_self(), gethrtime(),
 250000000LL+250000000LL * (unsigned) arg);

 rtl_printf("I’m here; my arg is %x iter:%d\n",(unsigned) arg,i++);
 rtl_printf("When i mod 5 -> pthread_kill(pthread_self(),%d)\n",signal);

 while (i<=10) {
   pthread_wait_np ();
   if (!(i%5)) pthread_kill(pthread_self(),signal);
   rtl_printf("I’m here; my arg is %x iter:%d\n",(unsigned) arg,i++);
 }

 rtl_printf("\n\n\n THREAD %d about to end\n\n\n",(unsigned) arg);
 return 0;
}

int init_module(void) {
  int i,err=0;
  for (i=0;i<MAX_TASKS;i++)
    err=pthread_create (&thread[i], NULL, start_routine,(void *) i);
  return err;
}

void cleanup_module(void) {
  int i;
  for (i=0;i<MAX_TASKS;i++)
    pthread_delete_np (thread[i]);
}
```

6.3. POSIX Timers

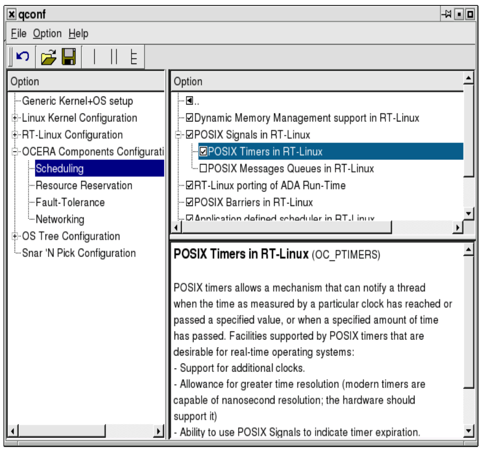
6.3.1. Description

POSIX timers provides mechanisms to notify a thread when the time (measured by a particular clock) has reached a specified value, or when a specified amount of time has passed.

This component provides the functionality to work with several timers per thread. Timer expiration is notified to the thread by mean of a POSIX signal.

6.3.2. Usage

Since a timer expiration causes a signal to be delivered, this facility depends on signal support. To use POSIX timers first you have to select POSIX signals and then select POSIX signals: OCERA Components Configuration -> Scheduling.



☑ Dynamic Memory Management support in RT-Linux
☑ POSIX Signals in RT-Linux
☑ POSIX Timers in RT-Linux
☐ POSIX Messages Queues in RT-Linux
☑ RT-Linux porting of ADA Run-Time
☑ POSIX Barriers in RT-Linux
☑ Application defined scheduler in RT-Linux

POSIX Timers in RT-Linux (OC_PTIMERS)

POSIX timers allows a mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed. Facilities supported by POSIX timers that are desirable for real-time operating systems:
- Support for additional clocks.
- Allowance for greater time resolution (modern timers are capable of nanosecond resolution; the hardware should support it)
- Ability to use POSIX Signals to indicate timer expiration.

This facility is included into the standard scheduler module (rtl_sched.o). Therefore, once the scheduler compiled with timers support is generated and the module is loaded, the user can use the new functions.

6.3.3. Programming interface (API)

Data structure used to specify which action will be performed upon timer expiration:

```
struct sigevent {
  int sigev_notify;    /* notification mechanism */
  int sigev_signo;     /* signal number */
  union sigval sigev_value;  /* signal data value */
}
```

Currently, only two values are defined for sigev_notify: SIGEV_SIGNAL means to send the signal described by the remainder of the struct sigevent; and SIVEV_NONE which means to send no notification at all upon timer expiration.

Next are the system calls to create and delete timers and for arming and consulting the state of an armed timer.

```
/* Creating a timer. Timer created is returned in timer_id location */
int timer_create(clockid_t clockid,
 struct sigevent *restrict evp,
 timer_t *restrict timer_id);
/* Removing timer referenced by timer_id */
int timer_delete(timer_t *timer_id);
/* Setting timer referenced by location timer_id */
int timer_settime(timer_t timer_id, int flags,
 const struct itimerspec *new_setting,
 struct itimerspec *old_setting);
/* Getting time remainning until next expiration*/
int timer_gettime(timer_t timer_id, struct itimerspec *expires);
int timer_getoverrun(timer_t timer_id);
```

The API is fully compliant with POSIX standard. Supported clocks are CLOCK_MONOTONIC and CLOCK_REALTIME. A complete description of POSIX timers and usage examples can be found in chapter five of the Bill O. Gallmeister book: "POSIX.4 Programming fro the Real World".

6.3.4. Example

```
#include <rtl.h>
#include <pthread.h>
#include <time.h>
#include <signal.h>
#define MY_SIGNAL RTL_SIGUSR1
pthread_t thread;
timer_t timer;

#define ONESEC  1000000000LL
#define MILISECS_PER_SEC 1000
hrtime_t start_time;

void timer_intr(int sig){
  rtl_printf("Timer handler called for signal:%d\n",sig);
  pthread_wakeup_np(pthread_self());
}

void *start_routine(void *arg){
  struct sched_param p;
  struct itimerspec new_setting,old_setting,current_timer_specs;
  struct sigaction sa;
  long long period= 120LL*ONESEC;
  hrtime_t now;
  int signal=MY_SIGNAL;

  sa.sa_handler=timer_intr;
  sa.sa_mask=0;
  sa.sa_flags=0;
  new_setting.it_interval.tv_sec= 1;
  new_setting.it_interval.tv_nsec= 0;

  new_setting.it_value.tv_sec=1;
  new_setting.it_value.tv_nsec=start_time;

  /* Install the signal handler */
  sigaction(signal, &sa, NULL))

  /* Arming the timer */
  timer_settime(timer[param],0,&new_setting,&old_setting);

  /* The period of this thread is  2 minutes!!! */
  /* But till will be awaked by the TIMER every second */
  pthread_make_periodic_np (pthread_self(), gethrtime(), period );
  now=gethrtime();
  while (1) {
    last_expiration=now;
    now=gethrtime();
    timer_gettime(timer[param],&current_timer_specs);
    rtl_printf("time passed since last expiration:%d (in milis)\n",
               (int)(now-last_expiration)/MILISECS_PER_SEC);
    pthread_wait_np();
   }
}


int init_module(void) {
  sigevent_t signal;

  /* Create the TIMER */
  signal.sigev_notify=SIGEV_SIGNAL;
  signal.sigev_signo=MY_SIGNAL;
  timer_create(CLOCK_REALTIME,&signal,&(timer[i]));

  start_time=ONEMILISEC;

  // Threads creation.
  pthread_create (&thread), NULL, start_routine, (void *) 0);
  return 0;
}

void cleanup_module(void) {
  pthread_delete_np (thread);
  timer_delete(timer);
}
```

6.4. POSIX tracing

6.4.1. Description

As realtime applications become bigger and more complex, the availability of event tracing mechanisms becomes more important in order to perform debugging and runtime monitoring. Recently, IEEE has incorporated tracing to the facilities defined by the POSIX® standard. The result is called the POSIX Trace standard. Tracing can be defined as the combination of two activities: the generation of tracing information by a running process, and the collection of this information in order to be analysed. The tracing facility plays an important role in the OCERA architecture. Besides its primary use as a debugging and tuning tool, the tracing component jointly with the application-defined scheduler component constitute the key tools for building fault-tolerance mechanisms.

The POSIX trace standard was firstly approved as the amendment 1003.1q of the POSIX 1003.1-1996 standard, and then integrated in the most recent version of POSIX, called 1003.1-2001. Considering that the Trace standard is quite recent, the reader may not be familiar with its concepts and terminology. The following sections provide an introduction to the concepts and the structure of the tracing system.

6.4.2. Main concepts

The POSIX Trace standard is founded on two main data types (trace event and trace stream) and is also based on three different roles which are played during the tracing activity: the trace controller process (the process who sets the tracing system up), the traced or target process (the process which is actually being traced), and the trace analyser process (the process who retrieves the tracing information in order to analyse it). All these concepts are detailed in the following sections.

6.4.3. Data types

6.4.3.1. Trace Event

When a program needs to be traced, it has to generate some information each time it reaches "a significant step" (certain instruction in the program s source code). In the POSIX Trace standard terminology, this step is called a trace point, and the tracing information which is generated at that point is called a trace event. A program containing one or more of this trace points is named instrumented application.

A trace event can be thus defined as a data object representing an action which is executed by either a running process or by the operating system. In this sense, there are two classes of trace events: user trace events, which are explicitly generated by an instrumented application, and system trace events, which are generated by the operating system1.

Any trace event, being either system or user, belongs to a certain trace event type (an internal identifier, of type trace_event_id_t) and it is associated with a trace event name (a human-readable string). For system events, the definition of event types and the mapping between these types and their corresponding names is hard-coded in the implementation of the trace system. Therefore, this event types are common for all the instrumented applications and never change (they are always traced). The trace standard predefines some event types, which are related to the trace system itself, and permits the operating system designer to add some others which may be interesting to that system. The definition of user event types is very different. When an instrumented application wants to generate trace event of a particular type, it has first to create this type. This is done by invoking a particular function (posix_trace_open()) that, given a new trace event name, returns a new trace event type; then, events of this type can be generated from that moment on. If the event name was already registered for that application, then the previously associated identifier is returned. The mapping between user event types and their names is private to each instrumented program and lasts while the program is running.

The generation of a trace event is done internally by the trace system for a system event and explicitly (by the application when invoking posix_trace_event()) for a user trace event. In both cases, the standard defines that the trace system has to store some information for each trace event being generated, including, at least, the following:

a. the trace event type identifier,
b. a timestamp,
c. the process identifier of the traced process (if the event is process-dependent),
d. the thread identifier (of the thread related to the event), if the event is process-dependent and the O.S. supports threads,
e. the program address at which the event was generated,
f. any extra data that the system or the instrumented application wants to associate with the event, along with the data size2.

6.4.3.2. Trace Stream

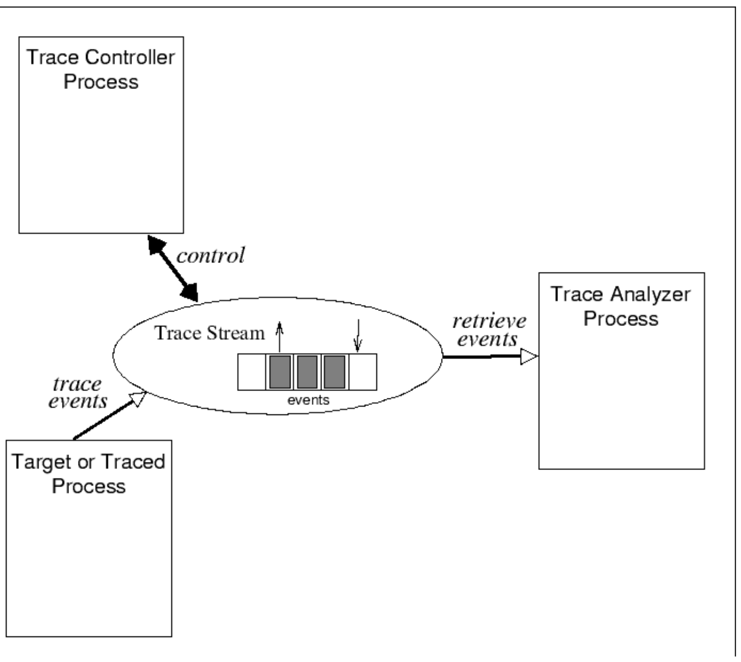
When the system or an application trace an event, all the information related to it has to be stored somewhere before it can be retrieved, in order to be analyzed. This place is a trace stream. Formally speaking, a trace stream is defined as a non-persistent, internal (opaque) data object containing a sequence of trace events plus some internal information to interpret those trace events. The standard does not define a stream as a persisten object and thus it is assumed to be volatile, that is, to reside in main memory.

The standard establishes that, before any event can be stored for a process, a trace stream has to be explicitly created to trace that particular process (the process pid is one of the arguments of the stream creation function). In the most general case, the relationship between streams and processes is many to many. On the one hand, many processes can be traced in a single stream; in particular, this happens if the target process forks after a stream has been created for the (parent) process. On the other hand, the standard permits that many streams are created to trace the same process; if so, each event generated by the process (or by the operating system) is registered in all these streams.

Streams also support filtering. The application can define and apply a filter to a trace stream. Basically, the filter establishes which event types the stream is accepting (and hence storing) and which are not. Therefore, trace events corresponding to types which are filtered out from a certain stream will not be stored in the stream. Each stream in the system(even if associated with the same process) can potentially be applied a different filter. This filter can be applied, removed or changed at any time.

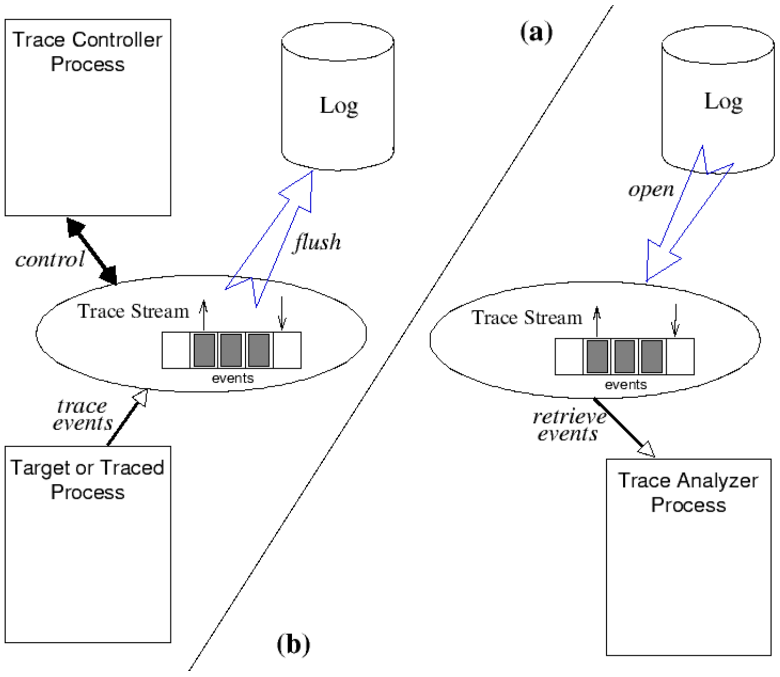
The standard defines two classes of trace streams: active and pre-recorded, which are described below.

a. Active trace stream. This is a stream that has been created for tracing events and has not yet been shut down. This means that it is now accepting events to store. An active trace stream can be of two different types, depending on whether it has been created with or without a log. In a trace stream with log, the stream is created along with a log.



A log is a persistent object (that is, a file) in which the events stored in the stream are saved each time the stream is flushed by the trace system. The trace controller process can create such a stream by calling the function posix_trace_create_withlog(). Thus, events traced from the target process are stored in the stream until it is flushed, either automatically by the trace system or when the trace controller process invokes the posix_trace_flush() function. In either case, the flushing then frees the resources previously occupied by the events just written to the log, making these resources available for new events to be stored. This is shown in Figure 2-(a). In streams with a log, events are never directly retrieved from the stream but from the log (see Pre-recorded trace stream below), once the stream has been shut down. That is, the log is not available for retrieving the events until the tracing of events is over. In a trace stream without log (created by calling posix_trace_create()), trace events are never written to any persistent media, but instead they remain in the stream (in memory) until they are explicitly retrieved. Thus, the stream is accessed concurrently for storing (target process) and retrieving (trace analyser process) events. These accesses can be done only while the stream is active (that is, before it is shut down) since, after that, all the stream resources are freed. Therefore, an active trace stream without a log is used for on-line analysis of events, as shown in Figure 1.

The standard establishes that the trace analyzer process retrieves the events one by one, with the trace system always reporting the oldest stored event first. When this oldest event has been reported, the resources that it was using in the stream have to be freed and then become available for new events to be traced.



If the rate at which events are being traced is higher than the rate at which the trace analyser process is retrieving them from the stream, then the stream may become full. If an active stream without log becomes full, it may either stop accepting events or loop; this depends on the so called stream full policy, which is one of its attributes. In the former case, the stream will start accepting events again when a certain amount of events in the stream have been retrieved, hence freeing resources for the new ones to be stored. In the latter (loop) case, when the stream is full, the oldest recorded events in the stream are lost as new events are stored (that is, the oldest events are overwritten).

b. Pre-recorded trace stream. A stream of this class is used for retrieving trace events which were previously stored in a log. In particular, the log file is opened into a (pre-recorded) stream from which events are then retrieved. Thus, off-line analysis of events is performed in two steps: first, events are traced into an active stream with log; second, after this stream is shut down, the log can be opened into a pre-recorded stream from which the events are retrieved. This process is shown in Figure 2.

6.4.3.3. Processes Involved in the Tracing Activity

The standard defines that up to three different roles can be played in each tracing activity: trace controller process, traced (or target) process and trace analyzer process. In the most general case, each of these roles is executed by a separate process. However, nothing in the standard prevents from having two (or even the three) of these roles executed by the same process. In a small, multi-threaded application, we can have, for example, the three roles played by different threads inside the same process. These roles are now explained in detail.

6.4.3.4. Trace Controller Process

The trace controller process is the process that sets the tracing system up in order to trace a (target) process, which can be the same process or a different one. In particular, this process is in charge of, at least, the following actions:

a. Creating a trace stream with its particular attributes (e.g, if the stream is with or without a log, the stream full policy, etc.). This is further detailed below.

b. Starting and stopping tracing when necessary. This is done by calling posix_trace_start() and posix_trace_stop(), respectively. Each active stream can be in two different states: running or suspended. These two states determine whether or not the stream is accepting events to be stored. The trace controller process can start and stop the stream as many times as it wants. If the stream full policy is to trace until full (POSIX_TRACE_UNTIL_FULL), the trace system will automatically stop the stream when full and start it again when some (or all) of its stored events have been retrieved.

c. Filtering the types of events to be traced. Each stream is initially created with an empty filter (that is, without filtering any event type). If this is not the required behaviour, the trace controller process can build a set of event types (trace_event_set_t), include the appropriate event types in it, and apply it as a filter to the stream (by invoking posix_trace_set_filter()). After that, the stream will reject any event whose type is in the filter set.

d. Shutting the stream down, when the tracing is over (posix_trace_shutdown()). The standard requires that shutting a stream down must free all the stream resources. That is, the stream is destroyed and no more operations can be done on it.

Among all these basic actions, the creation of the stream is the most complex one. This action is done in two steps:

1. Create a stream attribute object (trace_attr_t) and set each of its attributes appropriately. Since this type is also opaque to the user (that is, internal to the trace system), the standard provides a function to initialize an attribute object and then pairs of functions to get and set each of the individual attributes included in the object. Some of these attributes are: the stream name, the stream minimum size, the event data maximum size, the stream full policy, etc. This setting up is performed before invoking the call to create the stream.
2. Create the stream (trace_id_t). There are two different functions to create an active stream, depending on whether it has to be with or without a log. Respectively, these functions are posix_trace_create_withlog() and posix_trace_create(). In either case, the arguments of the creation function are the stream attribute object, previously initialised and set (see above), and the target process pid (process identifier). The main implication of this is that the target process has to exist before the trace controller process can create a stream to trace it. Besides, it has to have enough privileges over the target to do it. The exact definition of this latter requirement depends on the implementation of the trace system. The stream identifier returned in this function can only be used by the process that has created the stream. Only this process can thus directly access the stream in any way. This establishes some limitations that will be commented below.

Optionally, the trace controller process can also perform other actions on the stream, once the stream has been created:

Clearing the stream (posix_trace_clear()). This clears all the events that are now in the stream, but leaves its behaviour (attributes) intact. Clearing the stream makes it exactly in the same state that it was just after being created.

Flushing the stream (posix_trace_flush()). If the stream is created with a log, this action produces an automatic flushing of all the events which are now in the stream to the log. Otherwise, an error is returned.

Querying the stream attributes (posix_trace_get_attr()) and the stream current status (posix_trace_get_status()). The stream status includes whether the stream is currently running or suspended, whether or not an overrun has occurred, etc.

Retrieving the list of event types defined for the stream. The list is retrieved in order, since the function posix_trace_eventtypelist_getnext_id() returns the first event type when it is invoked for the first time, and the next event type in subsequent calls. At any time, the retrieval of event types can be initialised by calling posix_trace_eventtypelist rewind(). Actually, the standard establishes that the event types are not actually associated with a particular stream, but to a particular target process. In other words, the list of event types is the same for all the streams which are tracing the same target.

Mapping event names to event types (posix_trace_trid_eventid_open()). This is normally performed by the target process in order to create its own user event types. However, the trace controller process can use the mapping function in the opposite way: given a well-known user trace event name, the mapping function will return the event type identifier; then, the trace controller process can use that identifier to set up a stream filter, for example.

6.4.3.5. The Traced or Target Process

The traced or target process is the process that is being traced, that is, is the process for which a trace stream has been created and set up. According to the standard, only two functions can actually be called from a target process:

a. A function to register a new user event type for this process (posix_trace_eventid_open()). The input argument of this function is the (new) event type name. If this name has already being registered for that target, then the previously mapped event type identifier is returned. If not, then a new identifier is internally associated with this name and returned. If an implementation defined maximum amount of user event types had already been registered for that target process, then a predefined event type called POSIX TRACE_UNNAMED_USEREVENT is returned. If successful, this registration is valid for all the streams that have been created, or will be created, to trace the target process (even if no stream has still been created for that target). From the user viewpoint, therefore, the identification of user event types is done in a per-name basis (instead of using integer values, for example). This allows for a name space wide enough to avoid collisions when independent pieces of instrumented code are linked together into a single application. This include, for example, the case of linking an instrumented third-party library to our code, even when we do not have the library s source code.

b. A function to trace an event (posix_trace_event()). This function has three input arguments: the event type, which must have been previously registered (see above), a pointer to any extra data that has to be stored along with the event, and the size of this data3. The event is stored in all the streams created for that particular target which are currently running and which do not have the event s type being filtered out.

It is important to point out that neither of these functions accepts a stream identifier as a parameter. That is, according to the standard philosophy, the target is programmed to invoke these functions without being aware (and independently) of actually being traced or not. The result is that calling the posix_trace_event() function has no effect if no stream has been created for the target. In other words, an instrumented running program does not actually become a target process until at least one stream has been created for it. The case of the posix_trace_eventid_open() function is different since, as explained above, the trace system will register any new event type for the program even when no stream has been created for tracing the process.

This philosophy completely decouples the target from the trace controller process, with many interesting advantages. For example, imagine an application that runs for long periods of time without stop (a real-time application or a database, for instance). It may be interesting to know, every once in a while, how this application is performing. Therefore, this (instrumented) application can be the target of an inspector (trace controller) program that, periodically, creates one or more streams to trace it, gets the resulting events, and then destroys the stream(s). Depending on the application characteristics, this occasional tracing may be good enough to check how the application is behaving, and does not overload the system with a continuous tracing.

6.4.3.6. Trace Analyser Process

This process is in charge of retrieving the stored events in order to analyse them. The standard defines three alternative retrieval functions to be used by the trace analyser process:

a. posix_trace_getnext_event(). This function retrieves one event from the stream whose identifier is provided as a parameter. If no event is immediately available, the function blocks the invoking process (or thread) until an event is available.
b. posix_trace_timedgetnext_event(). This function works in a similar fashion than the previous one, but, when no event is immediately available, it blocks the process until either an event is available or an absolute timeout is reached (whatever of both happens first). If the timeout is produced first, the invoking process gets the corresponding error code.
c. posix_trace_trygetnext_event(). This function never blocks the invoking process: it either return a retrieved event or an error code, if no event is available at the moment.

If successful, any of these functions retrieve the oldest event stored in the stream which has not still been reported. The age of each event is calculated according to the automatic timestamp performed by the trace system when the event is recorded.

As explained above, the events can be only be retrieved from two different places: (1) from an active stream without log; (2) from the log of a (previously destroyed) stream with log, once this log has been opened into a (pre-recorded) trace stream. This defines the two kinds of analysis that the standard supports:

a. a)On-line analysis. In this kind of analysis, the trace analyzer process retrieves the events from an active trace stream (without log). As stated above, the retrieval function (any of them) needs to provide the stream s identifier; however, according to the standard, this identifier can only be used within the process that created the stream. This forces that, in an on-line analysis, the trace analyzer process and the trace controller process have to be the same one.
b. Off-line analysis. As explained in Trace Controller Process subsection, this analysis is done in two steps: in the first step, events are recorded into an active trace stream with log that, automatically or under request of the trace controller process, flushes these events to the log (file). Once this step is over, the trace analyser process opens the log into a private, pre-recorded stream (posix_trace_open()), from which it can start retrieving the events. Only the first of the three retrieval functions mentioned above can actually be used in a pre-recorded stream. Obviously, in this case, this function will never make the trace analyser process to block, since all the events are already stored in the stream. From a pre-recorded stream, events are always reported in order (according to the recording timestamp) but they are not erased from the stream after being retrieved. If necessary, the trace analyser process can start retrieving the events again from the oldest one by rewinding the stream (posix_trace_rewind()), without having to re-open the log.

In addition, the trace analyser process can also retrieve other information of the stream (either active or pre-recorded), including the list of registered event types and its names, the stream attribute object (and then each of its individual attributes), the stream current status (for an active stream), etc. All this information is intended to make the trace analyser process able to correctly interpret the trace events which it is retrieving.

6.4.4. Additional information

Since this part of the POSIX standard was published recently, there is still a lack of documentation in the printed form (as fas as the authors know there is not a book that covers this issues of the PSOXI standard), also the implementation done in OCERA was one of the first implementations of the standard. For more information the reader is referred to the online rationale and man pages available at the OpenGroup site: http://www.opengroup.org/onlinepubs/007904975/.

6.4.5. Example

The following example creates three new user event types and a trace stream, and then starts five RTLinux threads. Among them, three periodically execute and just consume CPU, another one periodically wakes up and trace these events, and the last one waits until a new event is available and then retrieves it and writes its contents to the console.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_sched.h>
#include <trace.h>
#include <rtl_ktrace.h>


static trace_id_t       trid;
static trace_event_id_t ev_char, ev_int, ev_string;
static pthread_t        thr1, thr2, thr3, thr4, thr5;

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/**********************************************************/
void *writer(void *dummy) {
  int   i, j, k;
  char  s[164] ="hello world!hello world!hello world!hello world!hello world!hello world!hello world!hello
  char  c;
  void  *data;

  // Create a new event type:
  posix_trace_eventid_open ("user event string", &ev_string);
  c = 'A';
  k = 0;

  pthread_wait_np();

  for (i=0; i<10; i++) {
    for (j=0; j<700000; j++);

    pthread_mutex_lock(&mutex);

    data = (void *) & c;
    posix_trace_event(ev_char, data, sizeof(char));
    data = (void *) & k;
    posix_trace_event(ev_int, data, sizeof(int));

    for (j=0; j<70000; j++);

    posix_trace_event(ev_string, s, sizeof(s));

    // Values for next loop:
    c += 1;
    k += 1;

    pthread_mutex_unlock(&mutex);
    pthread_wait_np();
  }
  return (void *) 0;
}

/**********************************************************/
void *just_execute(void *loops) {
  int i, j, nloops = (int) loops;

  for (i=0; i<100; i++) {
    for (j=0; j<nloops/4; j++);
    pthread_mutex_lock(&mutex);
    for (j=0; j<nloops/2; j++);
    pthread_mutex_unlock(&mutex);
    for (j=0; j<nloops/4; j++);

    pthread_wait_np();
  }

  return (void *) 0;
}

/************************************************************/
void *reader(void *loops) {
  int             error;
  trace_attr_t    trace_attr;
  char            str[TRACE_NAME_MAX];
  struct posix_trace_event_info event;
  char            data[64];
  size_t          datalen;
  int             unavailable;
  int             *ent;
  char            *car;
  trace_event_id_t evid;

  error = posix_trace_get_attr(trid, &trace_attr);
  rtl_printf("get attr (%d)\n", error);

  error = posix_trace_attr_getgenversion(&trace_attr, str);
  rtl_printf( "get genversion (%d): %s\n", error, str);

  posix_trace_eventtypelist_rewind(trid);
  posix_trace_eventtypelist_getnext_id (trid, &evid, &unavailable);
  while (! unavailable) {
    posix_trace_eventid_get_name (trid, evid, str);
    rtl_printf("Event %d name %s\n", evid, str);
    posix_trace_eventtypelist_getnext_id (trid, &evid, &unavailable);
  }

  error = 0; unavailable = 0;

  while (! error && ! unavailable) {

    event.posix_event_id = 1024;
    error = posix_trace_getnext_event(trid,
       &event,
       &data,
       sizeof(data),
       &datalen,
       &unavailable);

    if(error) {
      rtl_printf("No more events (%d). Exiting\n", error);

    } else if (unavailable) {
      rtl_printf( "   Event unavailable\n");

    } else {
      posix_trace_eventid_get_name (trid, event.posix_event_id, str);

      // Now switch depending on the event type (name):
      if (!strcmp(str,"user event char")) {
 car = (char *) data;
 rtl_printf( "  Time =%ld.%ld. Event %d (%s) with data=%c (size = %d)\n",
     event.posix_timestamp.tv_sec, event.posix_timestamp.tv_nsec,
     event.posix_event_id, str, *car, datalen);
      }
      else if (!strcmp(str,"user event int")) {
 ent = (int *) data;
 rtl_printf( "  Time =%ld.%ld. Event %d (%s) with data=%d (size = %d)\n",
     event.posix_timestamp.tv_sec, event.posix_timestamp.tv_nsec,
     event.posix_event_id, str, *ent, datalen);
      }
      else if (!strcmp(str,"user event string")) {
 rtl_printf( "  Time =%ld.%ld. Event %d (%s) with data=%s (size = %d)\n",
     event.posix_timestamp.tv_sec, event.posix_timestamp.tv_nsec,
     event.posix_event_id, str, (char *) data, datalen);
      }
      else {
 rtl_printf( "  Time =%ld.%ld. Event %d (%s) with data unknown\n",
     event.posix_timestamp.tv_sec, event.posix_timestamp.tv_nsec,
     event.posix_event_id, str);
      }
    }
  }
  rtl_printf("Error = %d   Unavailble = %d \n", error, unavailable);

  return (void *) 0;
}

/************************************************************/
int init_module(void) {

  trace_attr_t  attr;
  pthread_attr_t thattr;
  trace_event_set_t set;
  int            error;

  // Start the automatic tracing of kernel events:
  rtl_ktrace_start();

  // Create and set the trace attribute:
  error = posix_trace_attr_init(&attr);

  error = posix_trace_attr_setstreamfullpolicy (&attr, POSIX_TRACE_UNTIL_FULL);
  error = posix_trace_attr_setname(&attr, TRACE_STREAM1_NAME);
  error = posix_trace_attr_setmaxdatasize(&attr, 64);
  error = posix_trace_attr_setstreamsize(&attr, 4096);

 // Create the stream:
  error = posix_trace_create(0, &attr, &trid);
  if (error) return -1;

   // Create new event types associated with this stream:
  error = posix_trace_trid_eventid_open (trid,"user event char",  &ev_char);
  error = posix_trace_trid_eventid_open (trid,"user event int",   &ev_int);


  // Set the stream filter to only record user events:
  posix_trace_eventset_fill(&set, POSIX_TRACE_SYSTEM_EVENTS);

  error = posix_trace_set_filter(trid, (const trace_event_set_t *) &set,
    POSIX_TRACE_SET_EVENTSET);

  // Start tracing:
  error = posix_trace_start(trid);
  if (error) return -1;



  // Create the 'writer' task (the one which traces user events):
  pthread_attr_init (&thattr);
  pthread_create (&thr1, &thattr, writer, 0);
  pthread_make_periodic_np(thr1, 0, (hrtime_t) 400000000);

  // Create  other tasks which just consume cpu
  // This one awakes each 20 msec:
  pthread_create (&thr2, &thattr, just_execute, (void *) 100000);
  pthread_make_periodic_np(thr2, 0, (hrtime_t) 20000000);

  // This one awakes each 25 msec:
  pthread_create (&thr3, &thattr, just_execute, (void *) 300000);
  pthread_make_periodic_np(thr3, 0, (hrtime_t) 25000000);

  // This one awakes each 50 msec:
  pthread_create (&thr4, &thattr, just_execute, (void *) 200000);
  pthread_make_periodic_np(thr4, 0, (hrtime_t) 50000000);

  // Create the 'reader' task (awakes only once):
  pthread_create (&thr5, &thattr, reader, (void *) 200000);

  return 0;
}

/************************************************************/
void cleanup_module(void) {
  rtl_printf("rtl_tasks: CLEANUP!!!\n");

  // Stop and shutdown the stream:
  posix_trace_shutdown(trid);

  // Delete the tasks:
  pthread_delete_np(thr1);
  pthread_delete_np(thr2);
  pthread_delete_np(thr3);
  pthread_delete_np(thr4);
  pthread_delete_np(thr5);

  // Stop the tracing of kernel events:
  rtl_ktrace_stop();
}
```

6.5. POSIX message queues

6.5.1. Description

This component implements POSIX message queues facility which can be used to send messages between RTLinux threads.

UNIX systems offers several possibilities for interprocess communication: signals, pipes and FIFO queues, shared memory, sockets, etc. In RTLinux, the most flexible one is shared memory, but the programmer has to use alternative synchronisation mechanism to build a safe communication mechanism between process or threads. On the other hand, signals and pipes lack certain flexibility to establish communication channels between process. In order to cover some of these weaknesses, POSIX standard proposes a message passing facility that offers:

Protected and synchronised access to the message queue. Access to data stored in the message queue is properly protected against concurrent operations.

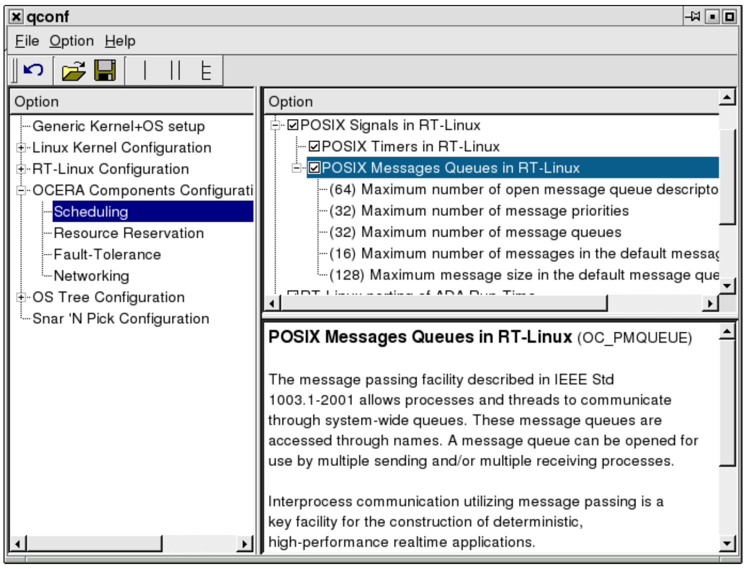
Prioritised messages. Processes can build several flows over the same queue, and it is ensured that the receiver will pick up the oldest message from the most urgent flow.

Asynchronous and temporised operation. Threads have not to wait for operation to be finish, i.e., they can send a message without having to wait for someone to read that message. They also can wait an specified amount of time or nothing at all, if the message queue is full or empty.

Asynchronous notification of message arrivals. A receiver thread can configure the message queue to be notified on message arrivals. That thread can be working on something else until the expected message arrives.

6.5.2. Usage

This component is compiled as a separate kernel module. In order to use this facility you have to select it at the main configuration screen and the compile the RTLinux sources. This facility depends on POSIX signals so you need to select POSIX signals in order to enable the message queues selection box.



Generic Kernel+OS setup
Linux Kernel Configuration
RT-Linux Configuration
OCERA Components Configurati
Scheduling
Resource Reservation
Fault-Tolerance
Networking
OS Tree Configuration
Snar 'N Pick Configuration

☑POSIX Signals in RT-Linux
☑POSIX Timers in RT-Linux
☑POSIX Messages Queues in RT-Linux
(64) Maximum number of open message queue descripto
(32) Maximum number of message priorities
(32) Maximum number of message queues
(16) Maximum number of messages in the default messa
(128) Maximum message size in the default message que

POSIX Messages Queues in RT-Linux (OC_PMQUEUE)

The message passing facility described in IEEE Std 1003.1-2001 allows processes and threads to communicate through system-wide queues. These message queues are accessed through names. A message queue can be opened for use by multiple sending and/or multiple receiving processes.

Interprocess communication utilizing message passing is a key facility for the construction of deterministic, high-performance realtime applications.

6.5.3. Programming interface (API)

This components follows the POSIX API specification for message passing facility defined in IEEE Std 1003.1-2001. This API also belongs to the Open Group Base Specifications Issue 6. The following synopsis presents the list of supported message queue functions:

```
/* Create and destroy message queues */
mqd_t mq_open (const char *, int, ...);
int mq_unlink (const char *)
int mq_close (mqd_t); int mq_getattr (mqd_t, struct mq_attr *);
int mq_notify (mqd_t, const struct sigevent *);
int mq_setattr (mqd_t, const struct mq_attr *, struct mq_attr *);
ssize_t mq_receive (mqd_t, char *, size_t, unsigned *);
int mq_send (mqd_t, const char *, size_t, unsigned );
ssize_t mq_timedreceive (mqd_t, char *, size_t, unsigned *, const struct timespec *);
int mq_timedsend (mqd_t, const char *, size_t, unsigned, const struct timespec *);
```

6.6. Ada Support

6.6.1. Description

This component is a porting of the Gnat compiler run time support to the RTLinux executive. With this porting it is possible to use the ADA language to program hard realtime applications in RTLinux.

Ada is a standard programming language that was designed with a special emphasis on real-time and embedded systems programming, also covering other parts of modern programming such as distributed systems, systems programming, object oriented programming or information systems.

6.6.2. Usage

The Gnat porting is a complex component that modifies the some scripts of the installed Gnat compiler code. The ported Gnat run time support runs on top of the RTLinux, and use the RTLinux API as a "normal" RTLinux application. That is, it neither modifies the OCERA-RTLinux code nor add any new file. For this reason it has not been integrated into the OCERA framework but has to be installed separately.

Although not necessary, it is convenient to have experience using Gnat the compiler before installing RTLGnat.

Next are the installation steps to install RTLGnat (Version 1.0) in the Gnat system (assuming that the running Linux kernel and RTLinux executive are the one of the OCERA framework):

```
1.Please, be sure that you have the original GNAT compiler distributed by ACT (ftp://cs.nyu.edu/pub/gnat/),

2.Select, at least, the following options in the main OCERA config tool:
1.RTLinux Configuration -> Priority inheritance (POSIX Priority Protection)
2.RTLinux Configuration ->  Floating point support
3.OCERA Component Conf. -> Scheduling ->  Dynamic memory manager...
4.OCERA Component Conf. -> Scheduling ->  POSIX Signals ...
5.OCERA Component Conf. -> Scheduling -> POSIX Trace (recommended)

3.Edit the (RTLGnat)/Makefile and modify the path variables to point to the right directories. Among others

4.Make sure that the proper version of gnat is at the beginning of the PATH variable, for example:  export

5.You may need root privileges (write access to the GNAT directory) to compile RTLGnat because it will add

6.Compile RTLGnat by running "make" from the (RTLGnat) directory.

7.Now RTLGnat has been installed and compiled jointly with the standard Gnat distribution. You can find the
```

RTLGnat will be installed in the directory where GNAT is already installed. The modifications to the GNAT installation will include a directory called rts-rtlinux, where the needed libraries will be located, and the executables rtlgnatmake, rtload and rtunload.

The "rtlgnatmake" script is the equivalent to "gnatmake" in GNAT for Linux. Simply run:

```
# rtlgnatmake my_app.adb
to obtain the application module "my_app"
```

Once you have created your application object module, RTLinux needs to be loaded in order to run your application:

```
# rtlinux start
```

Now you should start up your application by doing:

```
# rtload my_app
```

To terminate and remove your running application just run:

```
# rtunload my_app
```

6.6.3. Example

Following example makes use of the POSIX trace component to with Ada.Real_Time;

```
use Ada.Real_Time;
with RTL_Pt1;
use RTL_Pt1;

procedure Tasks is

   task type Std_Task (Id : Integer) is
     pragma Priority(Id);
     entry Call;
   end Std_Task;

   task body Std_Task is
      Next_Time : Time;
      Period : Time_Span := Microseconds (100);
      Next_While : Time;
      Period_While : Time_Span := Microseconds (10);
   begin
      accept Call;
      Next_Time := Clock + Period;
      loop
     --  Put ("I am "); Put (Id); New_Line;
 Next_While := Clock + Period_While;
         while Next_While > Clock loop
             null;
         end loop;
         delay until Next_Time;
         Next_Time := Clock + Period;
      end loop;
   end Std_Task;

   Std_Task1 : Std_Task(1);
   Std_Task2 : Std_Task(2);
  dev : Integer;
begin
--   Std_Task1.Call;
   dev := Integer(rtl_ktrace_start);
   Std_Task1.Call;
   Std_Task2.Call;
   delay 0.005;
   dev := Integer(rtl_ktrace_stop);
--   Std_Task2.Call;
end Tasks;
```

6.7. Posix Barriers

6.7.1. Description

Barriers, are defined in the advanced real-time POSIX (IEEE Std 1003.1-2001), as part of the advanced real-time threads extensions. A barrier is a simple and efficient synchronisation utility.

These are the steps to create and to use a barrier

1. The barrier attributes are initialized . This is accomplished trough the function pthread_barrierattr_init
2. 2.The barrier is initialized, only once, by calling the function pthread_barrier_init.

   This function set the attributes of the barrier (specified in the previous step, or it takes a default attribute object) and the parameter count, which specifies the number of threads that are going to synchronise at the barrier.

   Although standard posix recommends that the value specified by count must be greater than zero, if count is 1, the barrier will not take effect, since no blocking would be produced. Therefore, in this implementation, a value of count less or equal than 1 is not valid. Otherwise, EINVAL is returned.

3. When a thread wants to synchronise at the barrier, it calls the function pthread_barrier_wait .

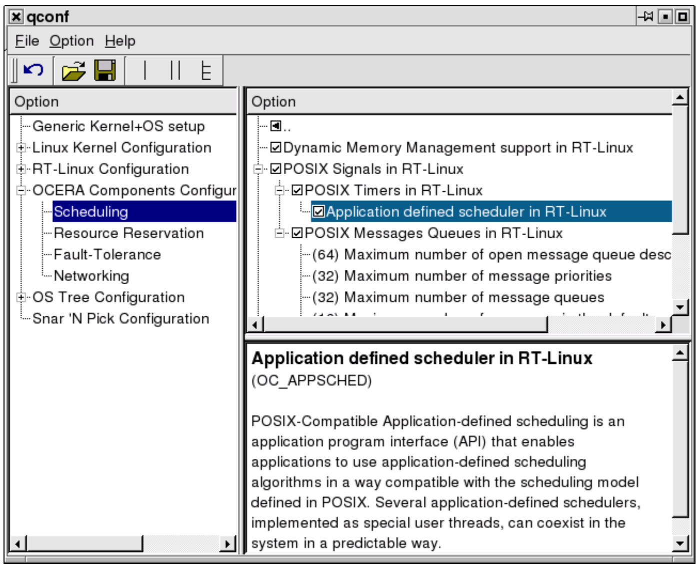
   At this point, the thread will wait until all the rest of the threads have reached the same function call. Threads will continue its execution when the last thread reaches the pthread_barrier_wait function.

4. Finally, both the barrier and the attributes have to be destroyed (pthreadd_barrierattr_destroy and pthread_barrier_destroy).

If there are threads waiting on the barrier, the function pthread_barrier_destroy does not destroy the barrier, but exits with error EBUSY.

6.7.2. Usage

To activate the component just mark the option "Posix Barriers in RT-Linux" inside of "Ocera Components Configuration -> Scheduling" in the configuration tool (Figure 3).



Generic Kernel+OS setup
Linux Kernel Configuration
RT-Linux Configuration
OCERA Components Configur
Scheduling
Resource Reservation
Fault-Tolerance
Networking
OS Tree Configuration
Snar 'N Pick Configuration

☑Dynamic Memory Management support in RT-Linux
☑POSIX Signals in RT-Linux
☑POSIX Timers in RT-Linux
☑Application defined scheduler in RT-Linux
☑POSIX Messages Queues in RT-Linux
(64) Maximum number of open message queue desc
(32) Maximum number of message priorities
(32) Maximum number of message queues

Application defined scheduler in RT-Linux
(OC_APPSCHED)

POSIX-Compatible Application-defined scheduling is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way.

This component has no dependencies with other Ocera components or RTLinux facilities. Posix Barriers are a high level RTLinux component, since it does not modify the RTLinux source code, but adds new features. Barriers are not implemented as a module, it is only necessary to insert the scheduler module (rtl_schedule.o). Barrier functionalities are implemented in two files: rtlinux/schedulers/rtl_barrier.c and rtlinux/include/rtl_barrier.h.

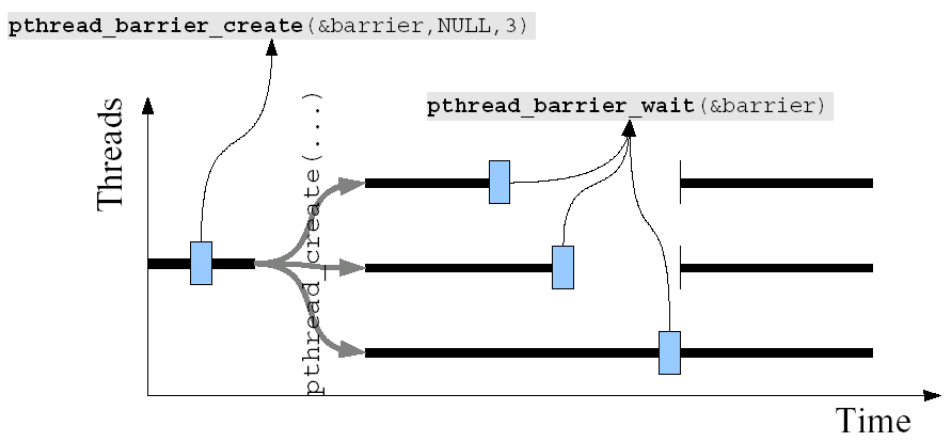
6.7.3. Programming interface (API)

The API is defined by the POSIX standard. Here is a list of the functions that have been implemented.

```
int pthread_barrierattr_destroy(pthread_barrierattr_t * attr );
The pthread_barrierattr_destroy() function shall destroy a barrier attributes object. A destroyed attr attr
int pthread_barrierattr_init(pthread_barrierattr_t * attr );
The pthread_barrierattr_init() function shall initialize a barrier attributes object attr with the default
Results are undefined if pthread_barrierattr_init() is called specifying an already initialized attr attrib
int pthread_barrier_init(pthread_barrier_t * barrier, const pthread_barrierattr_t *attr, unsigned int count
The pthread_barrier_init() function shall allocate any resources required to use the barrier referenced by
int pthread_barrier_destroy( pthread_barrier_t * barrier );
The pthread_barrier_destroy() function shall destroy the barrier referenced by barrier and release any reso
int pthread_barrier_wait( pthread_barrier_t * barrier );
The pthread_barrier_wait() function shall synchronize participating threads at the barrier referenced by ba
When the required number of threads have called pthread_barrier_wait() specifying the barrier, the constant
int pthread_barrierattr_getpshared( const pthread_barrierattr_t * attr int * pshared );
The pthread_barrierattr_getpshared() function shall obtain the value of the       process-shared attribute
int pthread_barrier_wait( pthread_barrier_t * barrier );
The pthread_barrier_wait() function shall synchronize participating threads at the barrier referenced by ba
When the required number of threads have called pthread_barrier_wait() specifying the barrier, the constant
```

6.7.4. Example

A barrier can be used to force periodic threads to execute its first activation at the first time. This example, in this case, will consist of one barrier. Three threads block on the barrier before becoming periodic. When the last thread arrives to the barrier, then all threads are allowed to continue execution (see Figure 4).


pthread_barrier_create(&barrier,NULL,3)
Threads
pthread_create(...)
pthread_barrier_wait(&barrier)
Time

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <rtl_sched.h>
#include <rtl_barrier.h>
#include <rtl.h>
#include <rtl_time.h>

#define NTASKS 3

pthread_t tasks[NTASKS];
hrtime_t now;

pthread_attr_t attrib;
struct {
    int id;
    int compute;
    int period;
} sched_attrib[NTASKS];


pthread_barrierattr_t barrier_attr;
pthread_barrier_t my_barrier;

void * fun(void *arg) {

    int id = (int)arg;

    pthread_barrier_wait(&my_barrier);

    pthread_make_periodic_np(pthread_self(), now, sched_attrib[id].period);

    while (1){
     rtl_delay(sched_attrib[id].compute);
     pthread_wait_np();
     fin--;
    }

    pthread_exit(0);
    return (void *)0;
}


int init_module(void)
{
    int x;

    sched_attrib[0].compute=1000;
    sched_attrib[0].period=100000;
    sched_attrib[1].compute=1900;
    sched_attrib[1].period=170000;
    sched_attrib[2].compute=25000;
    sched_attrib[2].period=200000;

    now = gethrtime();

    //Initialize the barrier
    pthread_barrierattr_init(&barrier_attr);

    pthread_barrier_init(&my_barrier, &barrier_attr, NTASKS);

    //pthread_barrierattr_destroy(&barrier_attr);

    for (x=0; x<NTASKS; x++) {
      pthread_attr_init(&attrib);
      pthread_create(&(tasks[x]), &attrib , fun, (void *)x);
    }

    return 0;
}


void cleanup_module(void)
{
    int x;
    for (int x=0; x<NTASKS; x++){
        pthread_cancel(tasks[x]);
        pthread_join(tasks[x],NULL);
    }
    pthread_barrier_destroy(&my_barrier);
}
```

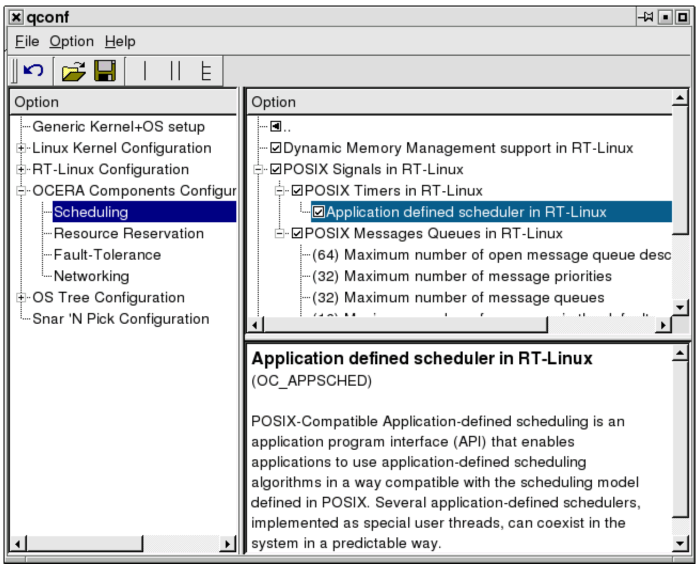
6.8. Application-Defined Scheduler

6.8.1. Description

POSIX-Compatible Application-defined scheduling (ADS) is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way. This way, users can implement their own scheduling algorithms that can be ported inmediately to other POSIX compliant RTOS.

6.8.2. Usage

This facility depends on POSIX signals and POSIX Timers, so you need to select them in order to enable the ADS selection box (Figure 6).



Generic Kernel+OS setup
Linux Kernel Configuration
RT-Linux Configuration
OCERA Components Configur
Scheduling
Resource Reservation
Fault-Tolerance
Networking
OS Tree Configuration
Snar 'N Pick Configuration

..
☑Dynamic Memory Management support in RT-Linux
☑POSIX Signals in RT-Linux
☑POSIX Timers in RT-Linux
☑Application defined scheduler in RT-Linux
☑POSIX Messages Queues in RT-Linux
(64) Maximum number of open message queue desc
(32) Maximum number of message priorities
(32) Maximum number of message queues

Application defined scheduler in RT-Linux
(OC_APPSCHED)

POSIX-Compatible Application-defined scheduling is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way.

Once the sources have been compiled you can create the sources of your scheduling algorithm. This sources will be compiled as a separate kernel module.

6.8.3. Programming interface (API)

The application defined scheduler facility API is a little more complex than "normal" operating systems services like file management since the ADS has to provide two different API's. One API for the application scheduler thread and another API for the application scheduled thread. ADS API has been designed to be included in the POSIX standard. Following is the list of functions that can be used by scheduler threads:

```
Program scheduling actions ( suspending or activating threads)
int posix_appsched_actions_addactivate (posix_appsched_actions_t * sched_actions, pthread_t  thread )
int posix_appsched_actions_addsuspend (posix_appsched_actions_t * sched_actions, pthread_t  thread )
int posix_appsched_actions_addlock (posix_appsched_actions_t * sched_actions, pthread_t  thread,const pthre
Execute Scheduling Actions
int  posix_appsched_execute_actions (const posix_appsched_actions_t * sched_actions,  const sigset_t * set,
Getting and setting application scheduled thread's data
int  pthread_remote_setspecific (pthread_key_t  key, pthread_t  th, void * value)
void * pthread_remote_getspecific  (pthread_key_t  key, pthread_t  th)

Set and get mutex-specific data
int  posix_appsched_mutex_setspecific( pthread_mutex_t * mutex, void * value)
int  posix_appsched_mutex_getspecific (const pthread_mutex_t * mutex, void ** data)
Scheduling events sets manipulation
int  posix_appsched_emptyset (posix_appsched_eventset_t * set, int posix_appsched_fillset  posix_appsched_e
int  posix_appsched_addset(posix_appsched_eventset_t * set, int  appsched_event)
int  posix_appsched_delset( posix_appsched_eventset_t * set ,int  appsched_event)
int  posix_appsched_ismember(const posix_appsched_eventset_t * set, int appsched_event)
int  posix_appsched_seteventmask (const posix_appsched_eventset_t * set, int  posix_appsched_geteventmask,
       posix_appsched_eventset_t * set )
While in the application scheduled thread's side the API is:
Explicit scheduler invocation
int  posix_appsched_invoke_scheduler(void * msg, size_t  msg_size)
Manipulate application scheduled threads attributes
int  pthread_attr_setthread_type (pthread_attr_t * attr, int type, int  pthread_attr_setappscheduler, pthre
int  pthread_attr_setappsched_param(pthread_attr_t * attr, void * param, int  size)
int  pthread_attr_getappscheduler (pthread_attr_t * attr, pthread_t  sched)
int  pthread_getappsched_param (pthread_attr_t * attr, pthread_t * sched, void * param, int *  size)
Application-defined Mutex Protocol
int  pthread_mutexattr_setappscheduler (pthread_mutexattr_t * attr, struct rtl_thread_struct * appscheduler
int  pthread_mutexattr_getappscheduler  (const pthread_mutexattr_t * attr,  struct rtl_thread_struct * apps
int  pthread_mutexattr_setappschedparam (pthread_mutexattr_t * attr, const struct pthread_mutex_schedparam 
int  pthread_mutexattr_getappschedparam  (const pthread_mutexattr_t * attr, struct pthread_mutex_schedparam
int  pthread_mutex_setappschedparam (pthread_mutex_t * mutex, const struct pthread_mutex_schedparam * sched
int  pthread_mutex_getappschedparam (const pthread_mutex_t * mutex, struct pthread_mutex_schedparam * sched
```

6.8.4. Example

This example creates a scheduler thread and two scheduled threads. The scheduler thread controls the execution of its scheduled threads following a Earliest Deadline First priority assignation. That is, in this example it is implemented the EDF scheduling algorithm. The scheduled threads are periodic with deadline equal to period. For each scheduled thread a periodic timer is programmed which spires each time the release time is reached. Threads are created in the file edf_threads.c. This is the source that will be compiled and inserted as a module. The algorithm is implemented in the files edf_sched.c and edf_sched.h.

```
/* edf_sched.h*/

#include "../misc/compat.h"
#include <rtl_debug.h>
#include <time.h>

struct edf_sched_param {
  struct timespec period;
};

#define ERROR(s) {perror (s); rtl_printf("\n");  exit (-1);}
//#define ERROR(s) {perror (s); set_break_point_here; exit (-1);}

void *edf_scheduler (void *arg);
```

```
#define MAX_TASKS 10
extern timer_t timer_ids[MAX_TASKS];
extern pthread_t tasks[MAX_TASKS];

extern long loops_per_second ;

/*
 * eat
 *
 * Executes during the interval of time 'For_Seconds'
 */
extern inline void eat (float for_seconds)
{
    long num_loop = (long)(loops_per_second * (float)for_seconds);
    long j = 1;
    long i;

    for (i=1; i<=num_loop; i++) {
        j++;
        if (j<i) {
            j = i-j;
        } else {
            j = j-1;
        }
    }
}

extern inline long subtract (struct timespec *a, struct timespec *b)
{
    long result, nanos;

    result = (a->tv_sec  - b->tv_sec)*1000000;
    nanos  = (a->tv_nsec - b->tv_nsec)/1000;
    return (result+nanos);
}


/*
 * adjust
 *
 * Measures the CPU speed (to be called before any call to 'eat')
 */
extern inline void adjust (void)
{
    struct timespec initial_time, final_time;
    long interval;
    int number_of_tries =0;
    long adjust_time = 1000000;
    int max_tries = 6;

    do {
        clock_gettime (CLOCK_REALTIME, &initial_time);
        eat(((float)adjust_time)/1000000.0);
        clock_gettime (CLOCK_REALTIME, &final_time);
        interval = subtract(&final_time,&initial_time);
        loops_per_second = (long)(
              (float)loops_per_second*(float)adjust_time/(float)interval);
        number_of_tries++;
    } while (number_of_tries<=max_tries &&
      labs(interval-adjust_time)>=adjust_time/50);
}



/*edf_sched.c*/

#include "edf_sched.h"
#include "../misc/timespec_operations.h"
#include "../misc/generic_lists.h"
#include "../misc/generic_lists_order.h"

typedef enum {ACTIVE, BLOCKED, TIMED} th_state_t;

/* Thread-specific data */
typedef struct thread_data {
  struct thread_data * next;
  th_state_t th_state;
  struct timespec period;
  struct timespec next_deadline; /* absolute time */
  int id;
  timer_t timer_id;
  pthread_t thread_id;
} thread_data_t;

thread_data_t th_data[MAX_TASKS];
#define free(ptr) do {} while(0)
/* Scheduling algorithm data */
list_t RQ = NULL;
int threads_count = 0; // to assign a different id to each thread
thread_data_t *current_thread = NULL; // thread currently chosen to execute
pthread_key_t edf_key=0;


/*
 * more_urgent_than
 */
int more_urgent_than (void *left, void *right)
{
  return smaller_timespec (&((thread_data_t *)left)->next_deadline,
    &((thread_data_t *)right)->next_deadline);
}

/*
 * schedule_next
 */
void schedule_next (posix_appsched_actions_t *actions)
{
  thread_data_t *most_urgent_thread = head (RQ);

  if (most_urgent_thread != current_thread) {

    if (most_urgent_thread != NULL) {
      // Activate next thread
      printf (" Activate:%d ptr:%d\n", most_urgent_thread->id,most_urgent_thread->thread_id);
      if (posix_appsched_actions_addactivate (actions,
       most_urgent_thread->thread_id))
 ERROR ("posix_appsched_actions_addactivate");
    }

    if (current_thread != NULL && current_thread->th_state != BLOCKED) {
      // Suspend "old" current thread
      printf (" Suspend:%d ptr:%d\n", current_thread->id,current_thread->thread_id);
      if (posix_appsched_actions_addsuspend (actions,
      current_thread->thread_id))
 perror ("posix_appsched_actions_addsuspend");
    }

    current_thread = most_urgent_thread;
  }
}

/*
 * add_to_list_of_threads
 */
void add_to_list_of_threads (pthread_t thread_id,
      const struct timespec *now)
{
  struct edf_sched_param param;
  thread_data_t *t_data;
  struct itimerspec timer_prog;

  if (pthread_getappschedparam (thread_id,(void *)&param,NULL))
    ERROR ("pthread_getschedparam");
  t_data = &th_data[threads_count];
  t_data->period = param.period;
  t_data->th_state = ACTIVE;
  t_data->id = threads_count++;
  add_timespec (&t_data->next_deadline, now, &t_data->period);
  t_data->thread_id = thread_id;
  t_data->timer_id =timer_ids[t_data->id];

  // Add to ready queue
  enqueue_in_order (t_data, &RQ, more_urgent_than);
  // Assign thread-specific data
  if (pthread_remote_setspecific (edf_key, thread_id, t_data))
    ERROR ("pthread_remote_setspecific");

  // Program periodic timer (period = t_data->period)
  timer_prog.it_value = t_data->next_deadline;
  timer_prog.it_interval = t_data->period;
  if (timer_settime (t_data->timer_id, TIMER_ABSTIME, &timer_prog, NULL))
    ERROR ("timer_settime");

  printf (" Add new thread:%d, period:%ds%dns\n", t_data->id,
   t_data->period.tv_sec, t_data->period.tv_nsec);
}

/*
 * eliminate_from_list_of_threads
 */
void eliminate_from_list_of_threads (pthread_t thread_id)
{
  thread_data_t *t_data;
  struct itimerspec null_ts={{0, 0},{0, 0}};
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");
  // disarm timer.
  timer_settime(t_data->timer_id,0,&null_ts,NULL);

  // Remove from scheduling algorithm lists
  if (t_data->th_state == ACTIVE)
    dequeue (t_data, &RQ);
  // Free used memory
  free (t_data);
}

/*
 * make_ready
 */
void make_ready (pthread_t thread_id, const struct timespec *now)
{
  thread_data_t *t_data;
  struct itimerspec timer_prog;
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");

  t_data->th_state = ACTIVE;

  add_timespec (&t_data->next_deadline, now, &t_data->period);

  // Program periodic timer
  timer_prog.it_value = t_data->next_deadline;
  timer_prog.it_interval = t_data->period;
  timer_settime (t_data->timer_id, TIMER_ABSTIME, &timer_prog, NULL);
}

/*
 * make_blocked
 */
void make_blocked (pthread_t thread_id)
{
  thread_data_t *t_data;
  struct itimerspec null_timer_prog = {{0, 0},{0, 0}};
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");

  t_data->th_state = BLOCKED;
  timer_settime (t_data->timer_id, 0, &null_timer_prog, NULL);
}

/*
 * reached_activation_time
 */
void reached_activation_time (thread_data_t *t_data)
{
  switch (t_data->th_state) {
  case TIMED:
    t_data->th_state = ACTIVE;
    enqueue_in_order (t_data, &RQ, more_urgent_than);
    incr_timespec (&t_data->next_deadline, &t_data->period);
    break;
  case BLOCKED:
    break;
  case ACTIVE:
    // Deadline missed
    printf (" Deadline missed in thread:%d !!\n", t_data->id);
    incr_timespec (&t_data->next_deadline, &t_data->period);
    break;
  default:
    printf (" Invalid state:%d in thread:%d !!\n", t_data->th_state, t_data->id);
  }

  // This is only, for debbuging purposes in RTLinux.
  rt_print_edf_request(events,t_data,FIFO);
}

/*
 * make_timed
 */
void make_timed (pthread_t thread_id)
{
  thread_data_t *t_data;
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");

  t_data->th_state = TIMED;

  // remove the thread from the ready queue
  dequeue (t_data, &RQ);
}

/*
 * EDF scheduler thread
 */
void *edf_scheduler (void *arg)
{
  posix_appsched_actions_t actions;
  struct posix_appsched_event event;
  sigset_t waited_signal_set;
  struct timespec now;
  int i;

  // Initialize the ’waited_signal_set’
  sigemptyset (&waited_signal_set);
  for (i=0;i<MAX_TASKS;i++)
    sigaddset (&waited_signal_set, (SIGUSR1+i));

  // Create a thread-specific data key
  if (pthread_key_create (&edf_key, NULL))
    ERROR ("pthread_create_key");

  // Initialize actions object
  if (posix_appsched_actions_init (&actions))
    ERROR ("posix_appsched_actions_init");

  while (1) {
    /* Actions of activation and suspension of threads */
    schedule_next (&actions);

    /* Execute scheduling actions */
    if (posix_appsched_execute_actions (&actions,&waited_signal_set,
 NULL, &now, &event))
 ERROR ("posix_appsched_execute_actions");

    /* Initialize actions object */
    if (posix_appsched_actions_destroy (&actions))
      ERROR ("posix_appsched_actions_destroy");
    if (posix_appsched_actions_init (&actions))
      ERROR ("posix_appsched_actions_init");

      /* Process scheduling events */
    printf ("\nEvent: %d\n", event.event_code);
    switch (event.event_code) {
    case POSIX_APPSCHED_NEW:
      add_to_list_of_threads (event.thread, &now);
      break;

    case POSIX_APPSCHED_TERMINATE:
      eliminate_from_list_of_threads (event.thread);
      break;

    case POSIX_APPSCHED_READY:
      make_ready (event.thread, &now);
      break;

    case POSIX_APPSCHED_BLOCK:
      make_blocked (event.thread);
      break;

    case POSIX_APPSCHED_EXPLICIT_CALL:
      rtl_printf("EXPLICIT_CALL: %d ptr:%d\n",event.thread->user[0]-2,event.thread);
      // The thread has done all its work for the present activation
      make_timed (event.thread);
      break;

    case POSIX_APPSCHED_SIGNAL:
      rtl_printf("SIGNAL %d\n",event.event_info.siginfo.si_signo-SIGUSR1);
      // This is a trick, since in RTLinux we don’t have REAL TIME SIGNALS, yet.
      reached_activation_time(&th_data[event.event_info.siginfo.si_signo-SIGUSR1]);
    }
  }

  return NULL;
}



/*edf_threads.c*/

#include "edf_sched.h"
#include <pthread.h>

#define NTASKS 2
timer_t timer_ids[MAX_TASKS];
pthread_t sched, tasks[MAX_TASKS];
#define MAIN_PRIO MAX_TASKS

long loops_per_second = 30000;

/*  Scheduled thread */
void * periodic (void * arg)
{
  float amount_of_work = *(float *) arg;
  int count=0;

  posix_appsched_invoke_scheduler (NULL, 0);
  while (count++<10000) {
    /* do useful work */
    rtl_printf("I am here id:%d, iter:%d\n",pthread_self()->user[0]-2,count);
    eat (amount_of_work);

    rtl_printf("th :%d about to invoke_scheduler\n",pthread_self()->user[0]-2,count);
    posix_appsched_invoke_scheduler (NULL, 0);
  }
}



int init_module(void)
{
  pthread_attr_t attr;
  struct edf_sched_param user_param;
  struct sched_param param;
  float load1, load2;
  struct sigevent evp;
  int ret=0;

  adjust ();
```

```
/* Creation of the scheduler thread */
pthread_attr_init (&attr);
param.sched_priority = MAIN_PRIO - 1;
if ((ret=pthread_attr_setappschedulerstate(&attr,PTHREAD_APPSCHEDULER))<0)
  printk("error while pthread_attr_setappschedulerstate(&attr,PTHREAD_APPSCHEDULER)\n");
if (pthread_attr_setschedparam (&attr, &param))
  ERROR ("pthread_attr_setschedparam scheduler");
if (pthread_create (&sched, &attr, edf_scheduler, NULL))
  ERROR ("pthread_create scheduler");

/* Set main task base priority */
param.sched_priority = MAIN_PRIO;
if (pthread_setschedparam (sched, SCHED_FIFO, &param))
  perror ("pthread_setschedparam");


pthread_attr_destroy(&attr);

/* Creation of one scheduled thread */
pthread_attr_init (&attr);
attr.initial_state=0;
pthread_attr_setfp_np(&attr, 1);
param.sched_priority = MAIN_PRIO - 3;
user_param.period.tv_sec  = 0;
user_param.period.tv_nsec = 20*1000*1000; // period = 20 ms
load1 = 0.001; // load = 1 ms


/*
  param.posix_appscheduler = sched;
  param.posix_appsched_param = (void *) &user_param;
  param.posix_appsched_paramsize = sizeof (struct edf_sched_param);
*/
evp.sigev_notify          = SIGEV_SIGNAL;
evp.sigev_signo           = SIGUSR1;
if (timer_create (CLOCK_REALTIME, &evp,&timer_ids[evp.sigev_signo-SIGUSR1]))
  ERROR ("timer_create");

if ((ret=pthread_attr_setappschedulerstate(&attr,PTHREAD_REGULAR))<0)
  printk("error while pthread_attr_setappschedulerstate\n");

if ((ret=pthread_attr_setappschedparam(&attr,(void *) &user_param,sizeof(user_param))<0))
  printk("error while pthread_attr_setappschedparam\n");

if (pthread_attr_setappscheduler (&attr, sched))
  ERROR ("pthread_attr_setappscheduler 1");


if (pthread_attr_setschedparam (&attr, &param))
  ERROR ("pthread_attr_setschedparam 1");

if (pthread_create (&tasks[0], &attr, periodic, &load1))
  ERROR ("pthread_create 1");

/* Creation of other scheduled thread */
pthread_attr_init (&attr);
attr.initial_state=0;
pthread_attr_setfp_np(&attr, 1);
param.sched_priority = MAIN_PRIO - 1;
user_param.period.tv_sec  = 0;
user_param.period.tv_nsec = 50*1000*1000;// period = 50 ms
load2 = 0.005; // load = 5 ms
/*
  param.posix_appsched_param = (void *) &user_param;
  param.posix_appsched_paramsize = sizeof (struct edf_sched_param);
*/
evp.sigev_notify          = SIGEV_SIGNAL;
evp.sigev_signo           = SIGUSR1+1;
if (timer_create (CLOCK_REALTIME, &evp,&timer_ids[evp.sigev_signo-SIGUSR1]))
  ERROR ("timer_create");
if ((ret=pthread_attr_setappschedulerstate(&attr,PTHREAD_REGULAR))<0)
  printk("error while pthread_attr_setappschedulerstate\n");

if ((ret=pthread_attr_setappschedparam(&attr,(void *) &user_param,sizeof(user_param))<0))
  ERROR ("pthread_attr_setappschedparam 2");

if (pthread_attr_setappscheduler (&attr, sched))
```

```
    ERROR ("pthread_attr_setappscheduler 2");

  if (pthread_attr_setschedparam (&attr, &param))
    ERROR ("pthread_attr_setschedparam 2");

  if (pthread_create (&tasks[1], &attr, periodic, &load2))
    ERROR ("pthread_create 2");

  return 0;
}

void cleanup_module(void){
  int i;
  // Remove scheduled threads.
  for (i=0;i<NTASKS;i++){
    timer_delete(timer_ids[i]);
    pthread_delete_np(tasks[i]);
  }
  // Remove Application scheduler thread.
  pthread_delete_np(sched);


}
```

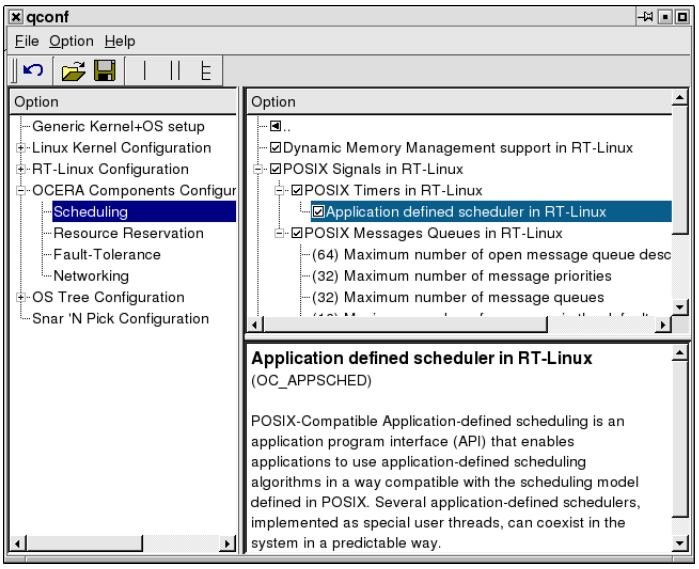
6.9. Application-Defined Scheduler

6.9.1. Description

POSIX-Compatible Application-defined scheduling (ADS) is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way. This way, users can implement their own scheduling algorithms that can be ported inmediately to other POSIX compliant RTOS.

6.9.2. Usage

This facility depends on POSIX signals and POSIX Timers, so you need to select them in order to enable the ADS selection box (Figure 6).



Generic Kernel+OS setup
Linux Kernel Configuration
RT-Linux Configuration
OCERA Components Configur
Scheduling
Resource Reservation
Fault-Tolerance
Networking
OS Tree Configuration
Snar 'N Pick Configuration

..
☑Dynamic Memory Management support in RT-Linux
☑POSIX Signals in RT-Linux
☑POSIX Timers in RT-Linux
☑Application defined scheduler in RT-Linux
☑POSIX Messages Queues in RT-Linux
(64) Maximum number of open message queue desc
(32) Maximum number of message priorities
(32) Maximum number of message queues

Application defined scheduler in RT-Linux
(OC_APPSCHED)

POSIX-Compatible Application-defined scheduling is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way.

Once the sources have been compiled you can create the sources of your scheduling algorithm. This sources will be compiled as a separate kernel module.

6.9.3. Programming interface (API)

The application defined scheduler facility API is a little more complex than "normal" operating systems services like file management since the ADS has to provide two different API's. One API for the application scheduler thread and another API for the application scheduled thread. ADS API has been designed to be included in the POSIX standard. Following is the list of functions that can be used by scheduler threads:

```
Program scheduling actions ( suspending or activating threads)
int posix_appsched_actions_addactivate (posix_appsched_actions_t * sched_actions, pthread_t  thread )
int posix_appsched_actions_addsuspend (posix_appsched_actions_t * sched_actions, pthread_t  thread )
int posix_appsched_actions_addlock (posix_appsched_actions_t * sched_actions, pthread_t  thread,const pthre
Execute Scheduling Actions
int  posix_appsched_execute_actions (const posix_appsched_actions_t * sched_actions,  const sigset_t * set,
Getting and setting application scheduled thread's data
int  pthread_remote_setspecific (pthread_key_t  key, pthread_t  th, void * value)
void * pthread_remote_getspecific  (pthread_key_t  key, pthread_t  th)

Set and get mutex-specific data
int  posix_appsched_mutex_setspecific( pthread_mutex_t * mutex, void * value)
int  posix_appsched_mutex_getspecific (const pthread_mutex_t * mutex, void ** data)
Scheduling events sets manipulation
int  posix_appsched_emptyset (posix_appsched_eventset_t * set, int posix_appsched_fillset  posix_appsched_e
int  posix_appsched_addset(posix_appsched_eventset_t * set, int  appsched_event)
int  posix_appsched_delset( posix_appsched_eventset_t * set ,int  appsched_event)
int  posix_appsched_ismember(const posix_appsched_eventset_t * set, int appsched_event)
int  posix_appsched_seteventmask (const posix_appsched_eventset_t * set, int  posix_appsched_geteventmask,
       posix_appsched_eventset_t * set )
While in the application scheduled thread's side the API is:
Explicit scheduler invocation
int  posix_appsched_invoke_scheduler(void * msg, size_t  msg_size)
Manipulate application scheduled threads attributes
int  pthread_attr_setthread_type (pthread_attr_t * attr, int type, int  pthread_attr_setappscheduler, pthre
int  pthread_attr_setappsched_param(pthread_attr_t * attr, void * param, int  size)
int  pthread_attr_getappscheduler (pthread_attr_t * attr, pthread_t  sched)
int  pthread_getappsched_param (pthread_attr_t * attr, pthread_t * sched, void * param, int *  size)
Application-defined Mutex Protocol
int  pthread_mutexattr_setappscheduler (pthread_mutexattr_t * attr, struct rtl_thread_struct * appscheduler
int  pthread_mutexattr_getappscheduler  (const pthread_mutexattr_t * attr,  struct rtl_thread_struct * apps
int  pthread_mutexattr_setappschedparam (pthread_mutexattr_t * attr, const struct pthread_mutex_schedparam
int  pthread_mutexattr_getappschedparam  (const pthread_mutexattr_t * attr, struct pthread_mutex_schedparam
int  pthread_mutex_setappschedparam (pthread_mutex_t * mutex, const struct pthread_mutex_schedparam * sched
int  pthread_mutex_getappschedparam (const pthread_mutex_t * mutex, struct pthread_mutex_schedparam * sched
```

6.9.4. Example

This example creates a scheduler thread and two scheduled threads. The scheduler thread controls the execution of its scheduled threads following a Earliest Deadline First priority assignation. That is, in this example it is implemented the EDF scheduling algorithm. The scheduled threads are periodic with deadline equal to period. For each scheduled thread a periodic timer is programmed which spires each time the release time is reached. Threads are created in the file edf_threads.c. This is the source that will be compiled and inserted as a module. The algorithm is implemented in the files edf_sched.c and edf_sched.h.

```
/* edf_sched.h*/

#include "../misc/compat.h"
#include <rtl_debug.h>
#include <time.h>

struct edf_sched_param {
  struct timespec period;
};

#define ERROR(s) {perror (s); rtl_printf("\n");  exit (-1);}
//#define ERROR(s) {perror (s); set_break_point_here; exit (-1);}

void *edf_scheduler (void *arg);
```

```
#define MAX_TASKS 10
extern timer_t timer_ids[MAX_TASKS];
extern pthread_t tasks[MAX_TASKS];

extern long loops_per_second ;

/*
 * eat
 *
 * Executes during the interval of time 'For_Seconds'
 */
extern inline void eat (float for_seconds)
{
    long num_loop = (long)(loops_per_second * (float)for_seconds);
    long j = 1;
    long i;

    for (i=1; i<=num_loop; i++) {
        j++;
        if (j<i) {
            j = i-j;
        } else {
            j = j-1;
        }
    }
}

extern inline long subtract (struct timespec *a, struct timespec *b)
{
    long result, nanos;

    result = (a->tv_sec  - b->tv_sec)*1000000;
    nanos  = (a->tv_nsec - b->tv_nsec)/1000;
    return (result+nanos);
}


/*
 * adjust
 *
 * Measures the CPU speed (to be called before any call to 'eat')
 */
extern inline void adjust (void)
{
    struct timespec initial_time, final_time;
    long interval;
    int number_of_tries =0;
    long adjust_time = 1000000;
    int max_tries = 6;

    do {
        clock_gettime (CLOCK_REALTIME, &initial_time);
        eat(((float)adjust_time)/1000000.0);
        clock_gettime (CLOCK_REALTIME, &final_time);
        interval = subtract(&final_time,&initial_time);
        loops_per_second = (long)(
              (float)loops_per_second*(float)adjust_time/(float)interval);
        number_of_tries++;
    } while (number_of_tries<=max_tries &&
      labs(interval-adjust_time)>=adjust_time/50);
}



/*edf_sched.c*/

#include "edf_sched.h"
#include "../misc/timespec_operations.h"
#include "../misc/generic_lists.h"
#include "../misc/generic_lists_order.h"

typedef enum {ACTIVE, BLOCKED, TIMED} th_state_t;

/* Thread-specific data */
typedef struct thread_data {
  struct thread_data * next;
  th_state_t th_state;
  struct timespec period;
  struct timespec next_deadline; /* absolute time */
  int id;
  timer_t timer_id;
  pthread_t thread_id;
} thread_data_t;

thread_data_t th_data[MAX_TASKS];
#define free(ptr) do {} while(0)
/* Scheduling algorithm data */
list_t RQ = NULL;
int threads_count = 0; // to assign a different id to each thread
thread_data_t *current_thread = NULL; // thread currently chosen to execute
pthread_key_t edf_key=0;


/*
 * more_urgent_than
 */
int more_urgent_than (void *left, void *right)
{
  return smaller_timespec (&((thread_data_t *)left)->next_deadline,
    &((thread_data_t *)right)->next_deadline);
}

/*
 * schedule_next
 */
void schedule_next (posix_appsched_actions_t *actions)
{
  thread_data_t *most_urgent_thread = head (RQ);

  if (most_urgent_thread != current_thread) {

    if (most_urgent_thread != NULL) {
      // Activate next thread
      printf (" Activate:%d ptr:%d\n", most_urgent_thread->id,most_urgent_thread->thread_id);
      if (posix_appsched_actions_addactivate (actions,
       most_urgent_thread->thread_id))
 ERROR ("posix_appsched_actions_addactivate");
    }

    if (current_thread != NULL && current_thread->th_state != BLOCKED) {
      // Suspend "old" current thread
      printf (" Suspend:%d ptr:%d\n", current_thread->id,current_thread->thread_id);
      if (posix_appsched_actions_addsuspend (actions,
      current_thread->thread_id))
 perror ("posix_appsched_actions_addsuspend");
    }

    current_thread = most_urgent_thread;
  }
}

/*
 * add_to_list_of_threads
 */
void add_to_list_of_threads (pthread_t thread_id,
      const struct timespec *now)
{
  struct edf_sched_param param;
  thread_data_t *t_data;
  struct itimerspec timer_prog;

  if (pthread_getappschedparam (thread_id,(void *)&param,NULL))
    ERROR ("pthread_getschedparam");
  t_data = &th_data[threads_count];
  t_data->period = param.period;
  t_data->th_state = ACTIVE;
  t_data->id = threads_count++;
  add_timespec (&t_data->next_deadline, now, &t_data->period);
  t_data->thread_id = thread_id;
  t_data->timer_id =timer_ids[t_data->id];

  // Add to ready queue
  enqueue_in_order (t_data, &RQ, more_urgent_than);
  // Assign thread-specific data
  if (pthread_remote_setspecific (edf_key, thread_id, t_data))
    ERROR ("pthread_remote_setspecific");

  // Program periodic timer (period = t_data->period)
  timer_prog.it_value = t_data->next_deadline;
  timer_prog.it_interval = t_data->period;
  if (timer_settime (t_data->timer_id, TIMER_ABSTIME, &timer_prog, NULL))
    ERROR ("timer_settime");

  printf (" Add new thread:%d, period:%ds%dns\n", t_data->id,
   t_data->period.tv_sec, t_data->period.tv_nsec);
}

/*
 * eliminate_from_list_of_threads
 */
void eliminate_from_list_of_threads (pthread_t thread_id)
{
  thread_data_t *t_data;
  struct itimerspec null_ts={{0, 0},{0, 0}};
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");
  // disarm timer.
  timer_settime(t_data->timer_id,0,&null_ts,NULL);

  // Remove from scheduling algorithm lists
  if (t_data->th_state == ACTIVE)
    dequeue (t_data, &RQ);
  // Free used memory
  free (t_data);
}

/*
 * make_ready
 */
void make_ready (pthread_t thread_id, const struct timespec *now)
{
  thread_data_t *t_data;
  struct itimerspec timer_prog;
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");

  t_data->th_state = ACTIVE;

  add_timespec (&t_data->next_deadline, now, &t_data->period);

  // Program periodic timer
  timer_prog.it_value = t_data->next_deadline;
  timer_prog.it_interval = t_data->period;
  timer_settime (t_data->timer_id, TIMER_ABSTIME, &timer_prog, NULL);
}

/*
 * make_blocked
 */
void make_blocked (pthread_t thread_id)
{
  thread_data_t *t_data;
  struct itimerspec null_timer_prog = {{0, 0},{0, 0}};
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");

  t_data->th_state = BLOCKED;
  timer_settime (t_data->timer_id, 0, &null_timer_prog, NULL);
}

/*
 * reached_activation_time
 */
void reached_activation_time (thread_data_t *t_data)
{
  switch (t_data->th_state) {
  case TIMED:
    t_data->th_state = ACTIVE;
    enqueue_in_order (t_data, &RQ, more_urgent_than);
    incr_timespec (&t_data->next_deadline, &t_data->period);
    break;
  case BLOCKED:
    break;
  case ACTIVE:
    // Deadline missed
    printf (" Deadline missed in thread:%d !!\n", t_data->id);
    incr_timespec (&t_data->next_deadline, &t_data->period);
    break;
  default:
    printf (" Invalid state:%d in thread:%d !!\n", t_data->th_state, t_data->id);
  }

  // This is only, for debbuging purposes in RTLinux.
  rt_print_edf_request(events,t_data,FIFO);
}

/*
 * make_timed
 */
void make_timed (pthread_t thread_id)
{
  thread_data_t *t_data;
  // get thread-specific data
  if (!(t_data = pthread_remote_getspecific (edf_key, thread_id)))
    ERROR ("pthread_remote_getspecific");

  t_data->th_state = TIMED;

  // remove the thread from the ready queue
  dequeue (t_data, &RQ);
}

/*
 * EDF scheduler thread
 */
void *edf_scheduler (void *arg)
{
  posix_appsched_actions_t actions;
  struct posix_appsched_event event;
  sigset_t waited_signal_set;
  struct timespec now;
  int i;

  // Initialize the ’waited_signal_set’
  sigemptyset (&waited_signal_set);
  for (i=0;i<MAX_TASKS;i++)
    sigaddset (&waited_signal_set, (SIGUSR1+i));

  // Create a thread-specific data key
  if (pthread_key_create (&edf_key, NULL))
    ERROR ("pthread_create_key");

  // Initialize actions object
  if (posix_appsched_actions_init (&actions))
    ERROR ("posix_appsched_actions_init");

  while (1) {
    /* Actions of activation and suspension of threads */
    schedule_next (&actions);

    /* Execute scheduling actions */
    if (posix_appsched_execute_actions (&actions,&waited_signal_set,
 NULL, &now, &event))
 ERROR ("posix_appsched_execute_actions");

    /* Initialize actions object */
    if (posix_appsched_actions_destroy (&actions))
      ERROR ("posix_appsched_actions_destroy");
    if (posix_appsched_actions_init (&actions))
      ERROR ("posix_appsched_actions_init");

      /* Process scheduling events */
    printf ("\nEvent: %d\n", event.event_code);
    switch (event.event_code) {
```

```
    case POSIX_APPSCHED_NEW:
      add_to_list_of_threads (event.thread, &now);
      break;

    case POSIX_APPSCHED_TERMINATE:
      eliminate_from_list_of_threads (event.thread);
      break;

    case POSIX_APPSCHED_READY:
      make_ready (event.thread, &now);
      break;

    case POSIX_APPSCHED_BLOCK:
      make_blocked (event.thread);
      break;

    case POSIX_APPSCHED_EXPLICIT_CALL:
      rtl_printf("EXPLICIT_CALL: %d ptr:%d\n",event.thread->user[0]-2,event.thread);
      // The thread has done all its work for the present activation
      make_timed (event.thread);
      break;

    case POSIX_APPSCHED_SIGNAL:
      rtl_printf("SIGNAL %d\n",event.event_info.siginfo.si_signo-SIGUSR1);
      // This is a trick, since in RTLinux we don’t have REAL TIME SIGNALS, yet.
      reached_activation_time(&th_data[event.event_info.siginfo.si_signo-SIGUSR1]);
    }
  }

  return NULL;
}



/*edf_threads.c*/

#include "edf_sched.h"
#include <pthread.h>

#define NTASKS 2
timer_t timer_ids[MAX_TASKS];
pthread_t sched, tasks[MAX_TASKS];
#define MAIN_PRIO MAX_TASKS

long loops_per_second = 30000;

/*  Scheduled thread */
void * periodic (void * arg)
{
  float amount_of_work = *(float *) arg;
  int count=0;

  posix_appsched_invoke_scheduler (NULL, 0);
  while (count++<10000) {
    /* do useful work */
    rtl_printf("I am here id:%d, iter:%d\n",pthread_self()->user[0]-2,count);
    eat (amount_of_work);

    rtl_printf("th :%d about to invoke_scheduler\n",pthread_self()->user[0]-2,count);
    posix_appsched_invoke_scheduler (NULL, 0);
  }
}



int init_module(void)
{
  pthread_attr_t attr;
  struct edf_sched_param user_param;
  struct sched_param param;
  float load1, load2;
  struct sigevent evp;
  int ret=0;

  adjust ();
```

```
/* Creation of the scheduler thread */
pthread_attr_init (&attr);
param.sched_priority = MAIN_PRIO - 1;
if ((ret=pthread_attr_setappschedulerstate(&attr,PTHREAD_APPSCHEDULER))<0)
  printk("error while pthread_attr_setappschedulerstate(&attr,PTHREAD_APPSCHEDULER)\n");
if (pthread_attr_setschedparam (&attr, &param))
  ERROR ("pthread_attr_setschedparam scheduler");
if (pthread_create (&sched, &attr, edf_scheduler, NULL))
  ERROR ("pthread_create scheduler");

/* Set main task base priority */
param.sched_priority = MAIN_PRIO;
if (pthread_setschedparam (sched, SCHED_FIFO, &param))
  perror ("pthread_setschedparam");


pthread_attr_destroy(&attr);

/* Creation of one scheduled thread */
pthread_attr_init (&attr);
attr.initial_state=0;
pthread_attr_setfp_np(&attr, 1);
param.sched_priority = MAIN_PRIO - 3;
user_param.period.tv_sec  = 0;
user_param.period.tv_nsec = 20*1000*1000; // period = 20 ms
load1 = 0.001; // load = 1 ms


/*
  param.posix_appscheduler = sched;
  param.posix_appsched_param = (void *) &user_param;
  param.posix_appsched_paramsize = sizeof (struct edf_sched_param);
*/
evp.sigev_notify          = SIGEV_SIGNAL;
evp.sigev_signo           = SIGUSR1;
if (timer_create (CLOCK_REALTIME, &evp,&timer_ids[evp.sigev_signo-SIGUSR1]))
  ERROR ("timer_create");

if ((ret=pthread_attr_setappschedulerstate(&attr,PTHREAD_REGULAR))<0)
  printk("error while pthread_attr_setappschedulerstate\n");

if ((ret=pthread_attr_setappschedparam(&attr,(void *) &user_param,sizeof(user_param))<0))
  printk("error while pthread_attr_setappschedparam\n");

if (pthread_attr_setappscheduler (&attr, sched))
  ERROR ("pthread_attr_setappscheduler 1");


if (pthread_attr_setschedparam (&attr, &param))
  ERROR ("pthread_attr_setschedparam 1");

if (pthread_create (&tasks[0], &attr, periodic, &load1))
  ERROR ("pthread_create 1");

/* Creation of other scheduled thread */
pthread_attr_init (&attr);
attr.initial_state=0;
pthread_attr_setfp_np(&attr, 1);
param.sched_priority = MAIN_PRIO - 1;
user_param.period.tv_sec  = 0;
user_param.period.tv_nsec = 50*1000*1000;// period = 50 ms
load2 = 0.005; // load = 5 ms
/*
  param.posix_appsched_param = (void *) &user_param;
  param.posix_appsched_paramsize = sizeof (struct edf_sched_param);
*/
evp.sigev_notify          = SIGEV_SIGNAL;
evp.sigev_signo           = SIGUSR1+1;
if (timer_create (CLOCK_REALTIME, &evp,&timer_ids[evp.sigev_signo-SIGUSR1]))
  ERROR ("timer_create");
if ((ret=pthread_attr_setappschedulerstate(&attr,PTHREAD_REGULAR))<0)
  printk("error while pthread_attr_setappschedulerstate\n");

if ((ret=pthread_attr_setappschedparam(&attr,(void *) &user_param,sizeof(user_param))<0))
  ERROR ("pthread_attr_setappschedparam 2");

if (pthread_attr_setappscheduler (&attr, sched))
```

```
    ERROR ("pthread_attr_setappscheduler 2");

  if (pthread_attr_setschedparam (&attr, &param))
    ERROR ("pthread_attr_setschedparam 2");

  if (pthread_create (&tasks[1], &attr, periodic, &load2))
    ERROR ("pthread_create 2");

  return 0;
}

void cleanup_module(void){
  int i;
  // Remove scheduled threads.
  for (i=0;i<NTASKS;i++){
    timer_delete(timer_ids[i]);
    pthread_delete_np(tasks[i]);
  }
  // Remove Application scheduler thread.
  pthread_delete_np(sched);


}
```

IV. Soft Real Time Components

Table of Contents

7. Accessing Soft Real time functionalities 55
8. Soft Real time API 57

State of this part

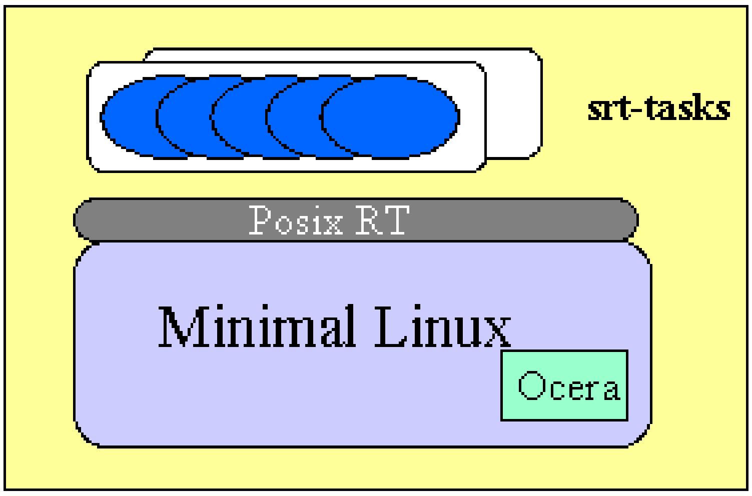
This part is under development and must be considered as a draft.
Some chapter are left blank.

Chapter 7. Accessing Soft Real time functionalities

This chapter is intended for people who need to integrate a Soft Real Time application in the OCERA framework.

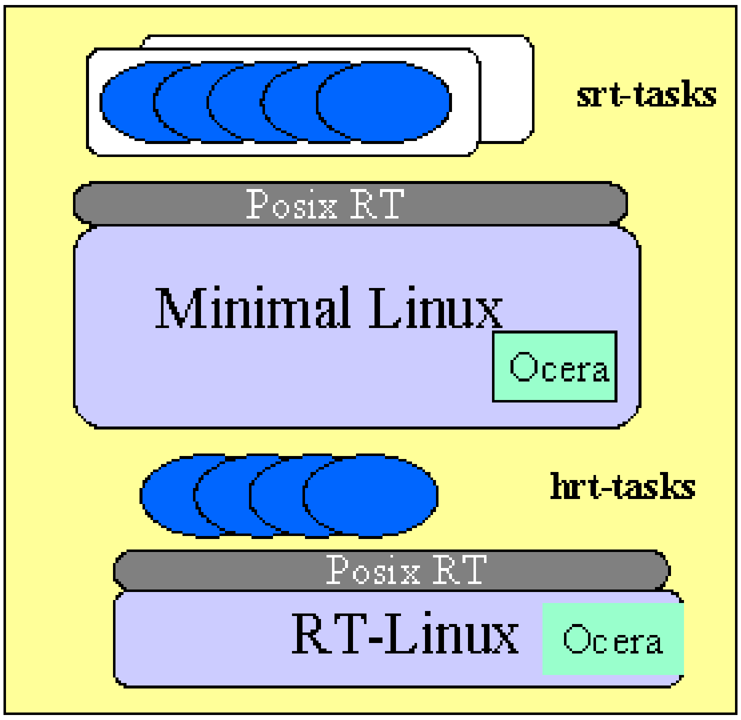
Remember that OCERA propose a Framework for embedded Real Time applications. We will see some more detailed aspects of how you can use this framework, but you will need to refer to the *Programmer's guide* if you need details on each functions.

7.1. What is an OCERA Soft Realtime application



7.2. Underlying Hard Real Time system

7.2.1. Overview of functionalities



7.2.2. Dependencies

7.3. Building an application

7.3.1. Makefile

7.3.2. Compilers

7.3.3. Libraries

7.3.4. Choosing the components

7.3.4.1. Boot

7.3.4.2. Filesystem

7.3.4.3. Scheduler

7.3.4.4. QOS

7.3.4.5. IPC

7.3.4.6. Streams

7.3.4.7. Network

7.3.4.8. RT Network

7.3.4.9. Disk access

7.3.4.10. CAN driver

7.3.4.11. Other drivers

7.4. Downloading

7.4.1. Embedded systems

7.4.2. Hosts systems

Chapter 8. Soft Real time API

This chapter is intended for people who need to integrate a Soft Real Time application in the OCERA framework.

V. Using new hardware

Table of Contents

9. Driver Framework **59**

10. Porting to new hardware **60**

State of this part

This part is under development and must be considered as a draft.

Some chapter are left blank.

Chapter 9. Driver Framework

9.1. control command with CAN

9.2. control command with CAN

9.3. real time Ethernet

9.4. two real time level application

9.5. streaming video

Chapter 10. Porting to new hardware

Even if porting to new hardware is not the goal of this document, we give here a light overview of the main drawbacks.

10.1. Supporting new processors

To support a new processor, i.e. one not supported by OCERA you will need to do a specific work depending on what already exists for this processor

10.1.1. The port of Linux and RTLinux exists

You will need to rewrite part of some of the OCERA components, like *preemption patch*, *the ADA port*, *QOS*.

10.1.2. The port of Linux exists, but not RTLinux

You will need to port RTLinux and OCERA extensions for this processor.

10.2. Supporting new Cards

Like every port to a new hardware, you will need to modify some code of the original OCERA software to achieve the work.

The first thing to do is to find the nearest architecture family , actually, OCERA support three architecture family: Intel, StrongARM and PowerPC. The different pieces of software you need to change for a specific board is referenced as the BSP (Board Support Package), examples are often provided by the hardware vendor. When you choose the family your system belongs (look in the linux source tree ./arch/), you will see if your board is already supported. then you will have to verify and certainly to change the following items from the Linux port:

- the bootstrap program (arch/xxx/boot)
- the memory management (arch/xxx/mm)
- Math emulation if any(arch/xxx/math-emu)
- May be some of the core library routines (arch/xxx/lib/)
- Interrupt processing and PIC programming(arch/xxx/kernel/)
- drivers for the host bus bridges of the board (drivers/pci)
- drivers for the new devices (drivers/xxx)

Of course some of them may be available and used as is.

When supporting new cards for which processor the port of OCERA has been done, you will need to rewrite some of the major components of RTLinux and/or Linux. In this document we will assume that the Linux port have already been done. And we will concentrate on the RTLinux and OCERA component port for which you will need to rewrite the following things:

- the linux patch (main/xxx/arch.h)
- May be some of the core library routines (main/xxx/arch.h)
- Interrupt processing and PIC programming(main/xxx/arch.h)
- drivers for the new devices (drivers/xxx)

10.2.1. Interrupt processing

May be the main work will be here: writing new IRQ routine for the new processor and the new PIC. You will also need to change the way RTLinux dynamically patches the kernel on start. Take a look at the *main/arch* subdirectory of the RTLinux tree.

10.2.2. patch process

Manufacturer's BSP will of course help you rewriting interrupts control routine, and you will need it to rewrite major components of Linux and/or RTLinux if the port has not been completed for you board.

In this case this mean you will have to rewrite memory management or IO programming for Linux, RTLinux and OCERA components, and this may be a hard task.

VI. Man pages

Table of Contents

11. Manuel pages for components 63
12. Manuel pages RTLinux...64
ANNEX 1: Component Manages 65

State of this part

This part is under development and must be considered as a draft.
Some chapter are left blank.
A lot of work(quite all) has be done by J. Vidal from UPV.

Chapter 11. Manuel pages for components

Chapter 12. Manuel pages RTLinux

ANNEX 1: Component Manages

pthread_getappschedparam

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`pthread_mutexattr_setappscheduler` , `pthread_mutexattr_getappscheduler` , `pthread_mutexattr_setappschedparam` , `pthread_mutexattr_getappschedparam` , `pthread_mutex_setappschedparam` , `pthread_mutex_getappschedparam` , `posix_appsched_mutex_setspecific`, `posix_appsched_mutex_getspecific` — Application-scheduled mutexes.

Synopsis

```
#include <pthread.h>
```

DESCRIPTION

The POSIX-Compatible Application-defined scheduling API allows creating mutexes whose synchronization protocol is defined by the application scheduler. To allow this functionality a set of functions similar to those used with regular threads is available. These special mutexes are created like any other POSIX mutex but specifying the value PTHREAD_APPSCHED_PROTOCOL for their protocol attribute. For this kind of mutexes two new attributes has been added: the appscheduler attribute and the appschedparam attribute. The appscheduler attribute identifies the scheduler thread the mutex is attached to. The optional appschedparam attribute can be used for passing application-defined mutex scheduling attributes to the scheduler.

As for the application-scheduled threads, it is also important for the scheduler to have simple mechanism to attach and retrieve the scheduling specific data associated with an application-scheduled mutex. With this purpose the interface introduces a new functionality not defined in POSIX: the mutex-specific data, and two functions to get and set the value currently bound to a mutex.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`posix_appsched_execute_actions`

posix_appsched_actions_addactivate

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`posix_appsched_actions_addactivate`, `posix_appsched_actions_addsuspend`, `posix_appsched_actions_addlock` — Program scheduling actions.

Synopsis

```
#include <pthread.h>
int posix_appsched_actions_addactivate(posix_appsched_actions_t *sched_actions , pthread_t
 thread );
int posix_appsched_actions_addsuspend(posix_appsched_actions_t *sched_actions , pthread_t
 thread );
int posix_appsched_actions_addlock(posix_appsched_actions_t *sched_actions , pthread_t
 thread , pthread_mutex_t  *mutex );
```

DESCRIPTION

These functions allows applications schedulers to program scheduling actions by adding them to the actions queue referenced by locator actions. These actions will be executed when function `posix_appsched_execute_actions` is called. The actions that can be added are the following:

- Activate or suspend an application scheduled thread with `posix_appsched_actions_addactivate` and `posix_appsched_actions_addsuspend` respectively.
- Grant the lock of an application-scheduled mutex with `posix_appsched_actions_addlock` .

RETURN VALUE

If the call succeeds this functions shall return 0 and add a new action to the actions queue referenced by location actions.

ERRORS

This functions returns 0 after successfully completion or a value distinct of 0 if the actions queue is exhausted.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`posix_appsched_execute_actions`

posix_appsched_execute_actions

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`posix_appsched_execute_actions` — Execute scheduling actions.

Synopsis

```
#include <pthread.h>
int posix_appsched_execute_actions(const posix_appsched_actions_t *sched_actions , const
sigset_t *set , const struct timespec *timeout , struct timespec *current_time, struct
posix_appsched_event *event );
```

DESCRIPTION

Allows the applications scheduler to execute a list of scheduling actions and the it suspends waiting for the next scheduling event. If desired, a timeout can be set as an additional return condition which will occur when there is no scheduling event available but the timeout expires. The system time measured immediately before the function returns will be placed in current_time parameter, if no NULL. In addition to previously described return conditions, `posix_appsched_execute_actions` can be programmed to return when a POSIX signal is generated for the thread. This possibility eases the use of POSIX timers.

RETURN VALUE

If the call succeeds this functions shall return 0 and update the location referenced by event with next event to be processed. If there are no more events to process the NULL event is returned (event_code = -1).

ERRORS

This functions returns 0 after successfully completion. Otherwise returns:

- [EPERM]: The scheduled thread is trying to lock a no valid mutex for this tread.
- [EBUSY]: The scheduled thread is trying to lock a mutex that is already owned.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`posix_appsched_actions_addactivate`
`posix_appsched_explicit_call`

posix_appsched_invoke_scheduler

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`posix_appsched_invoke_scheduler` — Application scheduler invocation.

Synopsis

```
#include <pthread.h>
int posix_appsched_invoke_scheduler(char *msg , size_t  size);
```

DESCRIPTION

This function allows applications scheduled threads to explicitly call its scheduler thread. This could be necessary in some scheduling algorithms, for example as a mechanism to inform the scheduler a thread has finished its work for the current activation. Calling this function will cause a scheduling event of type POSIX_APPSCHED_EXPLICIT_CALL to be generated. Right now, no message can be attached to the event (this function is expected to change in new API release).

RETURN VALUE

If the call succeeds this functions shall return 0 and add a new event shall be generated.

ERRORS

If this function is called from a non regular thread (those scheduled by an application scheduler)or alternatively the scheduler thread event's queue is full, no event is generated and -1 is returned. With o without error the system scheduler always is called.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`posix_appsched_execute_actions`

pthread_appschedattr_seteventmask

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`pthread_appschedattr_seteventmask` , `pthread_appschedattr_geteventmask` , `posix_appsched_emptyset` , `posix_appsched_fillset` , `posix_appsched_addset` , `posix_appsched_delset` , `posix_appsched_ismember` — Scheduling events sets manipulation

Synopsis

```
#include <pthread.h>
int  pthread_appschedattr_seteventmask (posix_appsched_eventset_t *set);
int  pthread_appschedattr_geteventmask (posix_appsched_eventset_t *set);
int  posix_appsched_emptyset (posix_appsched_eventset_t *set);
int  posix_appsched_fillset (posix_appsched_eventset_t *set);
int  posix_appsched_addset (posix_appsched_eventset_t *set, int event);
int  posix_appsched_delset (posix_appsched_eventset_t *set, int event);
```

DESCRIPTION

In some scheduling algorithms could be interesting to discard all events but the relevant for the algorithm. This can be achieved thorough the use of events masks to filter some events in a way similar to posix signals mask manipulation. In to other things these functions allows you to set the mask of filtered events, get the filtered events of an application-defined scheduler thread, and manipulate the filtered events mask (add/remove events to be filtered, ask if an event is filtered, etc...).

ERRORS

No errors are defined.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`posix_appsched_execute_actions`

pthread_attr_setappscheduler

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`pthread_attr_setappscheduler, pthread_attr_getappscheduler` — Set/get application-scheduled thread scheduler.

Synopsis

```
#include <pthread.h>
int pthread_attr_setappscheduler(pthread_attr_t attr , pthread_t sched );
int pthread_attr_getappscheduler(pthread_attr_t attr , pthread_t *sched );
```

DESCRIPTION

These function allows to set/get the application-defined scheduler of an application-scheduled thread (also known as regular threads). Application-scheduled threads are threads scheduled by the operating system, but before they can be scheduled, they need to be activated by their application-defined scheduler.

RETURN VALUE

These functions always return 0.

ERRORS

No errors are defined.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`pthread_attr_setappschedulerstate`

pthread_attr_setappschedulerstate

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`pthread_attr_setappschedulerstate` — Set thread state (regular thread or application-defined scheduler).

Synopsis

```
#include <pthread.h>
int pthread_attr_setappschedulerstate(pthread_attr_t attr , int *type );
```

DESCRIPTION

According to the way a thread is scheduled, threads can be categorized into:

- System-scheduled threads: those scheduled directly by the operating system, without intervention of a scheduler thread.
- Application-scheduled threads: these threads are also scheduled by the operating system, but before they can be scheduled, they need to be activated by their application-defined scheduler.

This function allows to distinguish between system-scheduled threads and application-defined scheduling threads by setting type parameter to: PTHREAD_REGULAR (application-scheduled thread), PTHREAD_APPSCHEDULER (application-defined scheduler thread). By default, without calling this function a thread is supposed to be system-scheduled.

RETURN VALUE

If the call succeeds shall returns 0.

ERRORS

No errors are defined.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`pthread_attr_setappscheduler`

pthread_getappschedparam

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`pthread_getappschedparam, pthread_attr_setappschedparam` — Get/set application scheduled task specific parameters.

Synopsis

```
#include <pthread.h>
int pthread_getappschedparam(pthread_t thread , void *param , int *size );
int pthread_attr_setappschedparam(pthread_attr_t attr , void *param , int *size );
```

DESCRIPTION

In addition to the system priority, application-scheduled tasks have application scheduling parameters that are used to schedule that task contending with the other tasks attached to the same application scheduler.

With `pthread_getappschedparam` function application scheduling specific parameters can be retrieved. While with `pthread_attr_setappschedparam` function can be set.

RETURN VALUE

If the call succeeds `pthread_getappschedparam` function shall return 0 and copy into the locations referenced by param and size the thread's specific application-defined scheduling parameters and its size.

If the call succeeds `pthread_attr_setappschedparam` function shall return 0 and copy into the location referenced by attr the application scheduling parameters referenced by param.

ERRORS

No errors are defined.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`posix_appsched_execute_actions`

pthread_setspecific_for

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`pthread_setspecific_for, pthread_getspecific_from` — Set thread state (regular thread or application-defined scheduler).

Synopsis

```
#include <pthread.h>
int pthread_setspecific_for(pthread_key_t key, pthread_t thread , void *param);
int pthread_getspecific_from(pthread_key_t key, pthread_t thread , void **param, int *size);
```

DESCRIPTION

When a scheduler is processing an event, the scheduling actions to execute will depend on the current scheduling status of its scheduled threads and particularly on the status of the thread which caused that event. It would be very interesting to have a mechanism for obtaining that information in a straightforward and efficient way.

The POSIX-compatible application-defined scheduling API defines two functions that extends POSIX "thread-specific data" functionality: pthread_setspecific_for, pthread_getspecific_from. Those functions permit setting and getting thread-specific data from a thread different from th owner. The scheduler thread can use those functions for attaching and retrieving the scheduling status of its scheduled threads.

RETURN VALUE

If the call succeeds shall return 0.

ERRORS

No errors are defined.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`pthread_getappschedparam`

pthread_sigmask

`<jvidal@disca.upv.es>`

J. Vidal2002-12-12

Name

`pthread_sigmask` — examine and change blocked signals

Synopsis

```
#include <signal.h>
int pthread_sigmask(int  how, const sigset_t *restrict set, sigset_t *restrict oset);
```

DESCRIPTION

This is the RTLinux version of pthread_sigmask. The posix function used to examine or change (or both) the calling thread's signal mask. The argument how, indicates the way in which the mask is changed (SIG_SETMASK, SIG_BLOCK, SIG_UNBLOCK). For a detailed discussion see UNIX spec sigaction, UNIX spec pthread_sigmask(3)

RETURN VALUE

Upon successful completion pthread_sigmask() returns 0; otherwise, it returns -1 and sets the corresponding error number (EFAULT or EINVAL).

ERRORS

The pthread_sigmask() and sigprocmask() functions shall fail if:

- [EINVAL] The value of the how argument is not equal to one of the defined values.
- The pthread_sigmask() function shall not return an error code of [EINTR]

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

RATIONALE

See `pthread_kill(3rtl)` for a discussion of the requirement on delivery of signals.

SEE ALSO

`sigaction(2rtl)`
`sigsuspend(3rtl)`
`sigpending(3rtl)`

sigpending

`<jvidal@disca.upv.es>`

J. Vidal2002-12-12

Name

`sigpending` — examine pending signals.

Synopsis

```
#include <signal.h>
int int sigpending(const rtl_sigset_t * set);
```

DESCRIPTION

This is the RTLinux version of the POSIX function `sigpending`. This function requests the set of signals that are blocked for delivery to the calling thread. For further discussion see UNIX spec sigaction, UNIX spec sigpending(3)

RETURN VALUE

Upon successful completion, sigpending() shall returns 0; otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

No errors are defined.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

sigsuspend

`<jvidal@disca.upv.es>`

J. Vidal2002-12-12

Name

`sigsuspend` — wait for a signal

Synopsis

```
#include <signal.h>
int sigsuspend(const rtl_sigset_t * sigmask);
```

DESCRIPTION

This is the RTLinux version of the POSIX function used to suspend the calling thread until delivery of a non-blocked signal whose action is to execute a signal-catching function. For further discussion see UNIX spec sigaction, UNIX spec sigsuspend(3)

RETURN VALUE

Since sigsuspend() suspends thread execution indefinitely, there is no successful completion return value. If a return occurs, -1 shall be returned and

ERRORS

The sigsuspend() function shall fail if:

- [EINTR] A signal is caught by the calling process and control is returned from the signal-catching function.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

Signal Concepts, See `pause` , `pause` , `sigaction` , `sigaddset` , `sigdelset` , `sigemptyset` , `sigfillset`

timer_create

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`timer_create` — create a per RTLinux process timer.

Synopsis

```
#include <signal.h>
int timer_create(clockid_t  clockid, struct sigevent * restrict evp, timer_t *restrict
timerid);
```

DESCRIPTION

This is a RTLinux version of the POSIX function `timer_create()` . The timer_create() function creates a per RTLinux process timer using the specified clock, clock_id, as the timing base. For further details see UNIX spec sigaction, UNIX spec timer_create

NOTES

In RTLinux timer_create should be called from init_module. A timer created from the RTLinux process, is available to all its threads. RTLinux Timers implementation supports both CLOCK_REALTIME and CLOCK_MONOTONIC.

RETURN VALUE

If the call succeeds, timer_create() shall return zero and update the location referenced by timerid to a timer_t, which can be passed to the per-process timer calls. If an error occurs, the function shall return a value of -1 and set errno to indicate the error. The value of timerid is undefined if an error occurs.

ERRORS

The timer_create() function shall fail if:

- [EAGAIN]: timer_create isn't called from Linux thread (init_module).
- [EINVAL]: The specified clock ID is not defined.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`clock_getres`
`timer_delete`
`timer_getoverrun`

timer_delete

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`timer_delete` — delete a per RTLinux process timer.

Synopsis

```
#include <signal.h>
int timer_delete(timer_t *restrict timerid);
```

DESCRIPTION

This is the RTLinux version of the POSIX function `timer_delete` . The timer_delete() function deletes the specified timer, timerid, previously created by the timer_create() function. For further details see UNIX spec sigaction, UNIX spec timer_delete

NOTES

In RTLinux timer_delete should be called from the Linux thread (init_module, cleanup_module).

RETURN VALUE

If successful, the timer_delete() function shall return a value of zero. Otherwise, the function shall return a value of -1 and set errno to indicate the error.

ERRORS

The timer_delete() function fails if:

- [EINVAL]: The timer ID specified by timerid is not a valid timer ID.
- [EINVAL]: If timer_delete isn't called from Linux thread.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`timer_create`

timer_settime

<jvidal@disca.upv.es>

J. Vidal2002-12-12

Name

`timer_gettime`, `timer_settime` — per RTLinux process timers.

Synopsis

```
#include <signal.h>
int timer_gettime(timer_t timerid, struct itimerspec * value);
int timer_settime(timer_t timerid, int flags, const struct itimerspec * restrict value,
struct itimerspec * restrict ovalue);
```

DESCRIPTION

This are the RTLinux versions for the POSIX functions `timer_gettime()` `timer_settime()` used for requesting the time remaining until next expiration and for arming/disarming a timer, respectively. For further details see, UNIX spec sigaction, UNIX spec timer_gettime & UNIX spec sigaction, UNIX spec timer_settime

RETURN VALUE

If the timer_gettime() or timer_settime() functions succeed, a value of 0 shall be returned.

If an error occurs for any of these functions, the value -1 shall be returned, and errno set to indicate the error.

ERRORS

The timer_gettime(), and timer_settime() functions shall fail if:

- [EINVAL]: The timerid argument does not correspond to an ID returned by timer_create() but not yet deleted by timer_delete().

The timer_settime() function shall fail if:

- [EINVAL] A value structure specified a nanosecond value less than zero or greater than or equal to 1000 million, and the it_value member of that structure did not specify zero seconds and nanoseconds.

AUTHOR

Josep Vidal < jvidal@disca.upv.es>

SEE ALSO

`clock_getres(3rtl)`
`timer_create(3rtl)`

make_linux_task_cbs_server

<patbalbe@disca.upv.es>

P. Balbastre2002-11-06

Name

`make_linux_task_cbs_server` — Change linux task parameters to execute it with CBS scheduling policy.

Synopsis

```
#include <rtl_sched.h>
extern void make_linux_task_cbs_server(hrtime_t start, hrtime_t initbudget, hrtime_t
deadline, hrtime_t period, int priority);
```

DESCRIPTION

`make_linux_task_cbs_server` Change linux task parameters to execute it with `SCHED_CBS_NP` scheduling policy.

`SCHED_CBS_NP` scheduling policy efficiently handle soft real-time requests with a variable or unknown execution behavior under `SCHED_EDF_NP` scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task is assigned a deadline, computed as a function of the reserved bandwidth and its actual requests. If a thread requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded.

This function allows linux task to serve aperiodic events. This way, when an aperiodic event arrives, the linux task "inherits" the properties of the CBS thread by means of the `make_linux_task_cbs_server` function.

`SCHED_EDF_NP` is a scheduling policy that combines both static and dynamic priority scheduling. Threads are ordered by priority, and among the same static priority threads with closer deadline first. Each thread has two scheduling attributes: priority and deadline. It is a Rate Monotonic policy and an EDF policy at each priority level.

RETURN VALUE

No value returned.

ERRORS

Parameter checking is not performed. Incorrect parameters may result in random errors.

AUTHOR

Patricia Balbastre <Patricia@disca.upv.es>
Pau Mendoza <pabmench@disca.upv.es>

SEE ALSO

pthread_setinitbudget_np(3rtl). pthread_attr_setinitbudget_np(3rtl). pthread_initcbs_np(3rtl).

pthread_attr_setinitbudget_np

<patbalbe@disca.upv.es>

P. Balbastre2002-11-06

Name

pthread_attr_setinitbudget_np, pthread_attr_getinitbudget_np — examine and change the initbudget thread attribute

Synopsis

```
#include <rtl_sched.h>
int pthread_attr_setinitbudget_np(pthread_attr_t  *thread, hrtime_t initbudget);
int pthread_attr_getinitbudget_np(pthread_attr_t *thread, hrtime_t *initbudget);
```

DESCRIPTION

This function is a non-portable Real-Time Linux extension.

pthread_attr_setinitbudget_np() and pthread_attr_getinitbudget_np() function set and get the initbudget parameter of attr structure. Before calling these functions, structure pointed by attr must be initialized by calling pthread_attr_init().

The initbudget parameter is defined for those threads whose scheduling policy is SCHED_CBS_NP.

SCHED_CBS_NP scheduling policy efficiently handle soft real-time requests with a variable or unknown execution behaviour under EDF (SCHED_EDF_NP) scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task is assigned a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded.

SCHED_EDF_NP is a scheduling policy that combines both static and dynamic priority scheduling. Threads are ordered by priority, and among the same static priority threads with closer deadline first. Each thread has two scheduling attributes: priority and deadline. It is a Rate Monotonic policy and an EDF policy at each priority level.

On initializing attr, the attribute initbudget takes the value 0.

RETURN VALUE

Both functions returns zero.

ERRORS

Parameter checking is not performed. Incorrect parameters may result in random errors.

AUTHOR

Patricia Balbastre <patricia@disca.upv.es>

Pau Mendoza <pabmench@disca.upv.es>

SEE ALSO

pthread_setinitbudget_np(3rtl). pthread_initcbs_np(3rtl).

pthread_initcbs_np

<patbalbe@disca.upv.es>

P. Balbastre2002-11-06

Name

pthread_initcbs_np — Initializes a thread with SCHED_CBS_NP scheduling policy.

Synopsis

```
#include <rtl_sched.h>
int pthread_initcbs_np(pthread_t  thread, hrtime_t period);
```

DESCRIPTION

This function is a non-portable Real-Time Linux extension.

pthread_initcbs_np initializes a thread with SCHED_CBS_NP scheduling policy.

For those threads with SCHED_CBS_NP scheduling policy the period of the thread must be defined calling this function. For periodic threads with other scheduling policies it must be used the pthread_make_periodic_np function.

SCHED_CBS_NP scheduling policy efficiently handle soft real-time requests with a variable or unknown execution behaviour under SCHED_EDF_NP scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task is assigned a deadline, computed as a function of the reserved bandwidth and its actual requests. If a thread requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded.

SCHED_EDF_NP is a scheduling policy that combines both static and dynamic priority scheduling. Threads are ordered by priority, and among the same static priority threads with closer deadline first. Each thread has two scheduling attributes: priority and deadline. It is a Rate Monotonic policy and an EDF policy at each priority level.

RETURN VALUE

Returns zero.

ERRORS

Parameter checking is not performed. Incorrect parameters may result in random errors.

AUTHOR

Patricia Balbastre <patricia@disca.upv.es>

Pau Mendoza <pabmench@disca.upv.es>

SEE ALSO

pthread_setinitbudget_np(3rtl). pthread_attr_setinitbudget_np(3rtl).

pthread_setinitbudget_np

<patbalbe@disca.upv.es>

P. Balbastre2002-11-06

Name

`pthread_setinitbudget_np, pthread_getinitbudget_np` — set get the initial budget of a thread

Synopsis

```
#include <rtl_sched.h>
int pthread_setinitbudget_np(pthread_t thread, hrtime_t initbudget);
int pthread_getinitbudget_np(pthread_t thread, hrtime_t *initbudget);
```

DESCRIPTION

This function is a non-portable Real-Time Linux extension.

`pthread_setinitbudget_np()` and `pthread_getinitbudget_np()` function set and get the initbudget parameter of thread. The scheduling policy of a thread is automatically set to `SCHED_CBS_NP` when an initial budget is assigned with `pthread_setinitbudget_np()`.

`SCHED_CBS_NP` scheduling policy efficiently handle soft real-time requests with a variable or unknown execution behaviour under EDF (`SCHED_EDF_NP`) scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task is assigned a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded.

`SCHED_EDF_NP` is a scheduling policy that combines both static and dynamic priority scheduling. Threads are ordered by priority, and among the same static priority threads with closer deadline first. Each thread has two scheduling attributes: priority and deadline. It is a Rate Monotonic policy and an EDF policy at each priority level.

`SCHED_FIFO`, `SCHED_RR` and `SCHED_EDF_NP` are static priority policies, the difference is the way that each policy cope with threads of the same priority: `SCHED_FIFO` threads in a first in first out order; `SCHED_RR` thread are time sliced; and `SCHED_EDF_NP` threads are ordered by absolute deadline (closer absolute deadline are executed first). If all threads have different priorities, the three policies will execute threads in the same order.

There are two ways to set the initbudget value of a thread: by calling `pthread_setinitbudget_np()` to set the initbudget of an already created thread; or before it is created at the thread creation attributes (see `pthread_attr_setinitdeadline_np(3rtl)`).

RETURN VALUE

Both always functions returns zero.

ERRORS

Parameter checking is not performed. Incorrect parameters may result in random errors.

AUTHOR

Patricia Balbastre <patricia@disca.upv.es>
Pau Mendoza <pabmench@disca.upv.es>

SEE ALSO

`pthread_attr_setinitbudget_np(3rtl)`, `pthread_initcbs_np(3rtl)`,