WP10 -D10.8 Programmer's Guide

Document Presentation

Project Coordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14, 46022 Valencia, Spain
Phone:	+34 963877576
Fax:	+34 963877576
Email:	alfons@disca.upv.es

Participant List

Role	***Id.***	***Participant Name***	***Acronym***	***Country***
CO	1	Universidad Politecnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA/DRT/LIST/DTSI	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	***Date***	***Reason of change***
1_0	15/02/2004	First release
1_1	15/11/2004	Extend the configuration part
1_2	15/02/2005	Rewrite installation part, configuration part, add components, applications

Forword
that must disapear asap

This document is still in a draft version, the following actions are to be done to get the document in a final version:

1) verify the names of the chapter's authors
2) Index for tables and figures
3) Glossary
4) a standard figure caption style all along the documentation
5) chapters missing: Onetd: network interface programing by Pierre Morel
6) chapter missing: Driver's framework
7) Re-reading and corrections by Ocera members

Table of Contents

PART I

Programming environment

Introduction

By Pierre Morel - MNIS

This is the OCERA Programmer's Guide. We will try through this document to help you programming with OCERA.

In this first chapter we will explain the goal of this guide, we assume that you already read the User's Guide, so that you are familiar with the concepts used all along the guide.

Overview

Intended *Audience*

This guide is intended for

- **software engineers** who are building a real time embedded application for **commercial** or **educational** use with OCERA.
- **teachers** who want to rapidly build a real-time plate-form for training of **students**. In this case the teachers will also have interest in the OCERA document named *TRAINING DOCUMENTATION AND CASE STUDIES*, where they will find ready to use training examples.

Pre-requisite:

To use OCERA kernel and components and understand how they work together you will need a **basic knowledge on** how an **Operating System** is working, both for **real-time** OS and **time sharing** OS, and for some of the components, a basic knowledge of TCP/IP network architectures.

To develop applications using OCERA you will need some skills in a development language, **C** is the standard language for OCERA, while C++ is used in ORTE and Ada is supported for application's programming. Depending on the developments you intend to do and which components you intend to use and certainly a knowledge of **POSIX thread** programming would help a lot.

What will you find in the Programmer's Guide

The Programmer's guide will present you

- The **OCERA environment** where your application will execute: Linux and RTLinux, the OCERA Architecture. How they work together.
- **Programming in for Hard Real-Time**: the real-time Application Programming Interface, the tools you will need to compile your application and to let it run with the system.
- **Programming in for Soft Real-Time**: the soft real-time Application Programming Interface, the tools you will need to compile your application and to let it run with the system.
- A **driver's framework**, and how to write drivers in Linux taking care of the real-time necessities.
- **RTLinux/Linux Interfaces**: Interfaces between the real-time system and the time sharing system.

- **Qconf programming**: The way to add new components to the OCERA tree
- **The components** and the programming interface to the different components of OCERA like Fault Tolerance, CAN CAN/Open Interface and ORTE: OCERA Real Time Ethernet

What will you NOT find in the Programmer's Guide

The Programmer's guide will not present you a general aspect of OCERA nor will it present to you how to setup the development environment or to setup a cross compiler environment.

All these features are presented in the OCERA User's Guide.

Overview

By Pierre Morel – MNIS

OCERA architecture

RTLinux-GPL

Original RTLinux-GPL architecture can be divided in two levels:

- A **basic real-time operating system**, handling interrupts and providing a minimal development interface for real-time threads. We will sometime refer to this level by simply RTLinux-GPL. It is a Hard real-time level with interrupt latency and thread switch latency in the order of a few tens of micro seconds (on a PIII-1GHZ).
- A **time sharing operating system**, the Linux level, having the full functionalities of the original Linux operating system, and running as the idle thread of the basic real-time system.

Both operating systems co-operate in many ways:

- **interrupts**: the original Linux Operating system is modified so that it does not do any direct hardware access for interrupts handling, letting the work to be done by RTLinux-GPL. If a Linux ISR is associated with the Interrupt, RTLinux-GPL, mark the Interrupt as to be served and calls the Linux handlers as soon as nothing more is to be done at real-time level.

- **Realtime-FIFO**: if a real-time thread and a Linux task want to exchange data, they can do it through *real-time fifo*, this is a good way to ensure a proper switch between the real-time OS and the shared time OS. The *real-time fifo* use *soft IRQ to* synchronize the Linux task and the real-time thread.
- **Shared Memory**: is another way to exchange data between Linux and RTLinux-GPL. The synchronization must be done by the application by atomic_test_and_set() calls for exemple.
- **BSD Socket**: an implementation of the BSD Socket interface for UDP protocol allow a real-time thread and a Linux process to communicate. The synhronization, as with the real-time FIFOs is done by Soft-IRQs.

We also found of great interest to have the possibility to add a soft real-time level to Linux, using and enhancing the LOW LATENCY and the PREEMPTION patches, this are the first bricks to provide *Soft real-time* and Q*uality Of Service* at the Linux (shared time OS) level, and then to applications running on Linux, like Video Streaming.

You can see a much deeper description of the architecture in the document "*OCERA ARCHITECTURE*" D02-1.pdf and we advise you to do so if you want to have a good understanding of the internals of OCERA.

The components

We define a component as:

> *„A piece of software that brings some new functionality or feature at different levels in some of the fields: Scheduling, Quality Of Service, Fault Tolerance or Communications."*

Remember that our goal is to enhance an existing Operating system, RTLinux-GPL, to achieve an industry ready operating system and that to achieve this want to give RTLinux:

- A real POSIX 1.1 development interface and new scheduling algorithm and new synchronization mechanisms for RTLinux.
- Quality Of Service, to allow bandwidth reservation
- Fault Tolerance and reconfiguration
- Communication with industry standard control/command devices

All the component interact with some of the other components:

Communication

CANBUS

CANBUS drivers works under Linux and/or RTLinux and provides a virtual interface for testing and development purpose under Linux.

We will explain deeper the CANBUS drivers and the usage of the drivers in the chapter XX: CANBUS.

Socket Interface

A BSD like socket interface provides access to the network for the RealTime threads by using the Linux socket implementation.

We will explain deeper the socket interface and its usage in the chapter XX: Onetd.

ORTE

ORTE stands for **OCERA Real Time Ethernet** and implements the **Real Time Publisher Subscriber** protocol.

The RTSP protocol allow publishers to reserve some of the ethernet bandwidth and manage this reservation so that the bandwidth allocated for each participant allow the data transfert time over ethernet to be predictable.

ORTE is able to work under Linux or under RTLinux with the Onetd socket interface, the choice is made at compile time.

We will see ORTE in deep in the chapter XX: ORTE.

Fault Tolerance

Fault tolerance can collaborate with the Quality Of Service component to handle budget reservation exceptions.

In the case of a distributed network, Fault Tolerance must use a real-time aware communication protocol like the RTSP protocol implemented by the communication component ORTE.

We will investigate the way to use Fault Tolerance in deep in the chapter XX: Faul Tolerance

Quality Of Service

Quality Of Service in OCERA allows a Linux Process to do a CPU Bandwidth reservation.

This means that, the process having done this reservation is given access to the CPU at regular times without the influence of any other processes.

This has the following implications:

- First, the Linux Scheduler must be made preemptive. To do this we have to use the preemptive patch for Linux. We also reduced the Linux latency by using the low latency patch.
- Second the Linux scheduler algorithm must be modified to allow a **CBS** Constant Bandwidth Scheduler, algorithm.
- Third, in the case of Linux working over RTLinux-GPL, RTLinux-GPL scheduling algorithm must be changed to also allow a CBS algorithm.
- Fourth: both Linux and RTLinux scheduler must be aware of the bandwidth reservation

This Quality Of Service is, for example, very useful in the case we have real time constraint at both Linux and RTLinux levels.

A good exemple for this is the real time video streaming application made by Visual Tools and presented at the end of this guide.

We will go deeper in the way to use the Quality Of service in the chapter XXX: Quality Of Service.

RTLinux components

RTLinux-GPL had to be enhanced to achieve our goals. As we saw earlier, we have to provide a POSIX 1.1 development interface, and modify the scheduling algorithm.

By the way OCERA integrated new components: a socket interface used by the RTLinux-GPL implementation of ORTE and Ada runtime.

We will see all POSIX components in the Programmer's Guide, and we will take a look at the possibilities offered by the Onetd Socket interface in the chapter XXX and at the Ada runtime in the chapter XX.

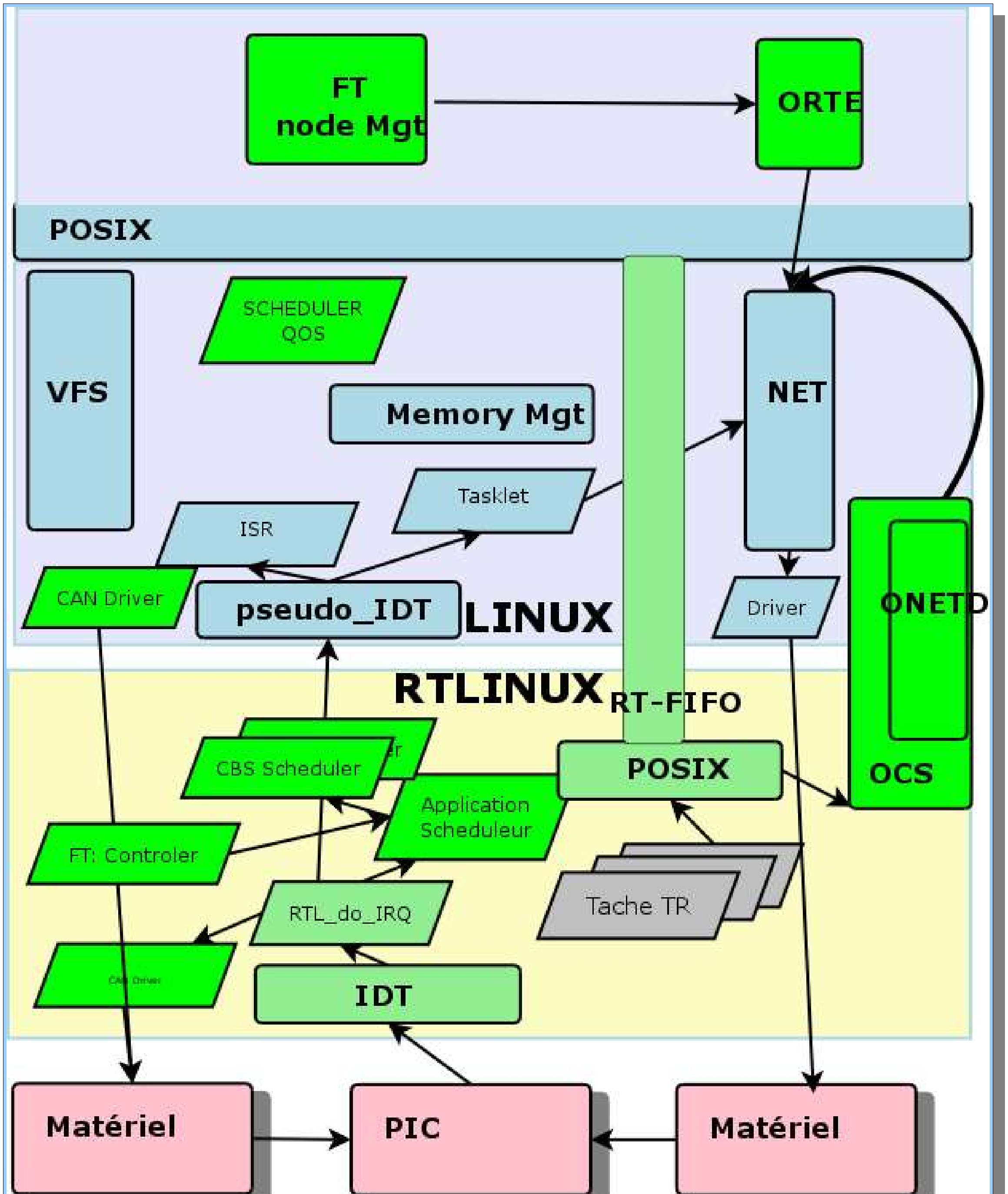


Illustration 1 Overview of OCERA implementation

How RTLinux and Linux work together

To understand this chapter, internal knowledge of Linux kernel and knowledge of the way Linux modules works is a good think.

However an engineer with good knowledge of operating systems will understand the way RTLinux takes control of Linux without deep Linux kernel knowledge.

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*

Interrupt handling

The interrupts are processed in four levels. The first three levels are called whenever an interrupt is called they are:

- *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 for the transition

```
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()
```

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
```

Supported target

Basicaly, OCERA is able to support all targets supported by RTLinux-GPL and Linux. The most restrictive being RTLinux-GPL.

OCERA support architectures based on

- Intel ix86,
- powerPC 603e / Mototola 8240
- ARM and Strong ARM,

while the Board Support Package (BSP) include:

- Standard PC
- PC104
- PPC6000
- iPAQ

The OCERA system can be loaded on the target on IDE and SCSI disks, Flash memory, SD-Memory, ROM or even use the network to be downloaded using TFTP or BootP and a PXE boot loader in ROM..

We provide more information on the different architectures and on cross compilation in the OCERA User's Guide.

Development tools

The tools used to develop applications with OCERA are of very common use:

The GNU Utilities: the GNU C compiler, assembler, linker, The CML2 Utilities: CML2 is the definition language used by the standard Linux kernel, above 2.5.2, to define the kernel configuration

To this minimal set of utilities, you will eventually need the OCERA tracing tools or the GNU Debugger: GDB.

If you develop with Ada language you will need a special Ada compiler. But this will be explained in a separate chapter dedicated to Ada in this document.

PART II

Realtime programming

Scheduling

timers

Inter thread communication
fifo
shared memory
semaphore, mutex, signals, spinlock

RTLinux API

By:
Patricia Balbastre
Alfons Crespo
Ismael Ripoll

Dynamic memory allocator

Description

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

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.

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.

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 allocation:
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);
```

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);
}
```

POSIX Signals

Description

This component extends the signalling 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) signalling infrastructure. Although real-time 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) .

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.

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

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]);
}
```

POSIX Timers

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.

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.

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.

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".

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);
}
```

POSIX message queues

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.

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.

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 *);
```

Posix Barriers

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

- The barrier attributes are initialized .
  This is accomplished trough the function pthread_barrierattr_init
- 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.
- 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.
- 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.

Usage

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

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.

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 attributes object can be reinitialized using pthread_barrierattr_init(); the results of otherwise referencing the object after it has been destroyed are undefined. An implementation may cause pthread_barrierattr_destroy() to set the object referenced by attr to an invalid value.

*int pthread_barrierattr_init(pthread_barrierattr_t * attr);*

The pthread_barrierattr_init() function shall initialize a barrier attributes object attr with the default value for all of the attributes defined by the implementation.

Results are undefined if pthread_barrierattr_init() is called specifying an already initialized attr attributes object.

*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 barrier and shall initialize the barrier with attributes referenced by attr. If attr is NULL, the default barrier attributes shall be used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if pthread_barrier_init() is called when any thread is blocked on the barrier (that is, has not returned from the pthread_barrier_wait() call). The results are undefined if a barrier is used without first being initialized. The results are undefined if pthread_barrier_init() is called specifying an already initialized barrier.

*int pthread_barrier_destroy(pthread_barrier_t * barrier);*

The pthread_barrier_destroy() function shall destroy the barrier referenced by barrier and release any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to pthread_barrier_init(). An implementation may use this function to set barrier to an invalid value. The results are undefined if pthread_barrier_destroy() is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

*int pthread_barrier_wait(pthread_barrier_t * barrier);*

The pthread_barrier_wait() function shall synchronize participating threads at the barrier referenced by barrier. The calling thread shall block until the required number of threads have called pthread_barrier_wait() specifying the barrier.

When the required number of threads have called pthread_barrier_wait() specifying the barrier, the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall be reset to the state it had as a result of the most recent pthread_barrier_init() function that referenced it.

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 from the attributes object referenced by attr. The pthread_barrierattr_setpshared() function shall set the process-shared attribute in an initialized attributes object referenced by attr. The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the barrier shall only be operated upon by threads created within the same process as the thread that initialized the barrier; if threads of different processes attempt to operate on such a barrier, the behavior is undefined.

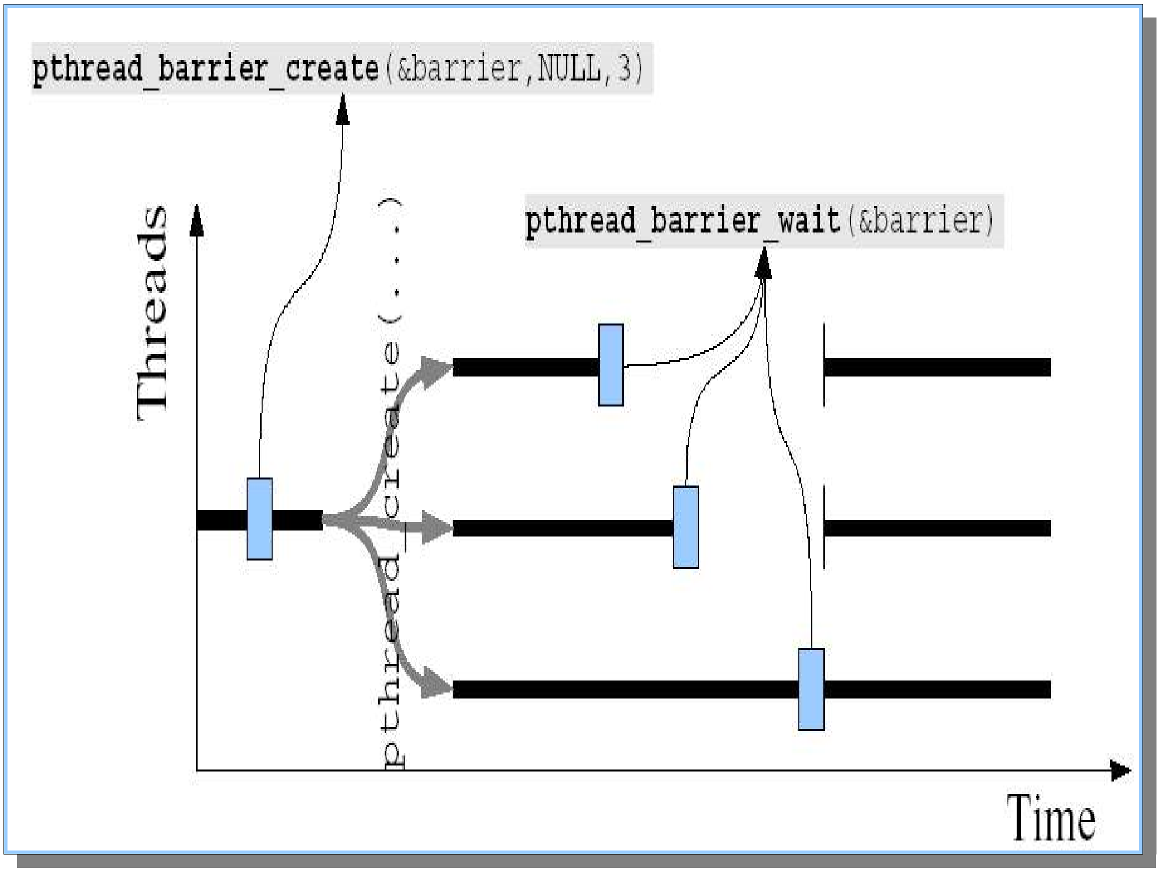
int pthread_barrier_wait(pthread_barrier_t * barrier);

The pthread_barrier_wait() function shall synchronize participating threads at the barrier referenced by barrier. The calling thread shall block until the required number of threads have called pthread_barrier_wait() specifying the barrier.

When the required number of threads have called pthread_barrier_wait() specifying the barrier, the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall be reset to the state it had as a result of the most recent pthread_barrier_init() function that referenced it.

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 next page).




```c
#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);
}
```

Application-Defined Scheduler

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.

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).

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.

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 pthread_mutex_t *mutex)

Execute Scheduling Actions

- 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)

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_eventset_t * set)
  - 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, pthread_attr_t * attr, pthread_t sched)
- 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 * appscheduler)
- int pthread_mutexattr_setappschedparam (pthread_mutexattr_t * attr, const struct pthread_mutex_schedparam * sched_param)
- int pthread_mutexattr_getappschedparam (const pthread_mutexattr_t * attr, struct pthread_mutex_schedparam * sched_param)
- int pthread_mutex_setappschedparam (pthread_mutex_t * mutex, const struct pthread_mutex_schedparam * sched_param)
- int pthread_mutex_getappschedparam (const pthread_mutex_t * mutex, struct pthread_mutex_schedparam * sched_param)

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);


}
```

Ada Support

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 real-time 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.

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/), and REMOVE any other gnat included in your Linux distribution. Otherwise, broken executables and libraries can be generated.

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, the GNAT_PATH variable is the location of the gnat compiler and libraries, which usually is /usr/gnat

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

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

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 examples in $GNAT_PATH/rtl_examples (usually at /usr/gnat/rtl_examples). To compile the examples, just run "make" from the mentioned directory.

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
```

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;
```

POSIX tracing

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.

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.

Data types

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.

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.

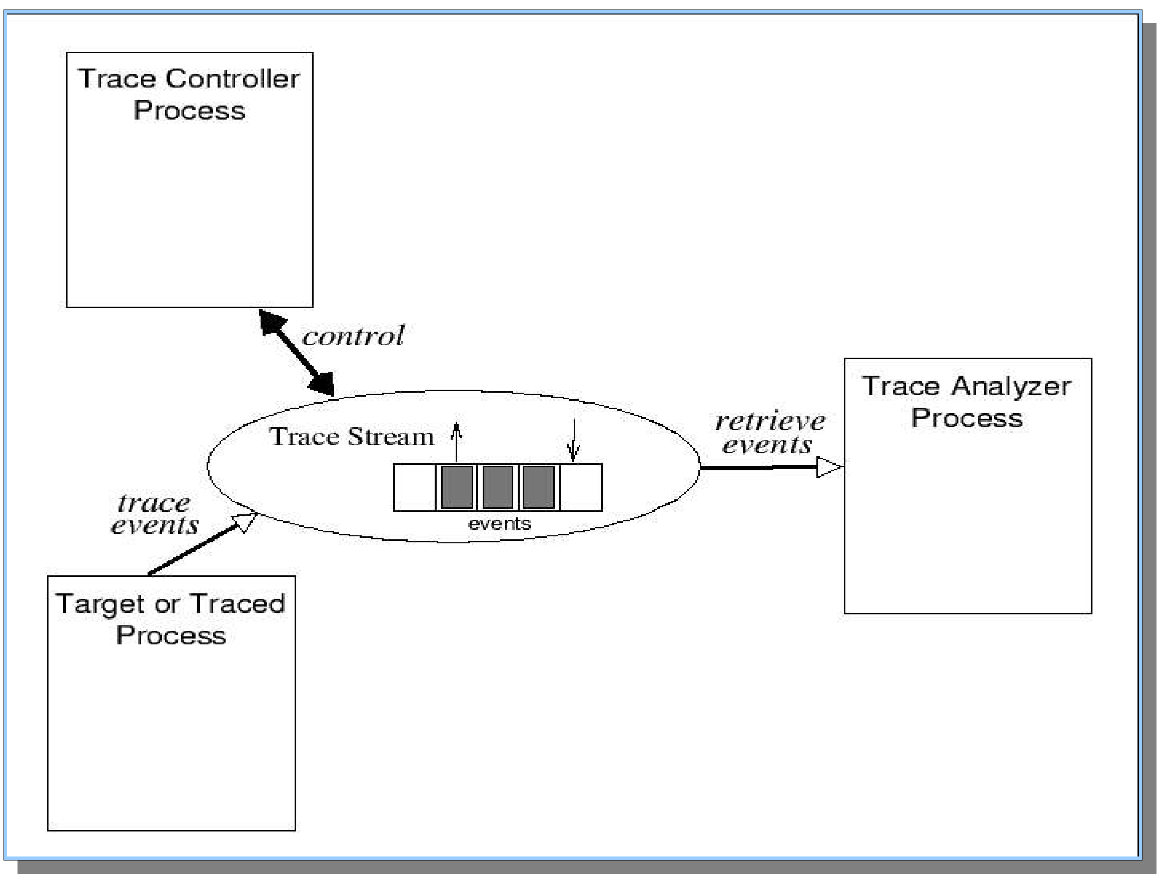
The standard establishes that, before any event can be stored for a process, a trace stream has to be explicitly created to trace that particular process (the process pid is one of the arguments of the stream creation function). In the most general case, the relationship between streams and processes 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.

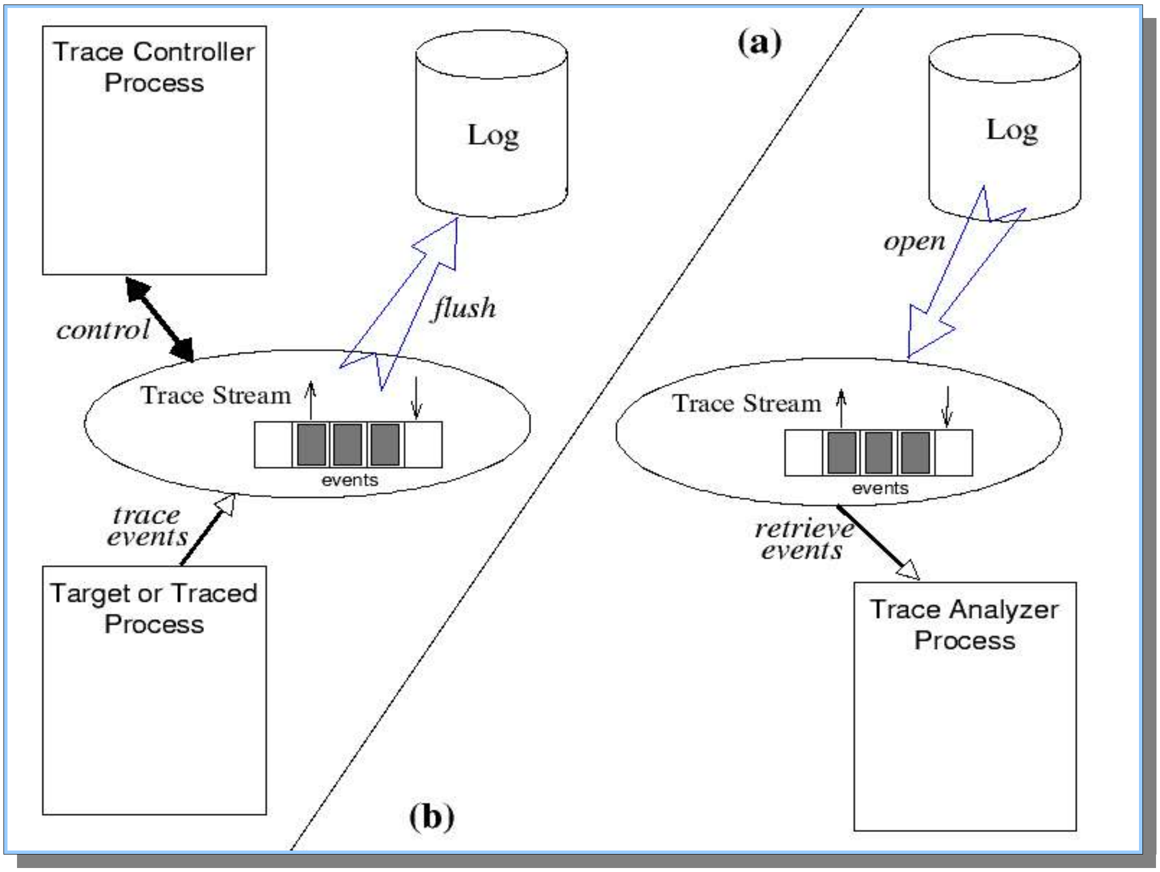
The standard defines two classes of trace streams: active and pre-recorded, which are described below.

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.

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.

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.

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.

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.

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/.

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 world!hello world!hello world!hello world!hello world!hello world!hello world!\0";
  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();
}
```

PART III

RTLinux/Linux interface

printf
fifo
shared memory
onetd
modules
/proc

Debuging and tracing
POSIX Traces

Driver
Driver

PART IV

Driver framework

Interrupt
System interface
low level routines rtl_a-set

Driver

PART V

OCERA Components

Network

Fault Tolerance

Quality of Services

ORTE

CAN

Quality Of Services

By
Luca Marzario - SSSA

in attachment there are two file that are a skeleton for program that want to use reservation and feedback scheduling
Before run that program, naturally, user have to insert the relative module (cbs_sched.o and qmgr_sched.o) into the kernel with insmod command. No other steps are needed (you are right: the patch is already integrated).

```
/*
 *  Copyright (C) 2003 Luca Marzario
 *  This is Free Software; see GPL.txt for details
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sched.h>
#define __QMGR__
#include "qmgr.h"
#define CBS_PERIOD      10000 /* period of reservation */
#define QMGR_MAX_BUDGET  5000 /* max budget that can be asssigned to CBS server */
#define QMGR_PERIOD     80000 /* period of feedback function */
#define QMGR_MIN_EXEC    5000 /* minimum exstimated execution time */
#define QMGR_MAX_EXEC   60000 /* maximum exstimated execution time */
#define QMGR_LOW_ERR        0 /* desired lower bound of scheduling error */
#define QMGR_HIGH_ERR    5000 /* desired upper bound of scheduling error */
#define QMGR_MIN_ERR        0 /* absolute minimum scheduling error */
#define QMGR_MAX_ERR    10000 /* absolute maximum scheduling error */

#define qmgr_end_cycle()

int main(int argc, char *argv[])
{
  struct sched_param sp;
  struct qmgr_param cs;
  int res, cond = 0;

  sp.sched_size = sizeof(struct qmgr_param);
```

```
  sp.sched_p = &cs;
  cs.cbs_period = CBS_PERIOD;
  cs.qmgr_max_b = QMGR_MAX_BUDGET;
  cs.qmgr_period = QMGR_PERIOD;
  cs.qmgr_signature = QMGR_SIGNATURE;

  /* parameters for feedback function */
  cs.h  = QMGR_MIN_EXEC;
  cs.H  = QMGR_MAX_EXEC;
  cs.ei = QMGR_LOW_ERR;
  cs.Ei = QMGR_HIGH_ERR;
  cs.e  = QMGR_MIN_ERR;
  cs.E  = QMGR_MAX_ERR;

  /* if you donn't need particular scheduling error target,
     you can use default values: substitute assignment of
     parameters fo feedback function with the following call:

     qmgr_init_default(&cs);
  */

  res = sched_setscheduler(getpid(), SCHED_CBS, &sp);
  if (res < 0) {
    perror("Error in setscheduler!");
    exit(-1);
  }

  do {

    /* my code */

    qmgr_end_cycle();

  } while ( cond ); /* exit condition */

  return 0;
}
```

Programs 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sched.h>
#include "cbs.h"

extern char **environ;

#define MAX_BUDEGET_US     10000 /* max budget per period in us */
#define RESERV_PERIOD_US  100000 /* period of reservation in us*/
```

```
int main(int argc, char *argv[])
{
  struct sched_param sp;     /* used to pass parameters to qres module throug setsched */
  struct cbs_param cs;       /*                    "                              */
  int res;

  sp.sched_size = sizeof(struct cbs_param);
  sp.sched_p = &cs;
  cs.signature = CBS_SIGNATURE;
  cs.type = CBS_TYPE_DEFAULT;
  cs.max_budget = MAX_BUDEGET_US;
  cs.period = RESERV_PERIOD_US;

  res = sched_setscheduler(getpid(), SCHED_CBS, &sp);
  if (res < 0) {
    perror("Error in setscheduler!");
    exit(-1);
  }

  /* my code */

  return 0;
}
```

Driver

OCERA Real-Time Ethernet

By
Jan Krakora CTU
Pavel Pisa CTU
Frantisek Vacek CTU
Zdenek Sebek CTU
Petr Smolik CTU
Zdenek Hanzalek CTU

The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. RTPS is new application layer protocol targeted to real-time communication area, which is build on the top of standard UDP stack. Since there are many TCP/IP stack implementations under many operating systems and RTPS protocol does not have any other special HW/SW requirements, it should be easily ported to many HW/SW target platforms. Because it uses only UDP protocol, it retains control of timing and reliability.

ORTE API

Data types

Table of Contents

struct ORTESubsProp -- properties of a subscription
struct ORTEAppInfo --
struct ORTEPubInfo -- information about publication
struct ORTESubInfo -- information about subscription
struct ORTEPublStatus -- status of a publication
struct ORTESubsStatus -- status of a subscription
struct ORTERecvInfo -- description of received data
struct ORTESendInfo -- description of sending data
struct ORTEDomainAppEvents -- Domain event handlers of an application
struct ORTETasksProp -- ORTE task properties, not supported
struct ORTEDomainProp -- domain properties

enum SubscriptionMode

Name

enum SubscriptionMode -- mode of subscription

Synopsis

```
enum SubscriptionMode {
   PULLED,
   IMMEDIATE
 };
```

Constants

PULLED

polled

IMMEDIATE

using callback function

Description

Specifies whether user application will poll for data or whether a callback function will be called by ORTE middleware when new data will be available.

enum SubscriptionType

Name

enum SubscriptionType -- type of subcsription

Synopsis

```
enum SubscriptionType {
   BEST_EFFORTS,
   STRICT_RELIABLE
 };
```

Constants

BEST_EFFORTS

best effort subscription

STRICT_RELIABLE

strict reliable subscription.

Description

Specifies which mode will be used for this subscription.

enum ORTERecvStatus

Name

enum ORTERecvStatus -- status of a subscription

Synopsis

```
enum ORTERecvStatus {
   NEW_DATA,
   DEADLINE
 };
```

Constants

NEW_DATA

new data has arrived

DEADLINE

deadline has occurred

Description

Specifies which event has occured in the subscription object.

enum ORTESendStatus

Name

enum ORTESendStatus -- status of a publication

Synopsis

```
 enum ORTESendStatus {
    NEED_DATA,
    CQL
  };
```

Constants

NEED_DATA

need new data (set when callback function specified for publciation is beeing called)

CQL

transmit queue has been filled up to critical level.

Description

Specifies which event has occured in the publication object. Critical level of transmit queue is specified as one of publication properties (ORTEPublProp.criticalQueueLevel).

---

struct ORTEIFProp

Name

struct ORTEIFProp -- interface flags

Synopsis

```
 struct ORTEIFProp {
    int32_t ifFlags;
    IPAddress ipAddress;
  };
```

Members

ifFlags

flags

ipAddress

IP address

Description

Flags for network interface.

---

struct ORTEMulticastProp

Name

struct ORTEMulticastProp -- properties for ORTE multicast (not supported yet)

Synopsis

```
 struct ORTEMulticastProp {
    Boolean enabled;
    unsigned char  ttl;
    Boolean loopBackEnabled;
    IPAddress ipAddress;
  };
```

Members

enabled

ORTE_TRUE if multicast enabled otherwise ORTE_FALSE

ttl

time-to-live (TTL) for sent datagrams

loopBackEnabled

ORTE_TRUE if data should be received by sender itself otherwise ORTE_FALSE

ipAddress

desired multicast IP address

Description

Properties for ORTE multicast subsystem which is not fully supported yet. Multicast IP address is assigned by the ORTE middleware itself.

struct ORTECDRStream

Name

struct ORTECDRStream -- used for serialization

Synopsis

```
struct ORTECDRStream {
   char * buffer;
   char * bufferPtr;
   Boolean needByteSwap;
   int length;
 };
```

Members

buffer

  buffer for data

bufferPtr

  current position within buffer

needByteSwap

  ORTE_TRUE if it is necessary to swap byte ordering otherwise ORTE_FALSE

length

  buffer length

Description

Struct `ORTECDRStream` is used by serialization and deserialization functions.

---

struct ORTETypeRegister

Name

struct ORTETypeRegister -- registered data type

Synopsis

```
struct ORTETypeRegister {
   const char             * typeName;
   ORTETypeSerialize serialize;
   ORTETypeDeserialize deserialize;
   unsigned int           getMaxSize;
 };
```

Members

typeName

name of data type

serialize

pointer to serialization function

deserialize

pointer to deserialization function

getMaxSize

max data type length in bytes

Description

Contains description of registered data type. See `ORTETypeRegisterAdd` function for details.

---

struct ORTEDomainBaseProp

Name

struct ORTEDomainBaseProp -- base properties of a domain

Synopsis

```
struct ORTEDomainBaseProp {
   NtpTime expirationTime;
   NtpTime refreshPeriod;
   NtpTime purgeTime;
   NtpTime repeatAnnounceTime;
   NtpTime repeatActiveQueryTime;
   NtpTime delayResponceTimeACKMin;
   NtpTime delayResponceTimeACKMax;
   unsigned int           HBMaxRetries;
   unsigned int           ACKMaxRetries;
   NtpTime maxBlockTime;
 };
```

Members

expirationTime

> specifies how long is this application taken as alive in other applications/managers (default 180s)

refreshPeriod

> how often an application refresh itself to its manager or manager to other managers (default 60s)

purgeTime

> how often the local database should be cleaned from invalid (expired) objects (default 60s)

repeatAnnounceTime

> This is the period with which the CSTWriter will announce its existence and/or the availability of new CSChanges to the CSTReader. This period determines how quickly the protocol recovers when an announcement of data is lost.

repeatActiveQueryTime

> ???

delayResponceTimeACKMin

> minimum time the CSTWriter waits before responding to an incoming message.

delayResponceTimeACKMax

> maximum time the CSTWriter waits before responding to an incoming message.

HBMaxRetries

> how many times a HB message is retransmitted if no response has been received until timeout

ACKMaxRetries

how many times an ACK message is retransmitted if no response has been received until timeout

maxBlockTime

timeout for send functions if sending queue is full (default 30s)

struct ORTEDomainWireProp

Name

struct ORTEDomainWireProp -- wire properties of a message

Synopsis

```
struct ORTEDomainWireProp {
   unsigned int          metaBytesPerPacket;
   unsigned int          metaBytesPerFastPacket;
   unsigned int          metabitsPerACKBitmap;
   unsigned int          userMaxSerDeserSize;
 };
```

Members

metaBytesPerPacket

maximum number of bytes in single frame (default 1500B)

metaBytesPerFastPacket

maximum number of bytes in single frame if transmitting queue has reached `criticalQueueLevel` level (see `ORTEPublProp` struct)

metabitsPerACKBitmap

not supported yet

userMaxSerDeserSize

maximum number of user data in frame (default 1500B)

struct ORTEPublProp

Name

struct ORTEPublProp -- properties of a publication

Synopsis

```
struct ORTEPublProp {
   PathName topic;
   TypeName typeName;
   TypeChecksum typeChecksum;
   Boolean expectsAck;
   NtpTime persistence;
   u_int32_t reliabilityOffered;
   u_int32_t sendQueueSize;
   int32_t strength;
   u_int32_t criticalQueueLevel;
   NtpTime HBNornalRate;
   NtpTime HBCQLRate;
   unsigned int           HBMaxRetries;
   NtpTime maxBlockTime;
  };
```

Members

topic

the name of the information in the Network that is published or subscribed to

typeName

the name of the type of this data

typeChecksum

a checksum that identifies the CDR-representation of the data

expectsAck

indicates wherther publication expects to receive ACKs to its messages

persistence

indicates how long the issue is valid

reliabilityOffered

reliability policy as offered by the publication

sendQueueSize

size of transmitting queue

strength

precedence of the issue sent by the publication

criticalQueueLevel

treshold for transmitting queue content length indicating the queue can became full immediately

HBNornalRate

how often send HBs to subscription objects

HBCQLRate

how often send HBs to subscription objects if transmittiong queue has reached `criticalQueueLevel`

HBMaxRetries

how many times retransmit HBs if no replay from target object has not been received

maxBlockTime

unsupported

struct ORTESubsProp

Name

struct ORTESubsProp -- properties of a subscription

Synopsis

```
struct ORTESubsProp {
   PathName topic;
   TypeName typeName;
   TypeChecksum typeChecksum;
   NtpTime minimumSeparation;
   u_int32_t recvQueueSize;
   u_int32_t reliabilityRequested;
   //additional parametersNtpTime                deadline;
   u_int32_t mode;
 };
```

Members

topic

the name of the information in the Network that is published or subscribed to

typeName

the name of the type of this data

typeChecksum

a checksum that identifies the CDR-representation of the data

minimumSeparation

minimum time between two consecutive issues received by the subscription

recvQueueSize

size of receiving queue

reliabilityRequested

reliability policy requested by the subscription

deadline

deadline for subscription, a callback function (see `ORTESubscriptionCreate`) will be called if no data were received within this period of time

mode

mode of subscription (strict reliable/best effort), see `SubscriptionType` enum for values

struct ORTEAppInfo

Name

struct ORTEAppInfo --

Synopsis

```
struct ORTEAppInfo {
   HostId hostId;
   AppId appId;
   IPAddress * unicastIPAddressList;
   unsigned char          unicastIPAddressCount;
   IPAddress * metatrafficMulticastIPAddressList;
   unsigned char          metatrafficMulticastIPAddressCount;
```

```
    Port metatrafficUnicastPort;
    Port userdataUnicastPort;
    VendorId vendorId;
    ProtocolVersion protocolVersion;
  };
```

Members

hostId

hostId of application

appId

appId of application

unicastIPAddressList

unicast IP addresses of the host on which the application runs (there can be multiple addresses on a multi-NIC host)

unicastIPAddressCount

number of IPaddresses in `unicastIPAddressList`

metatrafficMulticastIPAddressList

for the purposes of meta-traffic, an application can also accept Messages on this set of multicast addresses

metatrafficMulticastIPAddressCount

number of IPaddresses in `metatrafficMulticastIPAddressList`

metatrafficUnicastPort

UDP port used for metatraffic communication

userdataUnicastPort

UDP port used for metatraffic communication

vendorId

identifies the vendor of the middleware implementing the RTPS protocol and allows this vendor to add specific extensions to the protocol

protocolVersion

describes the protocol version

---

struct ORTEPubInfo

Name

struct ORTEPubInfo -- information about publication

Synopsis

```
struct ORTEPubInfo {
   const char           * topic;
   const char           * type;
   ObjectId objectId;
 };
```

Members

topic

: the name of the information in the Network that is published or subscribed to

type

: the name of the type of this data

objectId

: object providing this publication

struct ORTESubInfo

Name

struct ORTESubInfo -- information about subscription

Synopsis

```
struct ORTESubInfo {
   const char           * topic;
   const char           * type;
   ObjectId objectId;
 };
```

Members

topic

the name of the information in the Network that is published or subscribed to

type

the name of the type of this data

objectId

object with this subscription

struct ORTEPublStatus

Name

struct ORTEPublStatus -- status of a publication

Synopsis

```
struct ORTEPublStatus {
   unsigned int           strict;
   unsigned int           bestEffort;
   unsigned int           issues;
 };
```

Members

strict

count of unreliable subscription (strict) connected on responsible subscription

bestEffort

count of reliable subscription (best effort) connected on responsible subscription

issues

number of messages in transmitting queue

struct ORTESubsStatus

Name

struct ORTESubsStatus -- status of a subscription

Synopsis

```
struct ORTESubsStatus {
   unsigned int          strict;
   unsigned int          bestEffort;
   unsigned int          issues;
 };
```

Members

strict

count of unreliable publications (strict) connected to responsible subscription

bestEffort

count of reliable publications (best effort) connected to responsible subscription

issues

number of messages in receiving queue

struct ORTERecvInfo

Name

struct ORTERecvInfo -- description of received data

Synopsis

```
struct ORTERecvInfo {
    ORTERecvStatus status;
    const char           * topic;
    const char           * type;
    GUID_RTPS senderGUID;
    NtpTime localTimeReceived;
    NtpTime remoteTimePublished;
    SequenceNumber sn;
  };
```

Members

status

status of this event

topic

the name of the information

type

the name of the type of this data

senderGUID

GUID of object who sent this information

localTimeReceived

local timestamp when data were received

remoteTimePublished

remote timestam when data were published

sn

sequencial number of data

struct ORTESendInfo

Name

struct ORTESendInfo -- description of sending data

Synopsis

```
struct ORTESendInfo {
```

```
    ORTESendStatus status;
    const char            * topic;
    const char            * type;
    GUID_RTPS senderGUID;
    SequenceNumber sn;
  };
```

Members

status

status of this event

topic

the name of the information

type

the name of the type of this information

senderGUID

GUID of object who sent this information

sn

sequencial number of information

struct ORTEDomainAppEvents

Name

struct ORTEDomainAppEvents -- Domain event handlers of an application

Synopsis

```
 struct ORTEDomainAppEvents {
    ORTEOnMgrNew onMgrNew;
    void * onMgrNewParam;
    ORTEOnMgrDelete onMgrDelete;
    void * onMgrDeleteParam;
    ORTEOnAppRemoteNew onAppRemoteNew;
    void * onAppRemoteNewParam;
    ORTEOnAppDelete onAppDelete;
    void * onAppDeleteParam;
    ORTEOnPubRemote onPubRemoteNew;
    void * onPubRemoteNewParam;
    ORTEOnPubRemote onPubRemoteChanged;
    void * onPubRemoteChangedParam;
    ORTEOnPubDelete onPubDelete;
    void * onPubDeleteParam;
    ORTEOnSubRemote onSubRemoteNew;
```

```
    void * onSubRemoteNewParam;
    ORTEOnSubRemote onSubRemoteChanged;
    void * onSubRemoteChangedParam;
    ORTEOnSubDelete onSubDelete;
    void * onSubDeleteParam;
  };
```

Members

onMgrNew

new manager has been created

onMgrNewParam

user parameters for `onMgrNew` handler

onMgrDelete

manager has been deleted

onMgrDeleteParam

user parameters for `onMgrDelete` handler

onAppRemoteNew

new remote application has been registered

onAppRemoteNewParam

user parameters for `onAppRemoteNew` handler

onAppDelete

an application has been removed

onAppDeleteParam

user parameters for `onAppDelete` handler

onPubRemoteNew

new remote publication has been registered

onPubRemoteNewParam

user parameters for `onPubRemoteNew` handler onPubRemoteChanged

a remote publication's parameters has been changed

onPubRemoteChangedParam

user parameters for `onPubRemoteChanged` handler

onPubDelete

a publication has been deleted

onPubDeleteParam

user parameters for `onPubDelete` handler

onSubRemoteNew

a new remote subscription has been registered

onSubRemoteNewParam

user parameters for `onSubRemoteNew` handler

onSubRemoteChanged

a remote subscription's parameters has been changed

onSubRemoteChangedParam

user parameters for `onSubRemoteChanged` handler

onSubDelete

a publication has been deleted

onSubDeleteParam

user parameters for `onSubDelete` handler

Description

Prototypes of events handler fucntions can be found in file typedefs_api.h.

struct ORTETasksProp

Name

struct ORTETasksProp -- ORTE task properties, not supported

Synopsis

```
struct ORTETasksProp {
   Boolean realTimeEnabled;
   int smtStackSize;
   int smtPriority;
   int rmtStackSize;
   int rmtPriority;
 };
```

Members

realTimeEnabled

not supported

smtStackSize

not supported

smtPriority

not supported

rmtStackSize

not supported

rmtPriority

not supported

struct ORTEDomainProp

Name

struct ORTEDomainProp -- domain properties

Synopsis

```
 struct ORTEDomainProp {
    ORTETasksProp tasksProp;
    ORTEIFProp * IFProp;
    //interface propertiesunsigned char          IFCount;
    //count of interfacesORTEDomainBaseProp     baseProp;
    ORTEDomainWireProp wireProp;
    ORTEMulticastProp multicast;
    //multicast properiesORTEPublProp           publPropDefault;
    //default properties for a Publ/SubORTESubsProp
subsPropDefault;
    char * mgrs;
    //managerschar * keys;
    //keysIPAddress appLocalManager;
    //applicationschar * version;
    //string product versionint                      recvBuffSize;
    int sendBuffSize;
  };
```

Members

tasksProp

task properties

IFProp

properties of network interfaces

IFCount

number of network interfaces

baseProp

base properties (see `ORTEDomainBaseProp` for details)

wireProp

wire properties (see `ORTEDomainWireProp` for details)

multicast

multicast properties (see `ORTEMulticastProp` for details)

publPropDefault

default properties of publiciations (see `ORTEPublProp` for details)

subsPropDefault

default properties of subscriptions (see `ORTESubsProp` for details)

mgrs

list of known managers

keys

access keys for managers

appLocalManager

IP address of local manager (default localhost)

version

string product version

recvBuffSize

receiving queue length

sendBuffSize

transmitting queue length

Functions

Table of Contents

ORTESleepMs -- suspends calling thread for given time

IPAddressToString

Name

IPAddressToString -- converts IP address IPAddress to its string representation

Synopsis

```
char* IPAddressToString (IPAddress ipAddress, char * buff);
```

Arguments

`ipAddress`

source IP address

`buff`

output buffer

StringToIPAddress

Name

StringToIPAddress -- converts IP address from string into IPAddress

Synopsis

```
IPAddress StringToIPAddress (const char * string);
```

Arguments

`string`

source string

NtpTimeToStringMs

Name

NtpTimeToStringMs -- converts NtpTime to its text representation in miliseconds

Synopsis

```
char * NtpTimeToStringMs (NtpTime time, char * buff);
```

Arguments

`time`

time given in NtpTime structure

`buff`

output buffer

NtpTimeToStringUs

Name

NtpTimeToStringUs -- converts NtpTime to its text representation in microseconds

Synopsis

```
char * NtpTimeToStringUs (NtpTime time, char * buff);
```

Arguments

`time`

time given in NtpTime structure

`buff`

output buffer

ORTEDomainStart

Name

ORTEDomainStart -- start specific threads

Synopsis

```
void ORTEDomainStart (ORTEDomain * d, Boolean
recvMetatrafficThread, Boolean recvUserDataThread, Boolean
sendThread);
```

Arguments

`d`

domain object handle

`recvMetatrafficThread`

specifies whether recvMetatrafficThread should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

`recvUserDataThread`

specifies whether recvUserDataThread should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

`sendThread`

specifies whether sendThread should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

Description

Functions `ORTEDomainAppCreate` and `ORTEDomainMgrCreate` provide facility to create an object with its threads suspended. Use function `ORTEDomainStart` to resume those suspended threads.

ORTEDomainPropDefaultGet

Name

ORTEDomainPropDefaultGet -- returns default properties of a domain

Synopsis

```
Boolean ORTEDomainPropDefaultGet (ORTEDomainProp * prop);
```

Arguments

`prop`

pointer to struct ORTEDomainProp

Description

Structure ORTEDomainProp referenced by `prop` will be filled by its default values. Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainInitEvents

Name

ORTEDomainInitEvents -- initializes list of events

Synopsis

```
Boolean ORTEDomainInitEvents (ORTEDomainAppEvents * events);
```

Arguments

`events`

pointer to struct ORTEDomainAppEvents

Description

Initializes structure pointed by `events`. Every member is set to NULL. Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainAppCreate

Name

ORTEDomainAppCreate -- creates an application object within given domain

Synopsis

```
ORTEDomain * ORTEDomainAppCreate (int domain,
ORTEDomainProp * prop, ORTEDomainAppEvents * events,
Boolean suspended);
```

Arguments

`domain`

given domain

`prop`

properties of application

`events`

events associated with application or NULL

`suspended`

specifies whether threads of this application should be started as well (ORTE_FALSE) or stay suspended (ORTE_TRUE). See `ORTEDomainStart` for details how to resume suspended threads

Description

Creates new Application object and sets its properties and events. Return handle to created object or NULL in case of any error.

ORTEDomainAppDestroy

Name

ORTEDomainAppDestroy -- destroy Application object

Synopsis

```
Boolean ORTEDomainAppDestroy (ORTEDomain * d);
```

Arguments

`d`

domain

Description

Destroys all application objects in specified domain. Returns ORTE_TRUE or ORTE_FALSE in case of any error.

ORTEDomainAppSubscriptionPatternAdd

Name

ORTEDomainAppSubscriptionPatternAdd -- create pattern-based subscription

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternAdd (ORTEDomain *
d, const char * topic, const char * type,
ORTESubscriptionPatternCallBack subscriptionCallBack, void
* param);
```

Arguments

`d`

domain object

`topic`

pattern for topic

`type`

pattern for type

`subscriptionCallBack`

pointer to callback function which will be called whenever any data are received through this subscription

`param`

user params for callback function

Description

This function is intended to be used in application interesded in more published data from possibly more remote applications, which should be received through single subscription. These different publications are specified by pattern given to `topic` and `type` parameters.

For example suppose there are publications of topics like `temperatureEngine1`, `temperatureEngine2`, `temperatureEngine1Backup` and `temperatureEngine2Backup` somewhere on our network. We can subscribe to each of Engine1 temperations by creating single subscription with pattern for topic set to "temperatureEngine1*". Or, if we are interested only in values from backup measurements, we can use pattern "*Backup".

Syntax for patterns is the same as syntax for `fnmatch` function, which is employed for pattern recognition.

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainAppSubscriptionPatternRemove

Name

ORTEDomainAppSubscriptionPatternRemove -- remove subscription pattern

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternRemove (ORTEDomain
* d, const char * topic, const char * type);
```

Arguments

`d`

domain handle

`topic`

pattern to be removed

`type`

pattern to be removed

Description

Removes subscritions created by `ORTEDomainAppSubscriptionPatternAdd`. Patterns for `type` and `topic` must be exactly the same strings as when `ORTEDomainAppSubscriptionPatternAdd` function was called.

Returns ORTE_TRUE if successful or ORTE_FALSE if none matching record was found

ORTEDomainAppSubscriptionPatternDestroy

Name

ORTEDomainAppSubscriptionPatternDestroy -- destroys all subscription patterns

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternDestroy (ORTEDomain
* d);
```

Arguments

`d`

domain handle

Description

Destroys all subscription patterns which were specified previously by `ORTEDomainAppSubscriptionPatternAdd` function.

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainMgrCreate

Name

ORTEDomainMgrCreate -- create manager object in given domain

Synopsis

```
ORTEDomain * ORTEDomainMgrCreate (int domain,
ORTEDomainProp * prop, ORTEDomainAppEvents * events,
Boolean suspended);
```

Arguments

`domain`

-- undescribed --

`prop`

desired manager's properties

`events`

manager's event handlers or NULL

`suspended`

specifies whether threads of this manager should be started as well (ORTE_FALSE) or stay suspended (ORTE_TRUE). See `ORTEDomainStart` for details how to resume suspended threads

Description

Creates new manager object and sets its properties and events. Return handle to created object or NULL in case of any error.

ORTEDomainMgrDestroy

Name

ORTEDomainMgrDestroy -- destroy manager object

Synopsis

```
Boolean ORTEDomainMgrDestroy (ORTEDomain * d);
```

Arguments

`d`

manager object to be destroyed

Description

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEPublicationCreate

Name

ORTEPublicationCreate -- creates new publication

Synopsis

```
ORTEPublication * ORTEPublicationCreate (ORTEDomain * d,
const char * topic, const char * typeName, void * instance,
NtpTime * persistence, int strength, ORTESendCallBack
sendCallBack, void * sendCallBackParam, NtpTime *
sendCallBackDelay);
```

Arguments

`d`

pointer to application object

`topic`

name of topic

`typeName`

data type description

`instance`

output buffer where application stores data for publication

`persistence`

persistence of publication

`strength`

strength of publication

`sendCallBack`

pointer to callback function

`sendCallBackParam`

user parameters for callback function

`sendCallBackDelay`

periode for timer which issues callback function

Description

Creates new publication object with specified parameters. The `sendCallBack` function is called periodically with `sendCallBackDelay` periode. In strict reliable mode the `sendCallBack` function will be called only if there is enough room in transmitting queue in order to prevent any data loss. The `sendCallBack` function should prepare data to be published by this publication and place them into `instance` buffer.

Returns handle to publication object.

ORTEPublicationDestroy

Name

ORTEPublicationDestroy -- removes a publication

Synopsis

```
int ORTEPublicationDestroy (ORTEPublication * cstWriter);
```

Arguments

`cstWriter`

> handle to publication to be removed

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstWriter` is not valid cstWriter handle.

ORTEPublicationPropertiesGet

Name

ORTEPublicationPropertiesGet -- read properties of a publication

Synopsis

```
ORTEPublicationPropertiesGet (ORTEPublication * cstWriter,
ORTEPublProp * pp);
```

Arguments

`cstWriter`

> pointer to cstWriter object which provides this publication

`pp`

> pointer to ORTEPublProp structure where values of publication's properties will be stored

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstWriter` is not valid cstWriter handle.

ORTEPublicationPropertiesSet

Name

ORTEPublicationPropertiesSet -- set properties of a publication

Synopsis

```
int ORTEPublicationPropertiesSet (ORTEPublication *
cstWriter, ORTEPublProp * pp);
```

Arguments

`cstWriter`

> pointer to cstWriter object which provides this publication

`pp`

> pointer to ORTEPublProp structure containing values of publication's properties

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstWriter` is not valid publication handle.

ORTEPublicationGetStatus

Name

ORTEPublicationGetStatus -- removes a publication

Synopsis

```
int ORTEPublicationGetStatus (ORTEPublication * cstWriter,
ORTEPublStatus * status);
```

Arguments

`cstWriter`

> pointer to cstWriter object which provides this publication

`status`

> pointer to ORTEPublStatus structure

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `happ` is not valid publication handle.

ORTEPublicationSend

Name

ORTEPublicationSend -- force publication object to issue new data

Synopsis

```
int ORTEPublicationSend (ORTEPublication * cstWriter);
```

Arguments

`cstWriter`

publication object

Description

Returns ORTE_OK if successful.

ORTESubscriptionCreate

Name

ORTESubscriptionCreate -- adds a new subscription

Synopsis

```
ORTESubscription * ORTESubscriptionCreate (ORTEDomain * d,
SubscriptionMode mode, SubscriptionType sType, const char *
topic, const char * typeName, void * instance, NtpTime *
deadline, NtpTime * minimumSeparation, ORTERecvCallBack
recvCallBack, void * recvCallBackParam);
```

Arguments

`d`

pointer to ORTEDomain object where this subscription will be created

`mode`

see enum SubscriptionMode

`sType`

see enum SubscriptionType

`topic`

name of topic

`typeName`

name of data type

`instance`

pointer to output buffer

`deadline`

deadline

`minimumSeparation`

minimum time interval between two publications sent by Publisher as requested by Subscriber (strict - minumSep musi byt 0)

`recvCallBack`

callback function called when new Subscription has been received or if any change of subscription's status occures

`recvCallBackParam`

user parameters for `recvCallBack`

Description

Returns handle to Subscription object.

ORTESubscriptionDestroy

Name

ORTESubscriptionDestroy -- removes a subscription

Synopsis

```
int ORTESubscriptionDestroy (ORTESubscription * cstReader);
```

Arguments

`cstReader`

handle to subscriotion to be removed

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstReader` is not valid subscription handle.

ORTESubscriptionPropertiesGet

Name

ORTESubscriptionPropertiesGet -- get properties of a subscription

Synopsis

```
int ORTESubscriptionPropertiesGet (ORTESubscription *
cstReader, ORTESubsProp * sp);
```

Arguments

`cstReader`

handle to publication

`sp`

pointer to ORTESubsProp structure where properties of subscrition will be stored

ORTESubscriptionPropertiesSet

Name

ORTESubscriptionPropertiesSet -- set properties of a subscription

Synopsis

```
int ORTESubscriptionPropertiesSet (ORTESubscription *
cstReader, ORTESubsProp * sp);
```

Arguments

`cstReader`

handle to publication

`sp`

pointer to ORTESubsProp structure containing desired properties of the subscription

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstReader` is not valid subscription handle.

ORTESubscriptionWaitForPublications

Name

ORTESubscriptionWaitForPublications -- waits for given number of publications

Synopsis

```
int ORTESubscriptionWaitForPublications (ORTESubscription *
cstReader, NtpTime wait, unsigned int retries, unsigned int
noPublications);
```

Arguments

`cstReader`

handle to subscription

`wait`

time how long to wait

`retries`

number of retries if specified number of publications was not reached

`noPublications`

desired number of publications

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstReader` is not valid subscription handle or ORTE_TIMEOUT if number of retries has been exhausted..

ORTESubscriptionGetStatus

Name

ORTESubscriptionGetStatus -- get status of a subscription

Synopsis

```
int ORTESubscriptionGetStatus (ORTESubscription *
cstReader, ORTESubsStatus * status);
```

Arguments

`cstReader`

handle to subscription

`status`

pointer to ORTESubsStatus structure

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstReader` is not valid subscription handle.

ORTESubscriptionPull

Name

ORTESubscriptionPull -- read data from receiving buffer

Synopsis

```
int ORTESubscriptionPull (ORTESubscription * cstReader);
```

Arguments

`cstReader`

handle to subscription

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if `cstReader` is not valid subscription handle.

ORTETypeRegisterAdd

Name

ORTETypeRegisterAdd -- register new data type

Synopsis

```
int ORTETypeRegisterAdd (ORTEDomain * d, const char *
typeName, ORTETypeSerialize ts, ORTETypeDeserialize ds,
unsigned int gms);
```

Arguments

`d`

domain object handle

`typeName`

name of data type

`ts`

pointer to serialization function. If NULL data will be copied without any processing.

`ds`

deserialization function. If NULL data will be copied without any processing.

`gms`

maximum length of data (in bytes)

Description

Each data type has to be registered. Main purpose of this process is to define serialization and deserialization functions for given data type. The same data type can be registered several times, previous registrations of the same type will be overwritten.

Examples of serialization and deserialization functions can be found if contrib/shape/ortedemo_types.c file.

Returns ORTE_OK if new data type has been succesfully registered.

ORTETypeRegisterDestroyAll

Name

ORTETypeRegisterDestroyAll -- destroy all registered data types

Synopsis

```
int ORTETypeRegisterDestroyAll (ORTEDomain * d);
```

Arguments

`d`

domain object handle

Description

Destroys all data types which were previously registered by function `ORTETypeRegisterAdd`.

Return ORTE_OK if all data types has been succesfully destroyed.

ORTEVerbositySetOptions

Name

ORTEVerbositySetOptions -- set verbosity options

Synopsis

```
void ORTEVerbositySetOptions (const char * options);
```

Arguments

`options`

verbosity options

Description

There are 10 levels of verbosity ranging from 1 (lowest) to 10 (highest). It is possible to specify certain level of verbosity for each module of ORTE library. List of all supported modules can be found in linorte/usedSections.txt file. Every module has been aasigned with a number as can be seen in usedSections.txt file.

For instance

options = "ALL,7" Verbosity will be set to level 7 for all modules.

options = "51,7:32,5" Modules 51 (RTPSCSTWrite.c) will use verbosity level 7 and module 32 (ORTEPublicationTimer.c) will use verbosity level 5.

Maximum number of modules and verbosity levels can be changed in order to save some memory space if small memory footprint is neccessary. These values are defined as macros MAX_DEBUG_SECTIONS and MAX_DEBUG_LEVEL in file `include`/defines.h.

Return ORTE_OK if desired verbosity levels were successfuly set.

ORTEVerbositySetLogFile

Name

ORTEVerbositySetLogFile -- set log file

Synopsis

```
void ORTEVerbositySetLogFile (const char * logfile);
```

Arguments

`logfile`

  log file name

Description

Sets output file where debug messages will be writen to. By default these messages are written to stdout.

ORTEInit

Name

ORTEInit -- initialization of ORTE layer (musi se zavolat)

Synopsis

```
void ORTEInit ( void);
```

Arguments

`void`

  no arguments

ORTEAppSendThread

Name

ORTEAppSendThread -- resume sending thread in context of calling function.

Synopsis

```
void ORTEAppSendThread (ORTEDomain * d);
```

Arguments

`d`

domain object handle

Description

Sending thread will be resumed. This function never returns.

ORTESleepMs

Name

ORTESleepMs -- suspends calling thread for given time

Synopsis

```
void ORTESleepMs (unsigned int ms);
```

Arguments

`ms`

miliseconds to sleep

Macros

Table of Contents

SeqNumberCmp

Name

SeqNumberCmp -- comparison of two sequence numbers

Synopsis

```
SeqNumberCmp ( sn1, sn2);
```

Arguments

`sn1`

source sequential number 1

`sn2`

source sequential number 2

Return

1 if sn1 > sn2 -1 if sn1 < sn2 0 if sn1 = sn2

SeqNumberInc

Name

SeqNumberInc -- incrementation of a sequence number

Synopsis

```
SeqNumberInc ( res, sn);
```

Arguments

`res`

result

`sn`

sequential number to be incremented

Description

res = sn + 1

SeqNumberAdd

Name

SeqNumberAdd -- addition of two sequential numbers

Synopsis

```
SeqNumberAdd ( res, sn1, sn2);
```

Arguments

`res`

result

`sn1`

source sequential number 1

`sn2`

source sequential number 2

Description

res = sn1 + sn2

SeqNumberDec

Name

SeqNumberDec -- decrementation of a sequence number

Synopsis

```
SeqNumberDec ( res, sn);
```

Arguments

`res`

result

`sn`

sequential number to be decremented

Description

res = sn - 1

SeqNumberSub

Name

SeqNumberSub -- substraction of two sequential numbers

Synopsis

```
SeqNumberSub ( res, sn1, sn2);
```

Arguments

`res`

result

`sn1`

source sequential number 1

`sn2`

source sequential number 2

Description

res = sn1 - sn2

NtpTimeCmp

Name

NtpTimeCmp -- comparation of two NtpTimes

Synopsis

```
NtpTimeCmp ( time1, time2);
```

Arguments

`time1`

source time 1

`time2`

source time 2

Return value

1 if time 1 > time 2 -1 if time 1 < time 2 0 if time 1 = time 2

NtpTimeAdd

Name

NtpTimeAdd -- addition of two NtpTimes

Synopsis

```
NtpTimeAdd ( res, time1, time2);
```

Arguments

`res`

result

`time1`

source time 1

`time2`

source time 2

Description

res = time1 + time2

NtpTimeSub

Name

NtpTimeSub -- substraction of two NtpTimes

Synopsis

```
NtpTimeSub ( res, time1, time2);
```

Arguments

`res`

result

`time1`

source time 1

`time2`

source time 2

Description

res = time1 - time2

NtpTimeAssembFromMs

Name

NtpTimeAssembFromMs -- converts seconds and miliseconds to NtpTime

Synopsis

```
NtpTimeAssembFromMs ( time, s, msec);
```

Arguments

`time`

time given in NtpTime structure

`s`

seconds portion of given time

`msec`

miliseconds portion of given time

NtpTimeDisAssembToMs

Name

NtpTimeDisAssembToMs -- converts NtpTime to seconds and miliseconds

Synopsis

```
NtpTimeDisAssembToMs ( s, msec, time);
```

Arguments

`s`

seconds portion of given time

`msec`

miliseconds portion of given time

`time`

time given in NtpTime structure

NtpTimeAssembFromUs

Name

NtpTimeAssembFromUs -- converts seconds and useconds to NtpTime

Synopsis

```
NtpTimeAssembFromUs ( time, s, usec);
```

Arguments

`time`

time given in NtpTime structure

`s`

seconds portion of given time

`usec`

microseconds portion of given time

NtpTimeDisAssembToUs

Name

NtpTimeDisAssembToUs -- converts NtpTime to seconds and useconds

Synopsis

`NtpTimeDisAssembToUs ( s, usec, time);`

Arguments

`s`

seconds portion of given time

`usec`

microseconds portion of given time

`time`

time given in NtpTime structure

Domain2Port

Name

Domain2Port -- converts Domain value to IP Port value

Synopsis

`Domain2Port ( d, p);`

Arguments

`d`

domain

`p`

port

Domain2PortMulticastUserdata

Name

Domain2PortMulticastUserdata -- converts Domain value to userdata IP Port value

Synopsis

`Domain2PortMulticastUserdata ( d, p);`

Arguments

`d`

domain

`p`

port

Domain2PortMulticastMetatraffic

Name

Domain2PortMulticastMetatraffic -- converts Domain value to metatraffic IP Port value

Synopsis

```
Domain2PortMulticastMetatraffic ( d, p);
```

Arguments

`d`

domain

`p`

port

ORTE Implementation Issues

ORTE is network middleware for distributed, real-time application development that uses the real-time, publish-subscribe model. The middleware is available for a variety of platforms including RTAI, RTLinux, Windows, and a several versions of Unix. The compilation system is mainly based on autoconf.

ORTE is middleware composed of a database, and tasks. On the top of ORTE architecture is application interface (API). By using API users should write self application. The tasks perform all of the message addressing serialization/deserialization, and transporting. The ORTE components are shown in Figure 1-5

Figure 1-5. ORTE Architecture

The RTPS protocol defines two kinds of Applications:

- **Manager:** The manager is a special Application that helps applications automatically discover each other on the Network.

- **ManagedApplication:** A ManagedApplication is an Application that is managed by one or more Managers. Every ManagedApplication is managed by at least one Manager.

The manager is mostly designed like separate application. In RTPS architecture is able to create application which contains manager and managedapplication, but for easy managing is better split both. The ORTE contains a separate instance of manager located in directory `orte/manager`.

The manager is composed from five kinds of objects:

- **WriterApplicationSelf:** through which the Manager provides information about its own parameters to Managers on other nodes.
- **ReaderManagers:** CSTReader through which the Manager obtains information on the state of all other Managers on the Network.
- **ReaderApplications:** CSTReader which is used for the registration of local and remote managedApplications.
- **WriterManagers:** CSTWriter through which the Manager will send the state of all Managers in the Network to all its managees.
- **WriterApplications:** CSTWriter through which the Manager will send information about its managees to other Managers in the Network.

A Manager that discovers a new ManagedApplication through its readerApplications must decide whether it must manage this ManagedApplication or not. For this purpose, the attribute managerKeyList of the Application is used. If one of the ManagedApplication's keys (in the attribute managerKeyList) is equal to one of the Manager's keys, the Manager accepts the Application as a managee. If none of the keys are equal, the managed application is ignored. At the end of this process all Managers have discovered their managees and the ManagedApplications know all Managers in the Network.

The managedApplication is composed from seven kinds of objects:

- **WriterApplicationSelf:** a CSTWriter through which the ManagedApplication registers itself with the local Manager.
- **ReaderApplications:** a CSTReader through which the ManagedApplication receives information about another ManagedApplications in the network.
- **ReaderManagers:** a CSTReader through which the ManagedApplication receives information about Managers.
- **WriterPublications:** CSTWriter through which the Manager will send the state of all Managers in the Network to all its managees.
- **ReaderPublications:** a Reader through which the Publication receives information about Subscriptions.
- **WriterSubscriptions:** a Writer that provides information about Subscription to Publications.

- **ReaderSubscriptions:** a Reader that receives issues from one or more instances of Publication, using the publish-subscribe service.

The ManagedApplication has a special CSTWriter writerApplicationSelf. The Composite State (CS) of the ManagedApplication's writerApplicationSelf object contains only one NetworkObject - the application itself. The writerApplicationSelf of the ManagedApplication must be configured to announce its presence repeatedly and does not request nor expect acknowledgments.

The ManagedApplications now use the CST Protocol between the writerApplications of the Managers and the readerApplications of the ManagedApplications in order to discover other ManagedApplications in the Network. Every ManagedApplication has two special CSTWriters, writerPublications and writerSubscriptions, and two special CSTReaders, readerPublications and readerSubscriptions.

Once ManagedApplications have discovered each other, they use the standard CST protocol through these special CSTReaders and CSTWriter to transfer the attributes of all Publications and Subscriptions in the Network.

The ORTE stores all data in local database per application. There isn't central store where are data saved. If an application comes into communication, than will be created local mirror of all applications parameters. Parts of internal structures are shown in Figure 1-6.

Figure 1-6. ORTE Internal Attributes

Following example shows communication between two nodes (N1, N2). There are applications running on each node - MA1.2 on node N1 and MA2.1, MA2.2 on node N2. Each node has it own manager (M1, M2). The example shows, what's happen when a new application comes into communication (MA1.1).

1. MA1.1 introduces itself to local manager M1
2. M1 sends back list of remote managers Mx and other local applications MA1.x
3. MA1.1 is introduced to all Mx by M1
4. All remote MAs are reported now to M1.1
5. MA1.1 is queried for self services (publishers and subscriberes) from others MAx.
6. MA1.1 asks for services to others MAx.
7. All MAs know information about others.

The corresponding publishers and subscribers with matching Topic and Type are connected and starts their data communication.

Figure 1-7. RTPS Communication among Network Objects

ORTE Examples

This chapter expect that you are familiar with RTPS communication architecture described in the Section called *ORTE Description*.

Publications can offer multiple reliability policies ranging from best-efforts to strict (blocking) reliability. Subscription can request multiple policies of desired reliability and specify the relative precedence of each policy. Publications will automatically select among the highest precedence requested policy that is offered by the publication.

- **BestEffort:** This reliability policy is suitable for data that are sending with a period. There are no message resending when a message is lost. On other hand, this policy offer maximal predictable behaviour. For instance, consider a publication which send data from a sensor (pressure, temperature, ...).

  Figure 1-8. Periodic Snapshots of a BestEffort Publisher

- **StrictReliable:** The ORTE supports the reliable delivery of issues. This kind of communication is used where a publication want to be sure that all data will be delivered to subscriptions. For instance, consider a publication which send commands.

  Command data flow requires that each instruction in the sequence is delivered reliably once and only once. Commands are often not time critical.

BestEffort Communication

Before creating a Publication or Subscription is necessary to create a domain by using function `ORTEDomainAppCreate`. The code should looks like:

```
int main(int argc, char *argv[])
{
  ORTEDomain *d = NULL;
  ORTEBoolean suspended= ORTE_FALSE;

  ORTEInit();

  d = ORTEDomainAppCreate(ORTE_DEFAUL_DOMAIN, NULL, NULL, suspended);
  if (!d)
  {
    printf("ORTEDomainAppCreate failed\n");
    return -1;
  }
}
```

The ORTEDomainAppCreate allocates and initializes resources that are needed for communication. The parameter `suspended` says if ORTEDomain takes suspend communicating threads. In positive case you have to start threads manually by using `ORTEDomainStart`.

Next step in creation of a application is registration serialization and deserialization routines for the specific type. You can't specify this functions, but the incoming data will be only copied to output buffer.

```
ORTETypeRegisterAdd(d, "HelloMsg", NULL, NULL, 64);
```

To create a publication in specific domain use the function ORTEPublicationCreate.

```
char instance2send[64];
NtpTime persistence, delay;

NTPTIME_BUILD(persistence, 3);  /* this issue is valid for 3 seconds */
NTPTIME_DELAY(delay, 1);        /* a callback function will be called
every 1 second */
p = ORTEPublicationCreate( d,
                         "Example HelloMsg",
                         "HelloMsg",
                         &instance2Send,
                         &persistence,
                         1,
                         sendCallBack,
                         NULL,
                         &delay);
```

The callback function will be then called when a new issue from publisher has to be sent. It's the case when you specify callback routine in `ORTEPublicationCreate`. When there isn't a routine you have to send data manually by call function `ORTEPublicationSend`. This option is useful for sending periodic data.

```
void sendCallBack(const ORTESendInfo *info, void *vinstance, void
*sendCallBackParam)
{
  char *instance = (char *) vinstance;
  switch (info->status)
  {
    case NEED_DATA:
      printf("Sending publication, count %d\n", counter);
      sprintf(instance, "Hello world (%d)", counter++);
      break;

    case CQL:  //criticalQueueLevel has been reached
      break;
  }
}
```

Subscribing application needs to create a subscription with publication's Topic and TypeName. A callback function will be then called when a new issue from publisher will be received.

```
ORTESubscription *s;
NtpTime deadline, minimumSeparation;
```

```
NTPTIME_BUILD(deadline, 20);
NTPTIME_BUILD(minimumSeparation, 0);
p = ORTESubscriptionCreate( d,
                            IMMEDIATE,
                            BEST_EFFORTS,
                            "Example HelloMsg",
                            "HelloMsg",
                            &instance2Recv,
                            &deadline,
                            &minimumSeparation,
                            recvCallBack,
                            NULL);
```

The callback function is shown in the following example:

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void
*recvCallBackParam)
{
  char *instance = (char *) vinstance;
  switch (info->status)
  {
    case NEW_DATA:
      printf("%s\n", instance);
      break;

    case DEADLINE:  //deadline occurred
      break;
  }
}
```

Similarly examples are located in ORTE subdirectory `orte/examples/hello`. There are demonstrating programs how to create an application which will publish some data and another application, which will subscribe to this publication.

Reliable communication

The reliable communication is used especially in situations where we need guarantee data delivery. The ORTE supports the inorder delivery of issues with built-in retry mechanism

The creation of reliable communication starts like besteffort communication. Difference is in creation a subscription. Third parameter is just only changed to STRICT_RELIABLE.

```
ORTESubscription *s;
NtpTime deadline, minimumSeparation;
```

```
NTPTIME_BUILD(deadline, 20);
NTPTIME_BUILD(minimumSeparation, 0);
p = ORTESubscriptionCreate( d,
                          IMMEDIATE,
                          STRICT_RELIABLE,
                          "Example HelloMsg",
                          "HelloMsg",
                          &instance2Recv,
                          &deadline,
                          &minimumSeparation,
                          recvCallBack,
                          NULL);
```

Note:

Strict reliable subscription must set minimumSeparation to zero! The middleware can't guarantee that the data will be delivered on first attempt (retry mechanism).

Sending of a data is blocking operation. It's strongly recommended to setup sending queue to higher value. Default value is 1.

```
ORTEPublProp *pp;

ORTEPublicationPropertiesGet(publisher,pp);
pp->sendQueueSize=10;
pp->criticalQueueLevel=8;
ORTEPublicationPropertiesSet(publisher,pp);
```

An example of reliable communication is in ORTE subdirectory `orte/examples/reliable`. There are located a strictreliable subscription and publication.

Serialization/Deserialization

Actually the ORTE doesn't support any automatic creation of serialization/deserializaction routines. This routines have to be designed manually by the user. In next is shown, how should looks both for the structure BoxType.

```
typedef struct BoxType {
    int32_t  color;
    int32_t  shape;
} BoxType;

void
BoxTypeSerialize(ORTECDRStream *cdr_stream, void *instance) {
  BoxType  *boxType=(BoxType*)instance;

  *(int32_t*)cdr_stream->bufferPtr=boxType->color;
  cdr_stream->bufferPtr+=sizeof(int32_t);
  *(int32_t*)cdr_stream->bufferPtr=boxType->shape;
  cdr_stream->bufferPtr+=sizeof(int32_t);
```

```
}

void
BoxTypeDeserialize(ORTECDRStream *cdr_stream, void *instance) {
  BoxType  *boxType=(BoxType*)instance;

  boxType->color=*(int32_t*)cdr_stream->bufferPtr;
  cdr_stream->bufferPtr+=sizeof(int32_t);
  boxType->shape=*(int32_t*)cdr_stream->bufferPtr;
  cdr_stream->bufferPtr+=sizeof(int32_t);
}
```

When we have written a serialization/deserialization routine we need to register this routines to midleware by function `ORTETypeRegisterAdd`

```
  ORTETypeRegisterAdd(
                domain,
                "BoxType",
                BoxTypeSerialize,
                BoxTypeDeserialize,
                sizeof(BoxType));
```

The registration must be called before creation a publication or subscription. Normally is `ORTETypeRegisterAdd` called immediately after creation of a domain (`ORTEDomainCreate`).

All of codes are part of the Shapedemo located in subdirectory `orte/contrib/shape`.

ORTE Tests

There were not any serious tests performed yet. Current version has been intensively tested against reference implementation of the protocol. Results of these test indicate that ORTE is fully interoperable with implementation provided by another vendor.

ORTE Usage Information

Installation and Setup

In this chapter is described basic steps how to makes installation and setup process of the ORTE. The process includes next steps:

1. Downloading the ORTE distribution
2. Compilation
3. Installing the ORTE library and utilities
4. Testing the installation

Note:

On windows systems we are recommend to use Mingw or Cygwin systems. The ORTE support also MSVC compilation, but this kind of installation is not described here.

Downloading

The ORTE component can be obtained from OCERA SourceForge web page (http://www.sf.net/projects/ocera/). Here is the component located also in self distribution branch as well as in OCERA distribution. Before developing any application check if there is a new file release.

The CVS version of ORTE repository can be checked out be anonymous (pserver) CVS with the following commands.

```
cvs -d:pserver:anonymous@cvs.ocera.sourceforge.net:/cvsroot/ocera login
cvs -z3 -d:pserver:anonymous@cvs.ocera.sourceforge.net:/cvsroot/ocera
co ocera/components/comm/eth/orte/
```

Attention, there is developing version and can't be stable!

Compilation

Before the compilation process is necessary to prepare the source. Create a new directory for ORTE distribution. We will assume name of this directory `/orte` for Linux case. Copy or move downloaded ORTE sources to `/orte` (assume the name of sources `orte-0.2.3.tar.gz`). Untar and unzip this files by using next commands:

```
gunzip orte-0.2.3.tar.gz
tar xvf orte-0.2.3.tar
```

Now is the source prepared for compilation. Infrastructure of the ORTE is designed to support GNU make (needs version 3.81) as well as autoconf compilation. In next we will continue with description of autoconf compilation, which is more general. The compilation can follows with commands:

```
mkdir build
cd build
../configure
make
```

This is the case of outside autoconf compilation. In directory `build` are all changes made over ORTE project. The source can be easy move to original state be removing of directory `build`.

Installing

The result of compilation process are binary programs and ORTE library. For the next developing is necessary to install this result. It can be easy done be typing:

```
make install
```

All developing support is transferred into directories with direct access of design tools.

name	***target path***
ortemanager, orteping, ortespy	`/usr/local/bin`
library	`/usr/local/lib`
include	`/usr/local/include`

The installation prefix `/usr/local/` can be changed during configuration. Use command **../configure --help** for check more autoconf options.

Testing the Installation

To check of correct installation of ORTE open three shells.

1. In first shell type

   ```
   ortemanager
   ```

2. In second shell type

   ```
   orteping -s
   ```

   This command will invoked creation of a subscription. You should see:

   ```
   deadline occurred
   deadline occurred
   ...
   ```

3. In third shell type

   ```
   orteping -p
   ```

   This command will invoked creation of a publication. You should see:

   ```
   sent issue 1
   sent issue 2
   sent issue 3
   sent issue 4
   ...
   ```

If the ORTE installation is properly, you will see incoming messages in second shell (**orteping -s**).

```
received fresh issue 1
received fresh issue 2
received fresh issue 3
received fresh issue 4
...
```

It's sign, that communication is working correctly.

The ORTE Manager

A manager is special application that helps applications automatically discover each other on the Network. Each time an object is created or destroyed, the manager propagate new information to the objects that are internally registered.

Every application precipitate in communication is managed by least one manager. The manager should be designed like separated application as well as part of designed application.

Figure 1-9. Position of Managers in RTPS communication

The ORTE provides one instance of a manager. Name of this utility is `ortemanager` and is located in directory `orte/ortemanager`. Normally is necessary to start `ortemanager` manually or use a script on UNIX systems. For Mandrake and Red-hat distribution is this script created in subdirectory `rc`. Windows users can install `ortemanager` like service by using option `/install_service`.

Note:

Don't forget to run a manager (ortemanager) on each RTPS participate node. During live of applications is necessary to be running this manager.

Example of Usage ortemanager

Table of Contents

Each manager has to know where are other managers in the network. Their IP addresses are therefore specified as IPAddressX parameters of ortemanager. All managers participate in one kind of communication use the same domain number. The domain number is transferred to port number by equation defined in RTPS specification (normally domain 0 is transferred to 7400 port).

Let's want to distribute the RTPS communication of nodes with IP addresses 192.168.0.2 and 192.168.0.3. Private IP address is 192.168.0.1. The ortemanager can be execute with parameters:

```
ortemanager -p 192.168.0.2:192.168.0.3
```

To communicate in different domain use (parameter -d):

```
ortemanager -d 1 -p 192.168.0.2:192.168.0.3
```

Very nice feature of ortemanager is use event system to inform of creation/destruction objects (parameter -e).

```
ortemanager -e -p 192.168.0.2:192.168.0.3
```

Now, you can see messages:

```
[smolik@localhost smolik]$ortemanager -e -p 192.168.0.2:192.168.0.3
manager 0xc0a80001-0x123402 was accepted
application 0xc0a80002-0x800301 was accepted
application 0xc0a80002-0x800501 was accepted
application 0xc0a80002-0x800501 was deleted
manager 0xc0a80001-0x123402 was deleted
```

ortemanager

Name

ortemanager -- the utility for discovery others applications and managers on the network

Synopsis

ortemanager [-d *domain*] [-p *ip addresses*] [-k *ip addresses*] [-R *refresh*] [-P *purge*] [-D] [-E *expiration*] [-e] [-v *verbosity*] [-l *filename*] [-V] [-h]

Description

Main purpose of the utility **ortemanager** is debug and test ORTE communication.

OPTIONS

-d --domain

> The number of working ORTE domain. Default is 0.

-p --peers

> The IP addresses parsipiates in RTPS communication. See the Section called *The ORTE Manager* in Chapter 1 for example of usage.

-R --refresh

The refresh time in manager. Default 60 seconds.

`-P --purge`

The searching time in local database for finding expired application. Default 60 seconds.

`-E --expiration`

Expiration time in other applications.

`-m --minimumSeparation`

The minimum time between two issues.

`-v --verbosity`

Set verbosity level.

`-l --logfile`

All debug messages can be redirect into specific file.

`-V --version`

Print the version of **ortemanager**.

`-h --help`

Print usage screen.

Simple Utilities

Table of Contents

The simple utilities can be found in the `orte/examples` subdirectory of the ORTE source subtree. These utilities are useful for testing and monitoring RTPS communication.

The utilities provided directly by ORTE are:

orteping

the utility for easy creating of publications and subscriptions.

ortespy

monitors issues produced by other application in specific domain.

orteping

Name

orteping -- the utility for debugging and testing of ORTE communication

Synopsis

orteping [`-d` *`domain`*] [`-p`] [`-S` *`strength`*] [`-D` *`delay`*] [`-s`] [`-R` *`refresh`*] [`-P` *`purge`*] [`-E` *`expiration`*] [`-m` *`minimumSeparation`*] [`-v` *`verbosity`*] [`-q`] [`-l` *`filename`*] [`-V`] [`-h`]

Description

Main purpose of the utility **orteping** is debug and test ORTE communication.

OPTIONS

`-d --domain`

The number of working ORTE domain. Default is 0.

`-p --publisher`

Create a publisher with Topic - Ping and Type - PingData. The publisher will publish a issue with period by parameter delay.

`-s --strength`

Setups relative weight against other publishers. Default is 1.

`-D --delay`

The time between two issues. Default 1 second.

`-s --subscriber`

Create a subscriber with Topic - Ping and Type - PingData.

`-R --refresh`

The refresh time in manager. Default 60 seconds.

`-P --purge`

The searching time in local database for finding expired application. Default 60 seconds.

`-E --expiration`

Expiration time in other applications.

`-m --minimumSeparation`

The minimum time between two issues.

`-v --verbosity`

Set verbosity level.

`-q --quite`

Nothing messages will be printed on screen. It can be useful for testing maximal throughput.

`-l --logfile`

All debug messages can be redirect into specific file.

`-V --version`

Print the version of **orteping**.

`-h --help`

Print usage screen.

ortespy

Name

ortespy -- the utility for monitoring of ORTE issues

Synopsis

orteping [`-d` *domain*] [`-v` *verbosity*] [`-R` *refresh*] [`-P` *purge*] [`-e` *expiration*] [`-l` *filename*] [`-V`] [`-h`]

Description

Main purpose of the utility **ortespy** is monitoring data traffic between publications and subscriptions.

OPTIONS

`-d --domain`

The number of working ORTE domain. Default is 0.

`-v --verbosity`

Set verbosity level.

`-R --refresh`

The refresh time in manager. Default 60 seconds.

`-P --purge`

Create publisher

`-e --expiration`

Expiration time in other applications.

`-l --logfile`

All debug messages can be redirect into specific file.

`-V --version`

Print the version of **orteping**.

`-h --help`

Print usage screen.

Fault-Tolerance components

by A. Lanusse and P. Vanuxeem

Degraded Mode Management

Description

The Degraded Mode Management Framework has been designed to offer transparent management of dynamic reconfiguration of applications on detection of faulty situations.

Continuity of service is maintained in case of partial failure through graceful degradation management. The Degraded Mode Management Framework offers an integrated set of tools and components.

Design/build tool. The Ftbuilder permits the specification of application real-time constraints, different possible application modes along with related transition conditions.

From this specification, code generation is achieved in order to instantiate internal control data-bases of run-time components (ftappmon and ftcontroller) and to provide application model files.

ftappmon . The ftappmon component is devoted to global application handling. It is in charge of overall application setup and of reconfiguration decisions. It contains information on different possible application modes and on transition conditions. When an error is detected and notified by the ftcontroller, the ftappmon analyzes the event and issues reconfiguration orders (stop, awake, switch ft-task behavior) towards the ftcontroller.

ftcontroller . The ftcontroller is in charge of the direct control of application threads. It provides error detection (kill or timing error) and notification (towards ftappmon) and executes reconfiguration orders at task level.

API. A specific but reduced API has been defined for manipulating so called ft-tasks. Three main functions: ft_task_init(), ft_task_create, ft_task_end(). These ft-tasks are actually encapsulation of periodic RTLinux tasks. Other functions are used to init internal data-bases. Besides these specific functions , application developers can use any RTLinux programming feature.

The framework currently available relies on a simplified model of applications. According to this model only simple applications with periodic tasks are handled at the moment. Though these are indeed quite restrictive hypotheses, they represent a large range of effective current real-time embedded applications. Please refer to users's guide for the description of the main characteristics of applications handled and restrictions to applicability.

Usage

Degraded Mode Management configuration process takes three steps:

1. **OS Type Selection** : Soft and Hard real-time must be chosen and some sub-options must be checked (see below).
2. **Components Selection** : FT components + Hard Realtime + Degraded Mode Management.
3. **Core kernel Scheduling features selection** : Priority or EDF scheduling. Only EDF scheduling will offer support for deadline miss detection.

1. OS-Type Selection

FT Degraded Mode Management support requires the selection of Hard and Soft real-time in the OS type section. This enables hard-realtime standard RTLinux configuration options.

General dependencies of FT components are illustrated in the following figure.

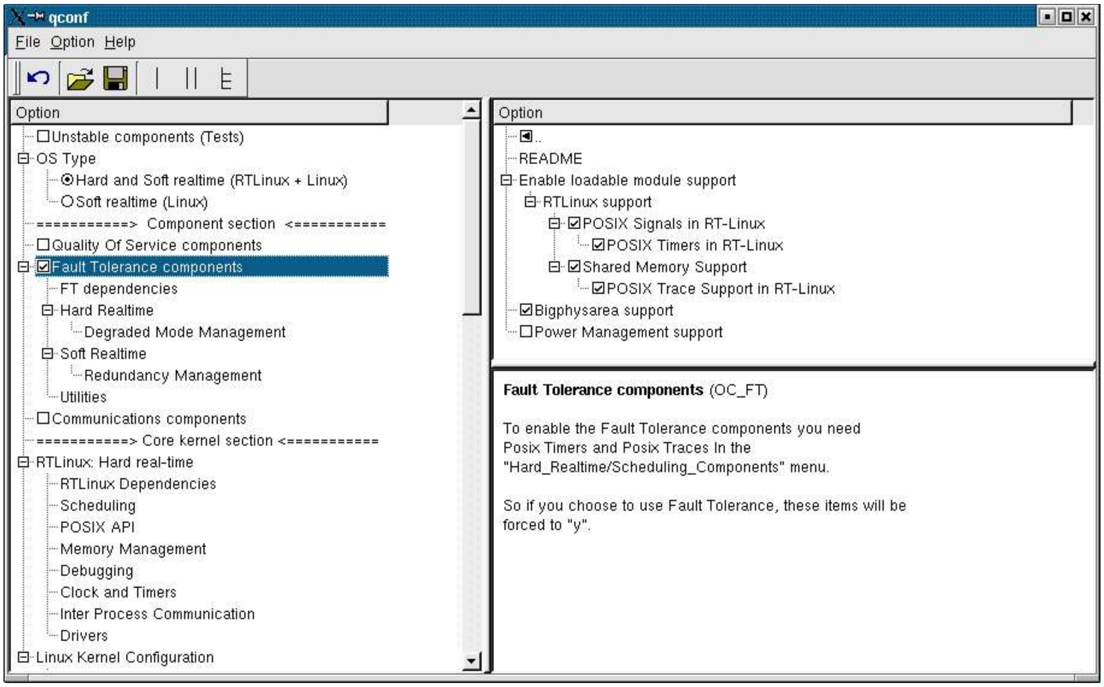


Figure 1 : FT Degraded Mode Management Configuration step1

Required options are : loadable module support, RT-Linux support including POSIX Signals and POSIX Timers, Shared Memory support, POSIX Trace Support in RTLinux., BigPhysarea support.

Note that POSIX Trace Support is mandatory for FT Degraded Mode Management components.

Power Management support should not be selected.

Normally all these options are posted correctly in the standard OCERA distribution, so just check.

2. FT Degraded Mode management components selection

FT components for degraded Mode Management have to be selected. If you select the Framework, these are set automatically, just verify. Two components are necessary, the **FT Controller** and the **FT Application Monitor,** they must be selected together. At compilation time they will be merged into one single module named **ftappmonctrl.**

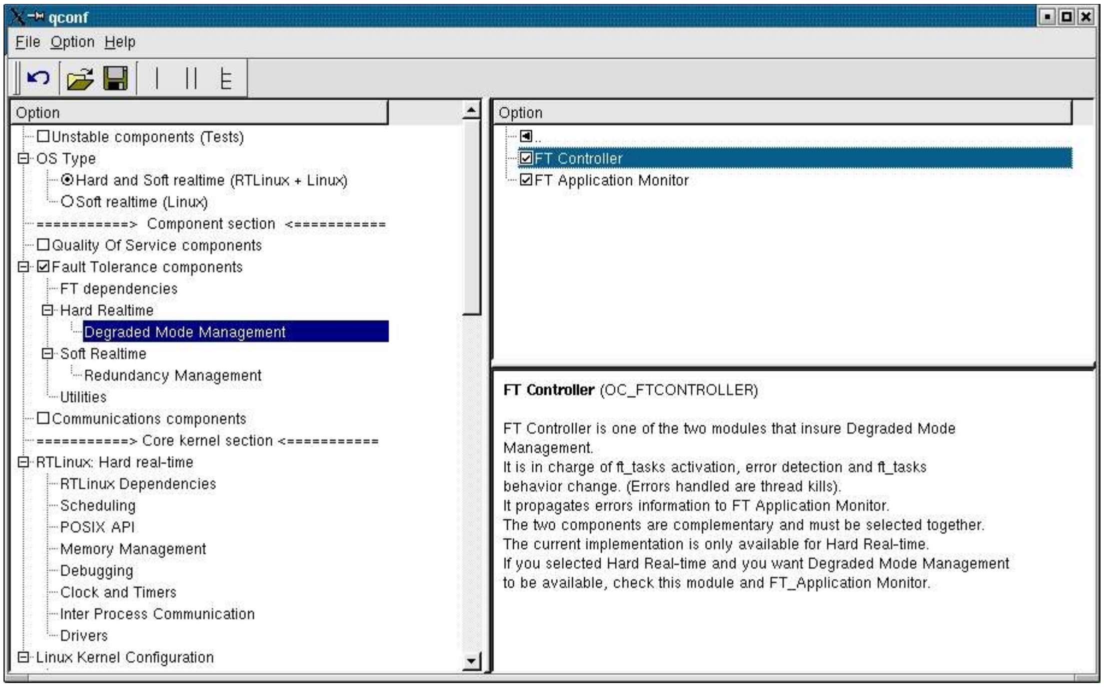
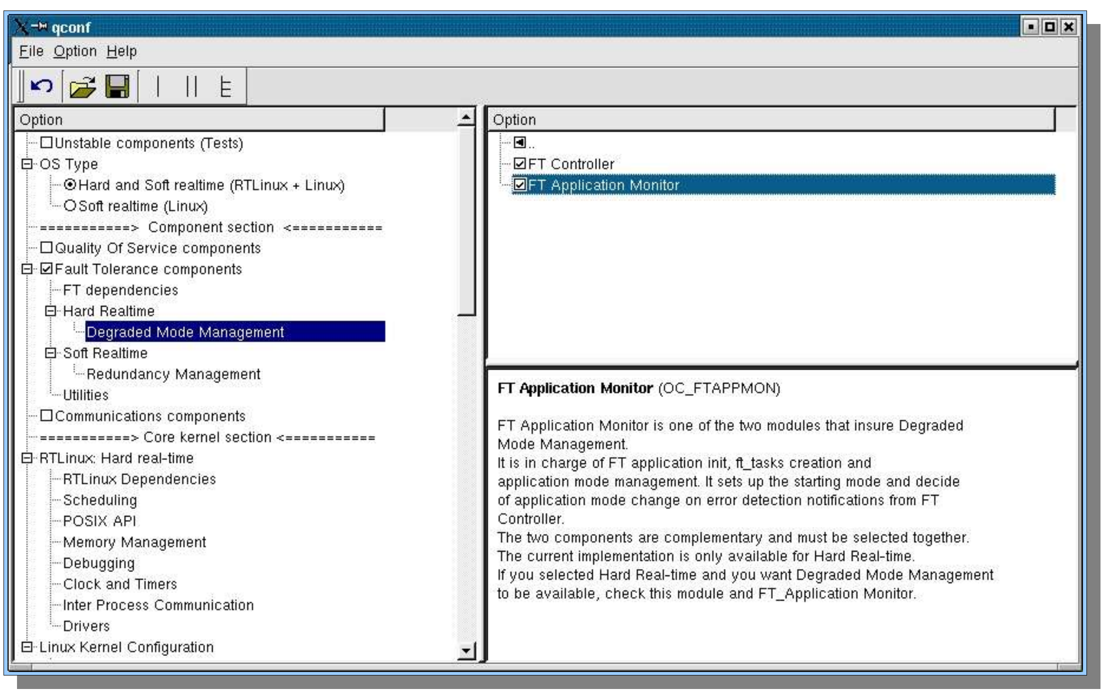


Figure 2 : FT Degraded Mode Management Configuration step2

The FTController is selected.



The FT Application Monitor is selected.

3. Core kernel Scheduling features selection

The last configuration step concerns the choice of scheduling policy. The functioning of the **Degraded Mode Management** can follow several types of scheduling, the facilities offered will depend on the choice done at configuration.

The error detection mechanism can handle two types of errors:

- Pthread_kill detection works with priority based or EDF scheduling policies;
- Timing errors (deadline miss) detection can only be detected if EDF scheduling is selected and related option Dealline Miss Detection.

Remark: Once a scheduling policy has been chosen during the configuration all the ft_tasks will be scheduled according to this policy.

Priority based scheduling

Standard prioity based scheduling can be configured by selecting the **Application defined scheduler** option in the the Scheduling section of the RTLinux Hard real-time part as indicated bellow. In this case the only type of errors to be detected are Pthread-kill events.

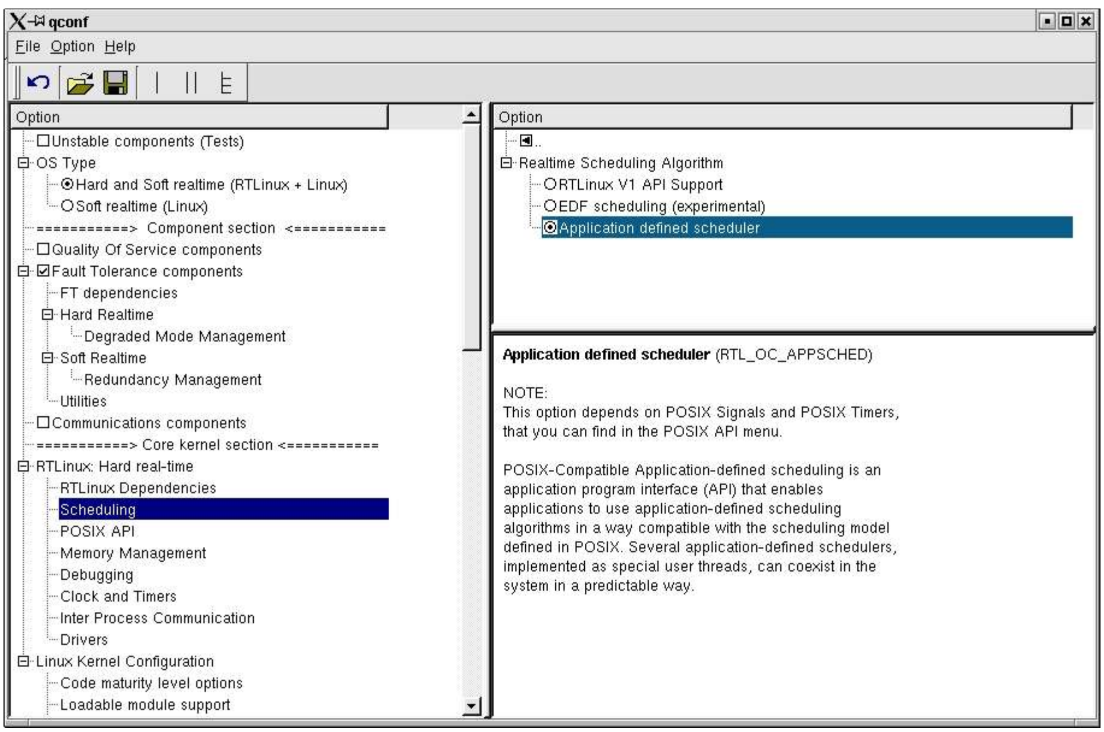


Figure 3 : FT Degraded Mode Management Configuration step3 – Priority scheduler

The **Application defined scheduler** component provides support for several types of scheduling policies defined above the RTLinux kernel itself. By default, the scheduling policy is based on priorities, which is the configuration that we must use in this case. For further details see section Scheduling.

EDF scheduling

This version of scheduling is implemented directly at RTLinux kernel level and not above as it is the case with the **Application Defined scheduler**.

So if you want to detect both timing errors and pthread_kill events, you should select EDF + Deadline-miss detection, as shown hereunder.

Remark: If you select only EDF, scheduling policy applied will be EDF but only pthread_kill events will be detected, there will be no emission of Deadline-miss event.

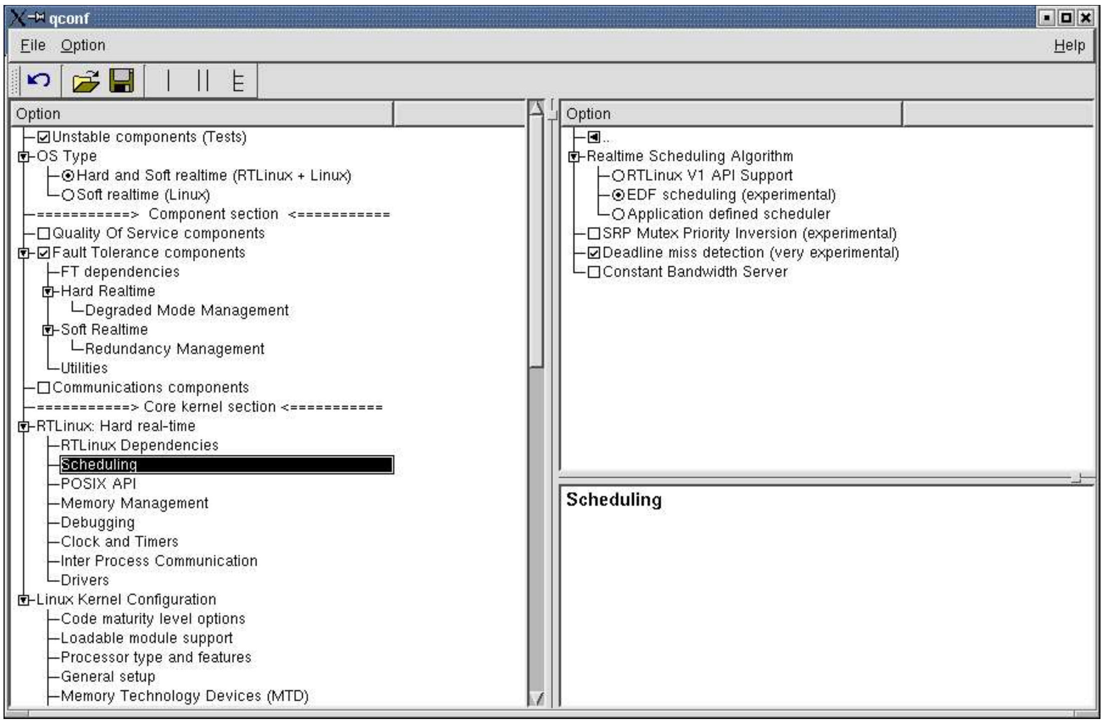


Figure 4 : FT Degraded Mode Management Configuration step3 - EDF + DLM scheduler

The configuration EDF scheduling with Deadline-miss detection (DLM) is still an experimental functionality. For the moment, support for EDF+DLM+SRP is not offered.

Programming Interface (API)

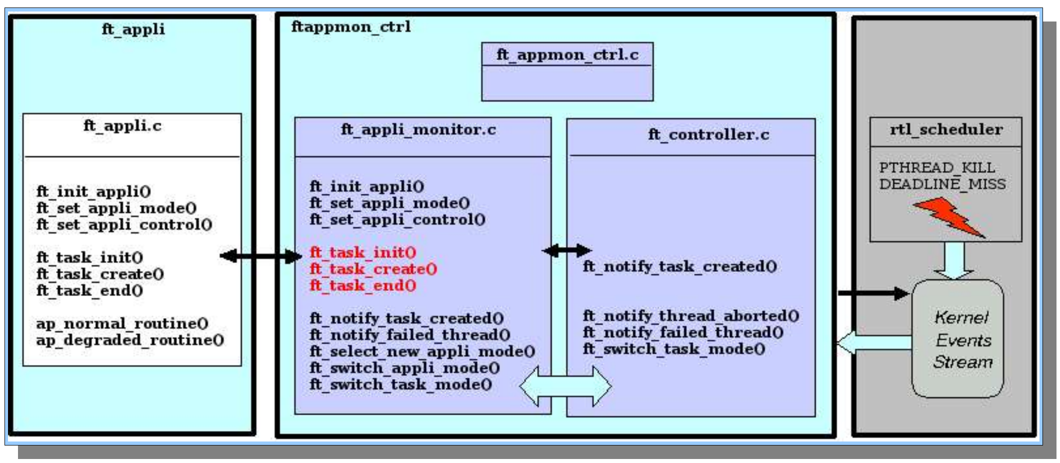
The **ftappmon** component offers to the application developer an application programming interface, named **FT-API,** that is restricted to very few functions **ft_task_init()**, **ft_task_create()**, **ft_task_end()**.

It offers also some additional FT-API functions used to instantiate the FT application model: **ft_init_appli(), ft_set_appli_mode(), ft_set_appli_control()**. Actually calls to these functions are automatically generated by the FT-Builder tool into a specific file devoted to init application configuration data structures. So these last functions may be considered as transparent to the application developer.

The **ftappmon** component has also an internal API for interactions with the **ftcontroller** component for the notification of failed thread.

The **ftcontroller** has an API for the **ftappmon** component for the notification of the ft-task created and for the switch of ft-task behavior. The **ftcontroller** has an API that could be used by the scheduler mainly to notify events to the **ftcontroller**.

These different intefaces are illustrated in the following figure.



The detailed definition of prototypes necessary at the application level is provided in files:

`${OCERA_DIR/components/ft/ftappmon/include/`

`ft_api_appmon_appli.h`

`ft_api_common.h`

```
// FT init appli
int ft_init_appli
 (
  int ft_appli_tasks_max_nb,      // Number max of appli tasks
  int ft_appli_modes_max_nb,      // Number max of appli modes
  FT_Appli_Mode ft_cur_appli_mode // Current appli mode
 );

// FT set_appli_control
extern int ft_set_appli_control
 (
  char *ft_appli_mode_name,
  FT_Event_Task_Appli_Mode_Elt *ft_evt_task_app_modes_tab
 );
```

```
// FT set appli mode
extern int ft_set_appli_mode
 (
  char *ft_appli_mode_name,
  FT_Task_Behavior_Elt *ft_task_behaviors_tab
 );
```

```
// FT task init
extern int ft_task_init
 (
  char *ft_task_name,
  void (*ft_normal_routine)(void*),
  void (*ft_degraded_routine)(void*),
  int ft_normal_param,
  int ft_degraded_param,
  FT_sched_param ft_normal_sched_param,
  FT_sched_param ft_degraded_sched_param
#ifdef FT_PREALLOCATED_THREAD_STACK
  /* fxr: stack address and size */
  ,
  void *ft_normal_stackaddr,
  int   ft_normal_stacksize,
  void *ft_degraded_stackaddr,
  int   ft_degraded_stacksize
#endif
 );

// FT task create
extern int ft_task_create
 (
  int ft_task_id,
  FT_task_behavior ft_task_behavior
);

// FT task end
extern int ft_task_end
 (
  int ft_task_id
 );
```

```
/*****************************************************************************/
/*                                                              */
/* FT API Controller : Other Functions                                  */
/*                                                              */
/*****************************************************************************/
extern int ft_appli_monitor_stop(void);
extern int ft_controller_stop(void);
```

Important types and structures

FT_task_behavior

```
// FT-Task behavior
typedef enum {
  FT_TASK_BEHAVIOR_NOT_DEFINED, // FT-task behavior is not defined
  FT_TASK_NOT_STARTED,          // FT_task has a not started behavior :
                                // normal and degraded thread are suspended
  FT_TASK_NORMAL,               // FT_task has a normal behavior :
                                // normal thread is started, degraded thread is suspended
  FT_TASK_DEGRADED,             // FT_task has a degraded behavior :
                                // normal thread is killed, degraded thread is waked up
  FT_TASK_TERMINATED            // FT_task has a terminated behavior :
                                // normal and degraded threads are killed
} FT_task_behavior;
```

FT_sched_param

```
// FT_sched_param
typedef struct {
  int prio;           // priority
  hrtime_t start_time;// start time (relative)
  hrtime_t period;    // period (relative)
                      // period <= 0 : aperiodic task
                      // period > 0  : periodic task
  hrtime_t deadline;  // deadline (relative)
  hrtime_t duration;  // estimated duration
  char algo[10];      // scheduling algorithm : either PRIO, EDF, EDF+SRP
  int hard_soft_deadline;  // Hard (=0) or soft (=1) deadline
  int count_soft_deadline; // Count value for soft deadline
} FT_sched_param;
```

Example

The following example named ftnormal+kill is a simple example that illustratres the functionning of dynamic reconfiguration on detection of Pthread_kill event.

The application consists of two tasks. The example can be found in :

${OCERA_DIR/ components/ft/ftcontroller/examples/ftnormal+kill

It is supposed that design has already been done using **Ftbuilder** and that generated model files have been copied respectively **include** and **src** subdirectories. We remind you that using degraded mode management facility implies the adoption of a specific design process and is restricted to a particular task model described in the user's guide. So it is greatly recommended to read the user'sguide FT section first.

Otherwise, the coding style is quite similar to standard RTLinux programming. All ft_tasks are periodic tasks. You have to provide two routines for each one. A routine for normal_behavior and a routine for degraded_behavior. The description of the global application model is generated by the ft builder and results into two files that must be included in your application.

Include application header

```
*/
/******************************************************************************/
/*                                                                            */
/* Include                                                                    */
/*                                                                            */
/******************************************************************************/

// FT application include
#include "ft_normal_kill.h"
```

Init module

```
/******************************************************************************/
/*                                                                            */
/*                  FT Application : init module                              */
/*                                                                            */
/******************************************************************************/
int init_module(void) {

  // Indice
  int ap_i=0;
  // String
  char ap_i_str[4];
  // Return code
  int ap_cr=0;

  // Scheduling parameters of normal and degraded threads
  FT_sched_param ap_normal_sched_param, ap_degraded_sched_param;
  // Behavior of appli task
  FT_task_behavior ap_task_behavior;

  rtl_printf("\n****************\n");
  rtl_printf(" FT_Normal+Kill \n");
  rtl_printf("****************\n");
```

Include model issued by Ftbuilder

```
/*--------------------------------------------------------------------------*/
/*                                                                          */
/* FT Application : Appli Modele Source Generated Code                      */
/*                                                                          */
/*--------------------------------------------------------------------------*/

  // FT application modele source code (generated code by FT-Builder)
```

```
#include "ft_appli_model.c"
```

Init and creation of ft_tasks

```
/*------------------------------------------------------------------------*/
/*                                                                        */
/* FT Application : Init and create FT-tasks                              */
/*                                                                        */
/*------------------------------------------------------------------------*/
```

Initialization of scheduling parameters

```
// Set scheduling parameters of the normal thread
ap_normal_sched_param.prio=APPLI_PRIORITY;
ap_normal_sched_param.period=APPLI_PERIODE;
ap_normal_sched_param.start_time=APPLI_START_TIME;
ap_normal_sched_param.deadline=APPLI_DEADLINE;
ap_normal_sched_param.duration=APPLI_DURATION;

// Set scheduling parameters of the degraded thread
ap_degraded_sched_param.prio=APPLI_PRIORITY;
ap_degraded_sched_param.period=APPLI_PERIODE;
ap_degraded_sched_param.start_time=APPLI_START_TIME;
ap_degraded_sched_param.deadline=APPLI_DEADLINE;
ap_degraded_sched_param.duration=APPLI_DURATION;
```

Loop of init for all ft_tasks

```
 // Loop of init of FT-tasks
 for (ap_i=1; ap_i < APPLI_TASKS_MAX_NB+1; ap_i++) {
   /*rtl_printf("\nApplication : ap_i=%d", ap_i);*/

   strcpy(&ap_task_name_tab[ap_i][0],"FT_TASK_");
   sprintf(ap_i_str, "%d", ap_i);
   strcat(&ap_task_name_tab[ap_i][0], ap_i_str);

   // Init a FT-task
   // A task has 2 threads with normal and degraded behavior
   /*rtl_printf("\nApplication : Function init_module : ft_task_init\n");*/
   if ((ap_cr=ft_task_init(&ap_task_name_tab[ap_i][0],

                 (void *)ap_normal_behavior_routine,

                 (void *)ap_degraded_behavior_routine,
                 ap_i,
                 ap_i,
                 ap_normal_sched_param,
                 ap_degraded_sched_param
#ifdef FT_PREALLOCATED_THREAD_STACK
                      /* fxr: stack address and size */
                 ,
                 NULL,
                 0,
                 NULL,
                 0
#endif
                 )) < 0) {
```

```
    rtl_printf("\nApplication : ERROR");
    rtl_printf("\nApplication : Function init_module");
    rtl_printf("\nApplication : Not valid ft_task_init return value");
    rtl_printf("\nApplication : ap_i=%d ap_cr=%d\n", ap_i, ap_cr);
    // Be careful : Necessary output of init_module with error !!!
    // because it is not possible to start the application
    // if one of the tasks initialisation has failed
    return -1;
  }
  else {
    ap_task_id_tab[ap_i]=ap_cr;
  }
}
```

If init is OK, then the tasks are added to ap_task_id_tab and the actual creation of ft_tasks can be started.

Loop of creation of ft_tasks.

Two tasks are created with default behavior FT_TASK_NORMAL.

Each ft_task creation results in the creation of two threads corresponding to the normal and degraded beahavior of the ft_task. The ft_task is created with a default behavior which is generally FT_TASK_NORMAL. The thread corresponding to normal behavior is then made periodic and started while the other one is suspended.

```
// Loop of creation of FT-tasks (inverse order : why not)
for (ap_i=APPLI_TASKS_MAX_NB; ap_i > 0; ap_i--) {
  /*rtl_printf("\nApplication : ap_task_id_tab[%d]=%d\n", ap_i, ap_task_id_tab[ap_i]);*/

  // Create a FT-task
  // Create and start the normal thread (awake each FT_APPLI_PERIODE)
  // Create, start and make wait the degraded thread
  if (ap_i == 1) {
    ap_task_behavior=FT_TASK_NORMAL;
    rtl_printf("\nApplication : ap_i=%d ap_task_behavior=%s\n",
                         ap_i, ft_task_behavior_str[ap_task_behavior]);
  }
  else if (ap_i == 2) {
    ap_task_behavior=FT_TASK_NORMAL;
    rtl_printf("\nApplication : ap_i=%d ap_task_behavior=%s\n",
                         ap_i, ft_task_behavior_str[ap_task_behavior]);
  }
  else {
    rtl_printf("\nApplication : ERROR");
    rtl_printf("\nApplication : Function init_module");
    rtl_printf("\nApplication : Not valid appli tasks number");
    rtl_printf("\nApplication : ap_i=%d\n", ap_i);
    return -1;
  }
  /*rtl_printf("\nApplication : ", ft_task_behavior_str[ap_task_behavior]);*/
  if ((ap_cr=ft_task_create(ap_task_id_tab[ap_i],
                              ap_task_behavior)) < 0) {
    rtl_printf("\nApplication : ERROR");
    rtl_printf("\nApplication : Function init_module");
```

```
      rtl_printf("\nApplication : Not valid ft_task_create return value");
      rtl_printf("\nApplication : ap_task_id_tab[%d]=%d ap_cr=%d\n",
                        ap_i, ap_task_id_tab[ap_i], ap_cr);
      // Be careful : Necessary output of init_module with error !!!
      // because it is not possible to start the application
      // if one of the tasks creation has failed
      return -1;
    }
  }
  /*rtl_printf("\nApplication : End of application !!!");*/
  return 0;
}
```

Cleanup module

```
/*****************************************************************************/
/*                                                                           */
/*                    FT Application : cleanup module                        */
/*                                                                           */
/*****************************************************************************/
void cleanup_module(void) {
  // Indice
  int ap_i=0;
  // Return code
  int ap_cr=0;
  rtl_printf("\nApplication : CLEANUP application threads !!!\n");
  // Delete all the application FT-tasks
  for (ap_i=1; ap_i < APPLI_TASKS_MAX_NB+1; ap_i++) {
    if ((ap_cr=ft_task_end(ap_task_id_tab[ap_i])) < 0) {
      rtl_printf("\nApplication : ERROR");
      rtl_printf("\nApplication : Function cleanup_module");
      rtl_printf("\nApplication : Not valid ft_task_end return value");
      rtl_printf("\nApplication : ap_task_id_tab[%d]=%d ap_cr=%d", ap_i, ap_task_id_tab[ap_i],
ap_cr);
      // Be careful : NOT necessary output of cleanup_module with error !!!
      // because it is recommended to stop all the tasks of the application
      // even if one of the tasks end has failed
    }
  }

 // Stop the FT controller
  /*ft_controller_stop();*/
  // Stop the FT appli monitor
  /*ft_appli_monitor_stop();*/
}
MODULE_LICENSE("GPL");
```

Normal behavior routine definition

In this example, the routine runs 20 cycles then kills itself so that it simulates an error . The event will then be detected by the controller and the replacement behavior will be activated.

```
/*****************************************************************************/
/*                                                                           */
/* FT Application task : normal behavior routine                             */
/*                                                                           */
/*****************************************************************************/
```

```
void *ap_normal_behavior_routine(void *arg) {

  int ap_no_cycle=0;
  int ap_j=0;
  int ap_nloops=0;
  int ap_task_id=0;
  /*int ap_cr=0;*/

  ap_task_id = (int) arg;

  rtl_printf("\nApplication : ft-task %d, thread %x started, normal behavior", ap_task_id,
pthread_self());
  rtl_printf("\nApplication : ft-task %d, thread %x switching to wait, normal behavior\n",
ap_task_id, pthread_self());

  // Infinite loop for appli
  while(1) {
    // Wait make periodic or next period of the normal behavior thread !!!
    pthread_wait_np();
    if (ap_no_cycle == 0)
      rtl_printf("\nApplication : ft-task %d, thread %x switching to running, normal behavior\n",
ap_task_id, pthread_self());
    ap_no_cycle++;
    /*rtl_printf("\n\nApplication : ap_task_id=%d ap_task_behavior=NORMAL ap_thread_kid=%x
ap_no_cycle=%d\n", ap_task_id, pthread_self(), ap_no_cycle);*/

    // Start simulation of cancelling the normal behavior thread
    if ((ap_no_cycle == 20) && (ap_task_id == 2)) {
      rtl_printf("\nApplication : ft-task %d, thread %x cancelling, normal behavior, no_cycle %d\n",
ap_task_id, pthread_self(), ap_no_cycle);
      pthread_cancel(pthread_self());
      break;
    }
    // End simulation of cancelling the normal behavior thread


    // For test
    if (ap_no_cycle == 10) {
      rtl_printf("\nApplication : ft-task %d, thread %x, no_cycle %d,
                normal behavior\n",
                ap_task_id, pthread_self(), ap_no_cycle);
    }
    // Timing loop
    ap_nloops=10000;
    for (ap_j=0; ap_j<ap_nloops; ap_j++);

  } // end of 'while'

  return (void *)0;
}
```

Degraded behavior routine

The degraded behavior routine has the same structure as the normal one.

```
/***************************************************************************/
/*                                                                         */
/* FT Application task : degraded behavior routine                         */
/*                                                                         */
/***************************************************************************/

void *ap_degraded_behavior_routine(void *arg) {

  int ap_no_cycle=0;
  int ap_j=0;
  int ap_nloops=0;
  int ap_task_id=0;
  /*int ap_cr=0;*/

  ap_task_id = (int) arg;

  /***
  rtl_printf("\nApplication : ft-task %d, thread %x started, degraded behavior",
             ap_task_id, pthread_self());
  rtl_printf("\nApplication : ft-task %d, thread %x switching to wait, degraded behavior\n",
             ap_task_id, pthread_self());
  ***/

  // Infinite loop for appli
  while(1) {

    // Wait make periodic or next period of the degraded behavior thread !!!
    pthread_wait_np();

    if (ap_no_cycle == 0)
      rtl_printf("\nApplication : ft-task %d, thread %x switching to running, degraded behavior\n",
ap_task_id, pthread_self());

    ap_no_cycle++;

    // For test
    if (ap_no_cycle == 10) {
      rtl_printf("\nApplication : ft-task %d, thread %x, no_cycle %d, degraded behavior\n",
          ap_task_id, pthread_self(), ap_no_cycle);
    }

    // Timing loop
    ap_nloops=10000;
    for (ap_j=0; ap_j<ap_nloops; ap_j++);

  } // end of 'while'

  return (void *)0;
}
```

Application compiling

In order to compile an ft application, it is necessary to have OCERA architecture installed and compiled (see general OCERA installation) with the following components selected :

- posixtrace
- ft_components : ftappmon and ft_controller

The application code given as an example is located in the following directory : **$OCERA_DIR/components/ft/ftcontroller/examples/ftnormal+kill**

The compilation of the example follows a classic module compilation procedure.

- Change to example directory

```
$ cd $OCERA_DIR/components/ft/ftcontroller/examples/ftnormal+kill
```

- Clean the $OCERA_DIR/components/ft/ftcontroller/examples/ftnormal+kill directory:

```
$ make clean
```

Old ftnormal+kill.o file is cleaned up if it exists.

- Compile the ftnormal+kill module:

```
$ make all
```

The ftnormal+kill.o module is now available under the following directory :

$OCERA_DIR/components/ft/ftcontroller/examples/ftnormal+kill/src

Running an application

The procedure to launch the example is the following :

- Go to the ft/ftcontroller/examples/<example_name> directory level :

```
$ cd $OCERA_DIR/comp/ft/ftcontroller/examples/ftnormal+kill
```

- Be a root user

```
$ su
Password:
#
```

At this stage, it is necessary to be a root user. Further, the user has to be a normal user.

- Install and execute all the module:

```
# make example
```

- Get the modules execution traces:

```
# tail -f /var/log/messages
```

Be careful to see only the last execution traces (not the previous ones).

Additional information and recommendations

prerequisites

FT facilities for degraded mode management of real_time embedded applications are available for Hard RTLinux environments only.

All application tasks are RTLinux tasks created within one single application module that can be dynamically loaded into the system. A user application must thus consist in one single RTLinux module. As usual this module must contain declarations, one init_module function and one cleanup_module function.

The prerequisites are thus a running OCERA RTLinux kernel with PosixTrace and FT_components installed (see FT configuration section in chapter three). More precisely, the prerequisites are :

Configuration requirements

-> OS Type

+ Hard and Soft realtime (RTLinux + Linux)

-> Fault Tolerance components

+ FT dependencies + Bigphysarea support

+ Hard Realtime + Degraded Management

+ FT Controller

+ FT Application Monitoring

+ Utilities + Fault Tolerant Building Tool

-> Scheduling

+ Application defined scheduler

+ or EDF

+ or EDF + Deadline miss detection (very experimental)

Scheduling of tasks versus event detection is chosen at the configuration level :

- either priority (PRIO) by Application defined scheduler or EDF for **only** Pthread_kill events detection,

- or EDF and Deadline_miss detection for Pthread_kill **and** Deadline_miss events detection.

It is important to consider that the scheduling choice versus event detection has to be consistent with application modes transitions in the application model specification in FT_builder. Remember that the scheduling configuration choice automatically configures the FT components at compilation level for Pthread_kill and/or deadline-miss events detection on threads by ftcontroller.

-> Posix API

+ Posix Trace support

In the example presented in the previous section, only pthread_kills errors were targetted. If you want to detect also deadline-miss errors you should first make sure that the EDF+DLM configuration options have been selected in the configuration steps.

Specify FT parameters of application with FTbuilder

Do not forget that application design must be done using Ftbuilder. This acquisition tool is necessary to specify application modes, ft_tasks parameters and application modes transition conditions. It then generates application model files that have to be included in your application.

FT_tasks real-time parameters

The ft_tasks real-time parameters (period, start_time, estimated_duration, deadline, priority) are entered via the FT_builder (see FT task specification in user'sguide :chapter eight). Static scheduling plan on ft-tasks has to be faisable.

Restrictions and recommandations for these real-time parameters are :

- 1 ms <= period < =100 s
- 0 <= start_time < period
- 0 < estimated_duration < period
- 1 ms <= deadline <= period
- 0 <= priority <=10

Note that the FT components ftappmon and ftcontroller have a priority value superior to the ft_tasks priority values.

Include files

The application header must include the following ft specific files:

- header of the ft_components API : ft_api_appmon_appli.h
- header file for the application model (generated by the FT_builder) : ft_appli_model.h

The application source file must include the following ftbuilder generated specific file:

- ft_appli_model.c

This file contains the FT application model source code that instanciates internal data structures of FT components and is used to monitor application and implement dynamic reconfiguration.

Redundancy Management

Description

The Redundancy Management facilities offered by OCERA consist of two complementary components: **ftredundancymgr** and **ftreplicamgr**. Used together, they provide a framework for implementing redundancy management support for user's application. They respectively control redundancy at the application level and at the task level on each node.

This first implementation is intended to provide a basic framework whose goal is to offer a global set of facilities that permit transparent implementation of redundancy for developers of real-time applications. It offers a passive replication model, the task model is a simplified one (periodic tasks), fault-detection is based on heartbeats and timeouts, consistency of replicas is ensured by periodic checkpointing.

The current implementation is located at Linux user-space level using ORTE component for communication between nodes. However implementation choices have been made in such a way as to facilitate the port to OCERA Hard Real-Time level when ORTE become available at this level. Indeed these facilities can be enriched in the future.

In order to support data consistency and to facilitate tasks recovering on node crash, a task model must verify synchronization properties. In the current implementation, we have introduced a specific task model described in the FT redundancy management section of the User's guide.

An application consists of a set of ftr_tasks (fault_tolerant redundant tasks). They are basically encapsulation of real-time periodic tasks. Creating a ftr_task results in the creation oone master (active) thread and several passive replicas (suspended threads). A context object is defined for each ftr_task, the application developer must define the set of variables which must be part of the context at design time. This context object is automatically updated and broadcasted at each end of cycle to all passive replicas.

Implementation principles are driven by the will to make redundancy management as transparent as possible to the application developer. So in order to develop an application, the user can almost forget about underlying ft redundancy management architecture.

To support the approach, two features are introduced and used within the user's process :

- creation of a control thread dedicated to redundancy control (ftr_control_thread)
- encapsulation of application tasks into ftr_tasks_threads

The ftr_control_thread is in charge of initialization and control of application. Created within the user application process it communicates with **ftredundancymgr** and **ftreplicamgr.**

The ftr_tasks_threads are generic encapsulation of redundant tasks. A ftr_task_thread is created for each user's application redundant task. It ensures periodic execution of user's task routine, management of context entity and of shared data entities and communication with **ftreplicamgr** for checkpointing.

Communication with **ftreplicamgr** and **ftredundancymgr** are achieved using ORTE publisher/subscriber mechanisms both within a node and between nodes, but this is transparent to the user since calls are made either from ftr_control_thread or from ftr_tasks_threads generic part using specific internal APIs that are described in the corresponding component sections below.

Usage

he FT facilities available at soft real-time level are Redundancy Management.

Redundancy Management configuration process takes three steps:

1. **OS Type Selection** : Soft real-time must be chosen.
2. **Components Selection** :
    - FT components/Soft Realtime/Redundancy Management.
    - Communication components/ORTE

1. OS Type Selection

Redundancy Management is provided only at soft real-time level . So the Soft real-time must be selected in the OS type section.

2.1 FT components selection

Select the Soft Realtime subsection in Fault Tolerance components, then the two components Task Replica Manager and Task Redundancy Manager are automatically selected.

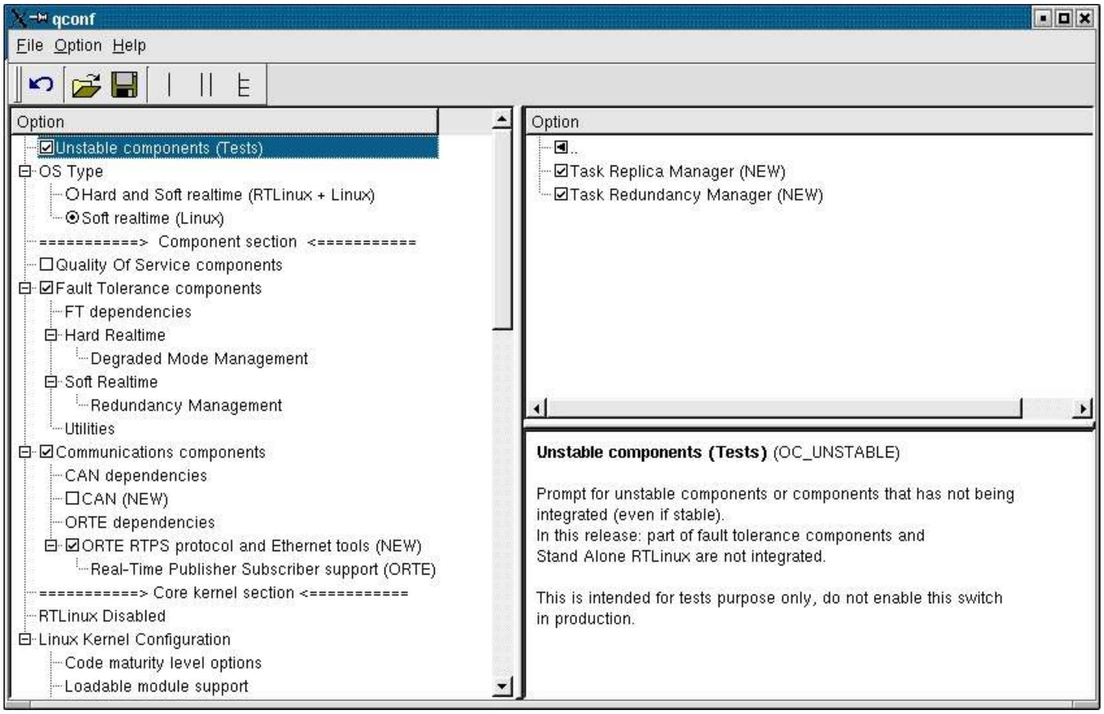


Figure 1 : FT Redundancy Management Configuration step1

2.2ORTE communication component selection

The Redundancy Management facility relies on ORTE (Realtime Ethernet) components that implement RTPS (RealTime Publisher Subscriber Protocol) communication protocol.

So you must select the following features in the communication components section.

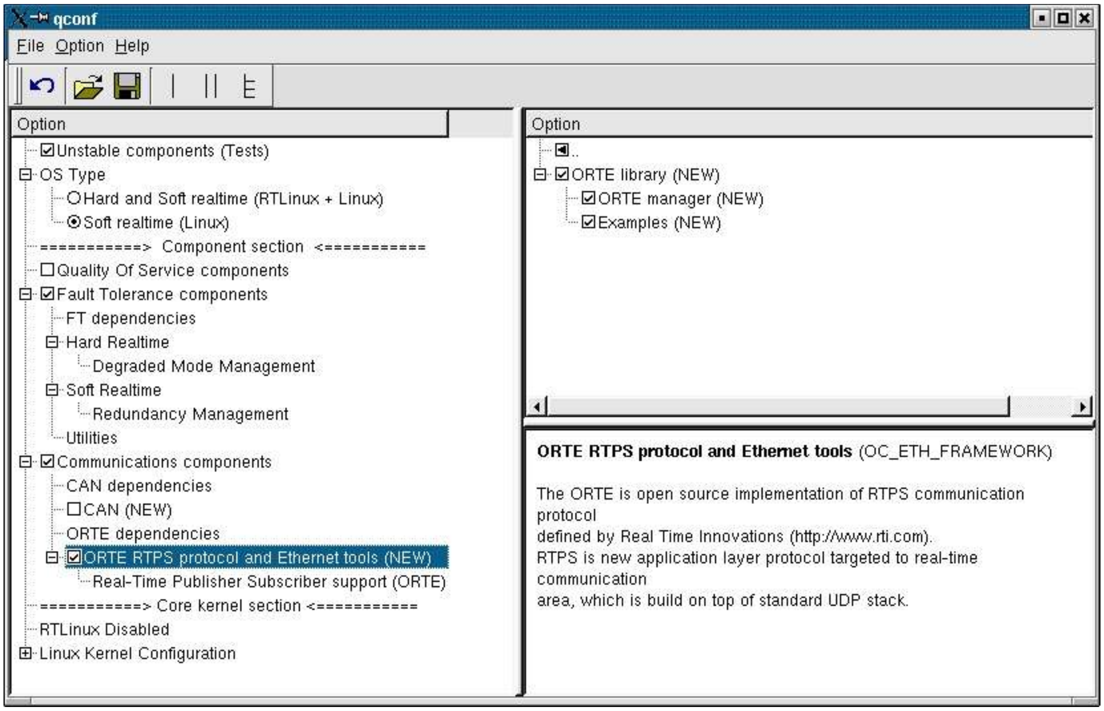


Figure 2 : FT Redundancy Management Configuration step2

Programming Interface (API)

The approach chosen results in a very limited user's API necessary mainly for initialization and termination of user application. Most of user' s application code consists in routines that will be run within ftr_tasks_threads. The important issue is to specify the context data and shared resources for each task at design. Concurrency control over such shared data is then automatically insured by the execution model. Then threads routine can be written simply in a usual way.

So the external user API is actually restricted to the followin few functions :

```
int ftr_application_register(char *, FTR_APPLI_DESC * ,
                        ManagedApp *);
int ftr_appli_desc_init(FTR_APPLI_DESC *);
int ftr_appli_task_create(FTR_APPLI_TASK_DESC *);
int ftr_appli_task_end(int );
int ftr_application_terminate(char*);
```

FT redundancy management User API

They are called within user's main application thread and handled by the ftr_control_thread (named hereafter **ftr_controller**) running within the application process. Then the **ftr_controller** uses internal API to communicate with **ftredundancymgr** and with **ftreplicamgr**.

The **ftredundancymgr** has a small external API that is used to start or end the redundancy management facility.

In addition, each component has also internal API(s) that permit interactions between them.

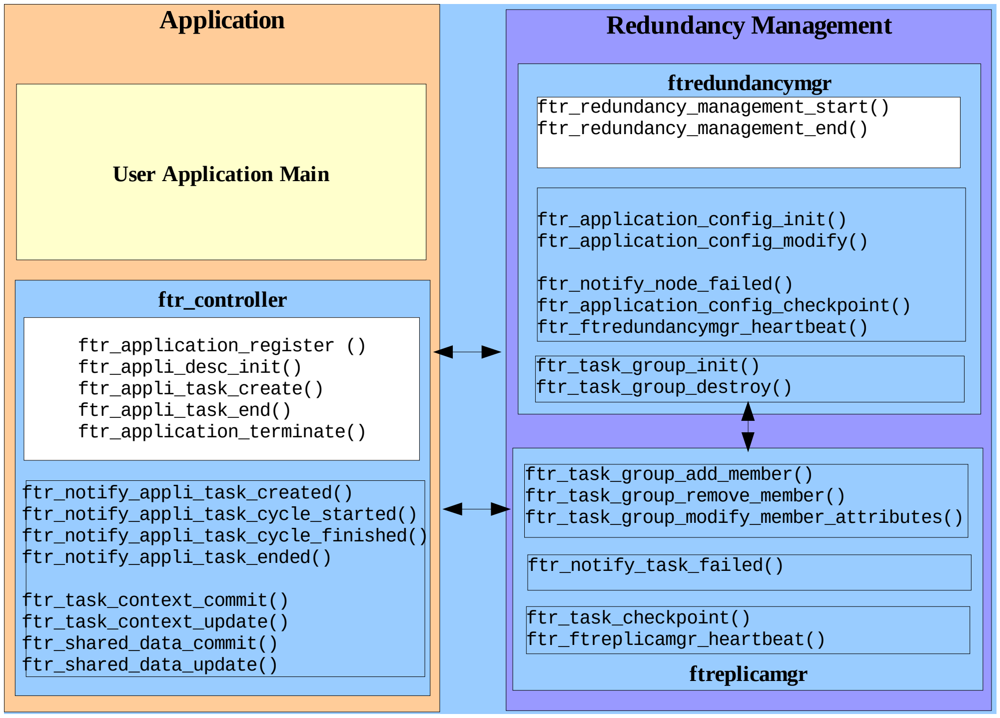


Figure 3 : FT Redundancy Management User's API

Principles of application execution

In the following figure we illustrate on a very simple example how an application is started.

Once the design is done, the resulting architecture on a node is composed of the user's process and of the Redundancy Management Facility process (in the following view we do not show ORTE process).

Within the user's process the yellow (or white) parts concern code written by users and blue (or gray) part concern generic ftr code.

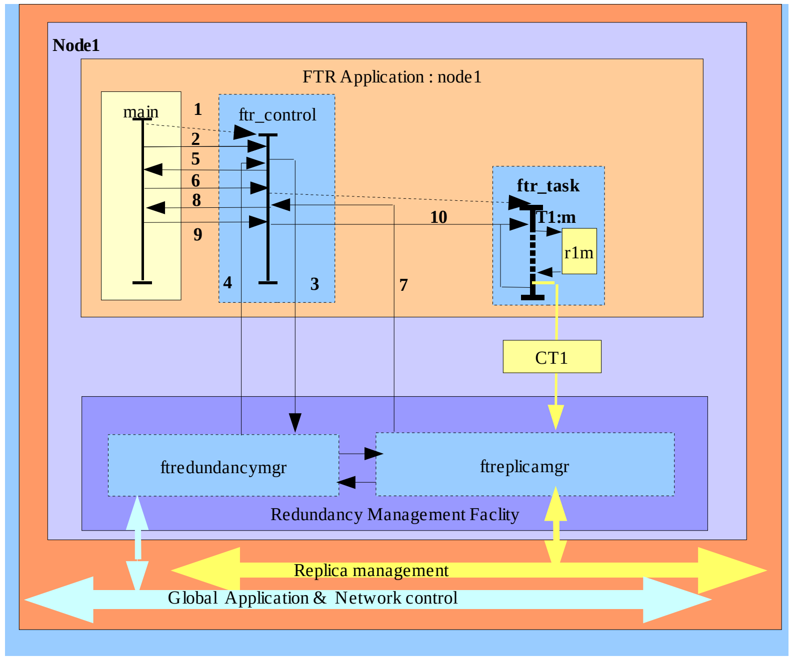


Figure 4 : FT Redundancy Management - application execution principles

First the application creates the ftr_control_thread (1), then it calls the **`ftr_application_register`** primitive to register the application (2), the ftr_control_thread then communicates with the **ftredundancymgr** to setup data (3) for the new application, and waits for acknowledgment (4) from it before returning OK (5) to the user main thread.

Then the **`ftr_appli_desc_init`** primitive is called to setup application data structures and ftr_tasks_threads (6). At this stage ftr_tasks_threads are created but the corresponding users routines are not started. When all the infrastructure is ready, the **ftreplicamgr** notifies the ftr_control_thread (7) which returns OK (8) to user's main thread.

Finally the user can call the **`ftr_appli_task_create`** primitive to start a ftr_task.(9). The ftr_controller_thread then makes the ftr_task_thread start periodic call to the corresponding user's ftr_task_routine (10).

Two other primitives are available to end an ftr_appli_task (**`ftr_appli_task_end`**) and to terminate the overall application(**`ftr_application_terminate`**).

The user has to define specific data structures, one to describe the overall application structure and one to describe each ftr_task.

It is intended that the **Ftbuilder** tool (already available for the specification of degraded mode management) will assist the designer to determine these features and automatically generate the corresponding data structures. For the moment this facility is not implemented yet, and data is provided in a file read by the **ftr_appli_desc_init** primitive.

Example

It is intended that the **Ftbuilder** tool (already available for the specification of degraded mode management) will assist the designer to determine these features and automatically generate the corresponding data structures. For the moment this facility is not implemented yet, and data is provided in a file read by the **ftr_appli_desc_init** primitive.

Coding steps

An application can be written rather simply following the different generic steps :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <orte.h>
#include <netdb.h>
#include <pthread.h>
#include <simple_appli.h>
#include <ftredundancymgr.h>
#include <appli_controller.h>
ManagedApp *appli;
pthread_t ftr_control_thread;
int main(void)
{
  int res = 0;
  void *ret;
  FTR_APPLI_DESC application_desc;
  FTR_APPLI_TASK_DESC application_task_desc_1;
  FTR_APPLI_TASK_DESC application_task_desc_2;
```

1. Declarations for ftr_application

```
/* Creation of ftr_control_thread */
 pthread_create(&ftr_control_thread, NULL,(*ftr_main_control_routine),
                NULL);
 if (res != 0) {
    perror("Redundancy Management thread creation failure ...
            exiting");
    exit(-1);
   };
```

2. Creation of ftr_control_thread of ftr_application

The ftr_control_thread of the application is created in the beginning of the main thread to install the ftr architecture within the application process. In the future, it will be replaced by a macro. The ftr_main_control_routine, is a generic control loop that monitors events from and to the ftr_process. It also accepts requests form the user main thread.

```
/* Init appli_desc structure */
 res = ftr_appli_desc_init(&application_desc);
```

```
 if (res == -1)    {
          perror("Redundancy Management : application desc init failed ...
           exiting");
    exit(-1);
   };
```

3. initialization of application data structures

During this step, data structures describing application and tasks are initialized.

```
 /* Register application */
 res = ftr_application_register(APPLI_NAME, &application_desc,appli);
 if (res == -1) {
    perror("Redundancy Management : application registration failed...
            exiting");
    exit(-1);
   };
```

4. Registration of application

Application registration is done towards ftr process which in turn propagate information over network (thanks to ORTE) to other ftr processes. (Application is also registered as ORTE Application). (Internal tables are initialized, groups of replicas are created and instances created on each node).

```
  /* Tasks creation */
 application_task_desc_1 = application_desc.appli_tasks_tab[1];
 application_task_desc_1.appli_task_routine = ft1;
 res = ftr_appli_task_create(&application_task_desc_1);
if (res == -1) {
    perror("Redundancy Management : task creation (1) failed...
            exiting");
    exit(-1);
   };
...
```

5. ftr_tasks creation for ftr_application

During this step each application task is created using the ftr_task_desc of each one. This steps defines mainly the routine to be run within the generic ftr_task_thread and the related real-time parameters (period, estimated_duration, deadline). At the end of each period, the current context is sent to all its replicas on other nodes.

Once this is done for each task, the application runs in a nominal way.

To end a task the following call is necessary.

```
 /* Requiring End of Task 1 */
 res = ftr_appli_task_end(1);
...
```

6. ftr_tasks ending for ftr_application

This ends the corresponding ftr_task (and all its replicas). All ftr_tasks have to be ended before application itself can be ended.

```
/* Requiring Application Termination */
 ftr_application_terminate(APPLI_NAME);
/* Waiting for end of control_thread */
```

```
 pthread_join(ftr_control_thread,&ret);
 if (ret != PTHREAD_CANCELED) {
    i = (int) ret;
    printf("Main : end of ftr_control_thread ret = %d\n", i);
   };

printf("\nAppli ending : ");
return 0;
}
```

7. Termination of ftr_application

Once all the ftr_tasks are ended, resources are freed and the ftr_control_thread is ended, then application terminates.

Obviously, the user must in addition provide the code of the routines that will be run within each ftr_tasks_thread. A pointer to this routine is a member of the ftr_task_desc structure.

In our simple example :

```
int ft1(int i)
{
  printf("Function ft1 running with arg %d\n",i);
  sleep(3);
  return 0;
}
```

The status of the current implementation is still in a testing phase. The example implemented tests application setup, execution and termination.

How to run the examples

Up to now, the examples developed are common to the two components.The example directory is located within the **ftredundancymgr** component :

ocera/components/ft/ftredundancymgr/examples/ftr_appli

It is the Makefile located within this directory that builds the test application. In order to run the example it is necessary to compile and start the **ftredundancy management** facility first.

Implementation :

The ***ft/ftredundancymgr/examples/*** directory has the following structure:

```
    examples
      ! --- README
      ! --- INSTALL
      ! --- Makefile
      ! --- ftr_appli
      !       !--- README
      !       !--- INSTALL
      !       !--- Makefile
      !       !--- include
      !       !       !---ftr_appli.h
      !       !--- src
      !       !       !---ftr_appli.c
```

The ***ftr_appli*** is a simple application that has been developed to test the **ftredundancy management** facility.

The general OCERA Makefile file permits the compilation of the overall OCERA tree provided options are selected in the configuration step (see OCERA HOWTO for OCERA configuration steps). However examples can be compiled separately afterwards.

Compilation :

In order to compile the example please follow next steps :

```
- Go to the ft/ftredundancymngr/examples directory:
$  cd ft/ftredundancymngr/examples

- Clean the ft/ftredundancymngr/examples directory:
$  make clean

- Compile the examples:
$  make
```

Installation/Execution :

Note that execution of examples requires a distributed architecture. So the ftcomponents and examples must be present on each machine that will be involved in the test. This requires additional operations and controls before the example can be run.

- Install OCERA (or at least ORTE and ftcomponents) on each machine.
- Insure that rights are set so as to allow for remote execution of the code corresponding to both components and application.
- Set up environment variables

(See section 2.7 for details)

The example runs on two nodes N1 and N2. The application has two tasks T1 and T2.

T1 master task is running on node N1 and T2 master task is running on Node2. Node1 is the master node on application start.

To run the application one must :

- start ftredundancy management

  A shell script allows for this, it is located in ***ft/ftredundancymngr/src*** :

```
$ ftrm_start  <Node1> <Node2>
where <Nodei> is an hostname
```

  It starts ORTEManager on each node, then starts ftredundancy components on each node. Actually the two components of a node are linked a single Linux executable named ft_redman.

The master node is the current node (it must be the same as the first argument , here Node1).

- start application on master node

  $ cd ftr_appli/src

  $ ./ftr_appli

  The application starts first on Node1 then on Node2. Replicas are created and ftr_tasks started.

  After a given number of cycles the application ends.

Comments

The current implementation is still a prototype one and the development status is very experimental.We have adopted an incremental development cycle and some functionalities have still very basic implementation. The main goal of this step was to provide a consistent overall framework for redundancy management. A lot of work has still to be done to make an efficient operational environment of it.

However, the example has permitted to test the ft redundancy management overall structure .

- Ft redundancy framework set-up and functioning
- Application registration
- Application execution
- Application termination.
- Node crash detection
- Application dynamic reconfiguration.

CAN

By
Frantisek Vacek - CTU
Pavel Pisa - CTU

Installation

CAN commponet uses the OMK make system. There is **no ./configure** script. The component can be built as a part of OCERA tree or as a stanalone. If it is build as a standalone you should run script **can/switch2standalone**.

```
[fanda@lab3-2 can]$ ./switch2standalone
Default config for /utils/suiut
Default config for /utils/ulut
Default config for /utils/flib
Default config for /utils
Default config for /canvca/libvca
Default config for /canvca/cantest
Default config for /canvca
Default config for /candev/cpickle
Default config for /candev/nascanhw
Default config for /candev
Default config for /canmon/canmond
Default config for /canmon/canmonitor
Default config for /canmon
Default config for /lincan/src
Default config for /lincan/utils
Default config for /lincan
Default config for

To modify required configuration options, create "config.omk" file
and add modified lines from "config.omk-default" file into it
```

```
To build project, call simple "make"

GNU make program version 3.81beta1 or newer is required to build project
check by "make --version" command
```

Default configuration of any subcommponent can be changed by introducing a file `config.omk` in the subcommponent directory. Defines in this file simply beats defines in file `config.omk-default`, so you can put there only defines that are different that the default ones in the `config.omk-default`.

For example by default the building of Java application is disabled. That means that there is a line *CONFIG_OC_CANMONITOR=n* in the `config.omk-default`. If you have the Java SDK and the ant build system installed, add the line *CONFIG_OC_CANMONITOR=y* to the file `config.omk` to enable the Java applications to be build.

When you switch to standalone, you can build any particular commponent by running make in the commponent directory.

For more details see file `can/README.makerules`.

You can download make version 3.81beta1 source from http://cmp.felk.cvut.cz/~pisa/can/make-3.81beta1.tar.gz or the binary from http://cmp.felk.cvut.cz/~pisa/can/make-3.81beta1-i586-0.gz.

Programs in this package does not need special installation. They can run from any directory. Just type **`make`** in `can/canmon` directory and copy desired files wherever you want. The make process is an out source build. After make you can find your binaries in directory `can/_compiled/bin`. If you want to compile only one component, type **`make`** in the component's directory. That commponent and all commponents in subdirectories will be build.

Restrictions on versions of GNU C or glibc are not known in this stage of project but gcc ver >= 3.0 is recommended. Java SDK ver. 1.4 or above is also recommended (assert keyword support).

API / Compatibility

VCA base API

Name

struct canmsg_t — structure representing CAN message

Synopsis

```
struct canmsg_t {
  int flags;
  int cob;
  unsigned long   id;
  canmsg_tstamp_t timestamp;
  unsigned short  length;
  unsigned char   * data;
};
```

Members

flags
: message flags `MSG_RTR` .. message is Remote Transmission Request, `MSG_EXT` .. message with extended ID, `MSG_OVR` .. indication of queue overflow condition, `MSG_LOCAL` .. message originates from this node.

cob
: communication object number (not used)

id
: ID of CAN message

timestamp
: not used

length
: length of used data

data

data bytes buffer

Header

canmsg.h

Name

struct canfilt_t — structure for acceptance filter setup

Synopsis

```
struct canfilt_t {
  int flags;
  int queid;
  int cob;
  unsigned long id;
  unsigned long mask;
};
```

Members

flags
: message flags `MSG_RTR` .. message is Remote Transmission Request, `MSG_EXT` .. message with extended ID, `MSG_OVR` .. indication of queue overflow condition, `MSG_LOCAL` .. message originates from this node. there are corresponding mask bits `MSG_RTR_MASK`, `MSG_EXT_MASK`, `MSG_LOCAL_MASK`. `MSG_PROCESSLOCAL` enables local messages processing in the combination with global setting

queid
: CAN queue identification in the case of the multiple queues per one user (open instance)

cob
: communication object number (not used)

id
: selected required value of cared ID id bits

mask
: select bits significand for the comparation; 1 .. take care about corresponding ID bit, 0 .. don't care

Header

canmsg.h

Name

vca_h5log — converts VCA handle to printable number

Synopsis

```
long vca_h5log (vcah);
vca_handle_t vcah;
```

Arguments

vcah
: VCA handle

Header

can_vca.h

Return Value

unique printable VCA handle number

Name

vca_open_handle — opens new VCA handle from CAN driver

Synopsis

```
int vca_open_handle (vcah_p,
                     dev_name,
                     options,
                     flags);
vca_handle_t *  vcah_p;
const char *    dev_name;
const char *    options;
int             flags;
```

Arguments

vcah_p
: points to location filled by new VCA handle

dev_name
: name of requested CAN device, if NULL, default VCA_DEV_NAME is used

options
: options argument, can be NULL

flags
: flags modifying style of open (VCA_O_NOBLOCK)

Header

can_vca.h

Return Value

VCA_OK in case of success

Name

vca_close_handle — closes previously acquired VCA handle

Synopsis

```
int vca_close_handle (vcah);
vca_handle_t vcah;
```

Arguments

vcah
: VCA handle

Header

can_vca.h

Return Value

Same as libc `close` returns.

Name

vca_send_msg_seq — sends sequentially block of CAN messages

Synopsis

```
int vca_send_msg_seq (vcah,
                      messages,
                      count);
vca_handle_t vcah;
canmsg_t *   messages;
int          count;
```

Arguments

vcah
: VCA handle

messages
: points to continuous array of CAN messages to send

count
: count of messages in array

Header

can_vca.h

Return Value

Number of sucessfully sent messages or error < 0

Name

vca_rec_msg_seq — receive sequential block of CAN messages

Synopsis

```
int vca_rec_msg_seq (vcah,
                     messages,
                     count);
```

vca_handle_t *vcah*;

canmsg_t * *messages*;
int *count*;

Arguments

vcah
: VCA handle

messages
: points to array for received CAN messages

count
: number of message slots in array

Header

can_vca.h

Return Value

number of received messages or error < 0

Name

vca_wait — blocking wait for the new message(s)

Synopsis

```
int vca_wait (vcah,
              wait_msec,
              what);
```

vca_handle_t *vcah*;
int *wait_msec*;
int *what*;

Arguments

vcah
VCA handle
wait_msec
number of miliseconds to wait, 0 => forever
what
0,1 => wait for Rx message, 2 => wait for Tx - free 3 => wait for both

Header

can_vca.h

Return Value

Positive value if wait condition is satisfied

Name

vca_gethex — gets one hexadecimal number from string

Synopsis

`int` **`vca_gethex`** `(`*str*, *u*`);`
const char * `str`;
int * `u`;

Arguments

str
scanned string
u
pointer to store got value

Return

the number of eaten chars

Header

can_vca.h

Name

vca_strmatch — get token from string

Synopsis

```
int vca_strmatch (str,
                  template);
```

const char * *str*;
const char * *template*;

Arguments

str
: scanned string

template
: token template template consists of characters and '~' matching one or more of spaces ie. '~hello' matches ' hello', ' hello', ' hello' etc.

Return

the number of used chars from str if match or negative value (number of partially matched chars from str - 1) if template does not match

Header

can_vca.h

Name

vca_msg2str — converts canmsg_t to the string

Synopsis

```
int vca_msg2str (can_msg,
                 buff,
                 buff_len);
```

const struct canmsg_t * *can_msg*;
char * *buff*;
int *buff_len*;

Arguments

can_msg
: pointer to the serialized CAN message

buff
: buffer for the serialized string

buff_len
: max length of serialized string, including terminating zero

Return

the number of written chars not including terminating zero

Header

can_vca.h

Name

vca_byte2str — converts byte to the string

Synopsis

```
const char* vca_byte2str (b,
                          base);
unsigned char  b;
int            base;
```

Arguments

b
: byte to convert

base
: base, can be (2, 8, 16)

Return

string representation of b in chosen base

Header

can_vca.h

Name

vca_str2msg — converts the string to the canmsg_t object

Synopsis

```
int vca_str2msg (can_msg,
                 str);
struct canmsg_t *  can_msg;
const char *       str;
```

Arguments

can_msg
: pointer to the serialized CAN message

str
: string representing CAN message

Return

number of read chars if succeed else zero or negative value.

Header

can_vca.h

Name

vca_cmp_terminated — compares two strings terminated either by '\0' or by `terminator`.

Synopsis

```
int vca_cmp_terminated (pa,
                        pb,
                        terminator);
const char *  pa;
const char *  pb;
char          terminator;
```

Arguments

pa
: first string

pb
: second string

terminator
: aditional char (\0 stil terminates string too), that indicates end of string

Description

Usefull when one works with the path names.

Return

the same value like libc strcmp does.

Header

can_vca.h

Name

vca_log — generic logging facility for VCA library

Synopsis

```
void vca_log (domain,
              level,
              format,
              ...);
```

const char * *domain*;
int *level*;
const char * *format*;
... *...*;

Arguments

domain
: pointer to character string representing source of logged event, it is `VCA_LDOMAIN` for library itself

level
: severity level

format
: printf style format followed by arguments

...
: variable arguments

Description

This functions is used for logging of various events. If not overridden by application, logged messages goes to the stderr. Environment variable `VCA_LOG_FILENAME` can be used to redirect output to file. Environment variable `VCA_DEBUG_FLG` can be used to select different set of logged events through vca_debug_flg.

Note

There is a global variable `vca_log_cutoff_level`. Only the messages with `level` <= `vca_log_cutoff_level` will be logged. see `can_vca`.h

---

Name

vca_log_redir — redirects default log output function

Synopsis

```
void vca_log_redir (log_fnc,
                    add_flags);
```

vca_log_fnc_t * *log_fnc*;

int *add_flags*;

Arguments

log_fnc
: new log output function. Value NULL resets to default function

add_flags
: some more flags

SDO processing API

Name

struct vcasdo_fsm_t — structure representing SDO FSM

Synopsis

```
struct vcasdo_fsm_t {
  unsigned srvcli_cob_id;
  unsigned clisrv_cob_id;
  unsigned node;
  unsigned index, subindex;
  struct timeval last_activity;
  int bytes_to_load;
  unsigned char toggle_bit;
  char is_server;
  char is_uploader;
  int state;
  vcasdo_fsm_state_fnc_t * statefnc;
  int err_no;
  ul_dbuff_t data;
  canmsg_t out_msg;
};
```

Members

srvcli_cob_id
: SDO server-client COB_ID (default is 0x580 + node), port on which master listen

clisrv_cob_id
: SDO client-server COB_ID (default is 0x600 + node), port on which slave listen

node
: CANopen node number

subindex
: subindex of communicated object

last_activity
: time of last FSM activity (internal use)

bytes_to_load
: number of stil not uploaded SDO data bytes (internal use)

toggle_bit

(internal use)

is_server
: type of FSM client or server (Master or Slave) (internal use)

is_uploader
: processing upload/download in state sdofsmRun, sdofsmDone

state
: state of SDO (`sdofsmIdle` = `0`, `sdofsmRun`, `sdofsmDone`, `sdofsmError`, `sdofsmAbort`)

statefnc
: pointer to the state function (internal use)

err_no
: error number in state `sdofsmError`.

data
: uploaded/downloaded bytes (see `ul_dbuff`.h)

out_msg
: if `vcasdo_taste_msg` generates answer, it is stored in the `out_msg`

Header

vcasdo_fsm.h

Name

vcasdo_fsm_upload1 — starts SDO upload using parameters set by previous calling `vcasdo_init_fsm`

Synopsis

`int` **`vcasdo_fsm_upload1`** `(`*fsm*`)`;
vcasdo_fsm_t * *fsm*;

Arguments

`fsm`
: FSM to work with

Return

the same as `vcasdo_fsm_upload1`

See also

`vcasdo_fsm_upload1`.

Header

vcasdo_fsm.h

Name

vcasdo_fsm_download1 — starts SDO download using parameters set by previous calling `vcasdo_init_fsm`

Synopsis

```
int vcasdo_fsm_download1 (fsm,
                          data);
```

vcasdo_fsm_t * *fsm*;

ul_dbuff_t * *data*;

Arguments

`fsm`
: FSM to work with

`data`
: pointer to &ul_dbuff_t structure where downloaded data will be stored

Return

the same as `vcasdo_fsm_download`

See also

`vcasdo_fsm_download`.

Header

vcasdo_fsm.h

Name

vcasdo_read_multiplexor — reads index and subindex from multiplexor part of CANopen mesage

Synopsis

```
void vcasdo_read_multiplexor (mult,
                              index,
                              subindex);
```

const byte * *mult*;
unsigned * *index*;
unsigned * *subindex*;

Arguments

`mult`
: pointer to the multiplexor part of CANopen mesage

index
: pointer to place to store read index

subindex
: pointer to place to store read subindex

Header

vcasdo_fsm.h

Name

vcasdo_error_msg — translates err_no to the string message

Synopsis

const char* **vcasdo_error_msg** (*err_no*);
int err_no;

Arguments

err_no
: number of error, if FSM state == sdofsmError

Return

textual error description.

Header

vcasdo_fsm.h

Name

vcasdo_init_fsm — init SDO FSM

Synopsis

```
void vcasdo_init_fsm (fsm,
                      srvcli_cob_id,
                      clisrv_cob_id,
                      node);
```

vcasdo_fsm_t *	*fsm*;
unsigned	*srvcli_cob_id*;
unsigned	*clisrv_cob_id*;
unsigned	*node*;

Arguments

fsm
: fsm to init

srvcli_cob_id
: port to use for server->client communication (default 0x850 used if `srvcli_cob_id==0`)

clisrv_cob_id
: port to use for client->server communication (default 0x600 used if `clisrv_cob_id==0`)

node
: number of node on CAN bus to communicate with

Header

vcasdo_fsm.h

Name

vcasdo_destroy_fsm — frees all SDO FSM resources (destructor)

Synopsis

`void` **`vcasdo_destroy_fsm`** `(`*fsm*`);`
vcasdo_fsm_t * *fsm*;

Arguments

fsm
: fsm to destroy

Header

vcasdo_fsm.h

Name

vcasdo_fsm_idle — sets SDO FSM to idle state

Synopsis

`void` **`vcasdo_fsm_idle`** `(`*fsm*`);`
vcasdo_fsm_t * `fsm`;

Arguments

fsm
: SDO FSM

Header

vcasdo_fsm.h

Name

vcasdo_fsm_run — starts SDO communication protocol for this FSM

Synopsis

```
void vcasdo_fsm_run (fsm);
vcasdo_fsm_t * fsm;
```

Arguments

fsm
: SDO FSM

Header

vcasdo_fsm.h

Name

vcasdo_fsm_abort — aborts SDO communication for this FSM, fill abort out_msg

Synopsis

```
void vcasdo_fsm_abort (fsm,
                       abort_code);
vcasdo_fsm_t * fsm;
uint32_t       abort_code;
```

Arguments

fsm
: SDO FSM

abort_code
: code to fill to out_msg

Header

vcasdo_fsm.h

Name

vcasdo_fsm_upload — starts upload SDO communication protocol for this FSM

Synopsis

```
int vcasdo_fsm_upload (fsm,
                       node,
                       index,
                       subindex,
                       srvcli_cob_id,
                       clisrv_cob_id);
vcasdo_fsm_t *  fsm;
int             node;
unsigned        index;
byte            subindex;
unsigned        srvcli_cob_id;
unsigned        clisrv_cob_id;
```

Arguments

fsm
: SDO FSM

node
: CANopen device node to upload from

index
: uploaded object index

subindex
: uploaded object subindex

srvcli_cob_id
: port to use for server->client communication (default 0x850 used if `srvcli_cob_id`==0)

clisrv_cob_id
: port to use for client->server communication (default 0x600 used if `clisrv_cob_id`==0)

Return

not 0 if `fsm`->out_msg contains CAN message to sent

Header

vcasdo_fsm.h

Name

vcasdo_fsm_download — starts download SDO communication protocol for this FSM

Synopsis

```
int vcasdo_fsm_download (fsm,
```

dbuff,
node,
index,
subindex,
srvcli_cob_id,
clisrv_cob_id);

vcasdo_fsm_t *	*fsm*;
ul_dbuff_t *	*dbuff*;
int	*node*;
unsigned	*index*;
byte	*subindex*;
unsigned	*srvcli_cob_id*;
unsigned	*clisrv_cob_id*;

Arguments

fsm
: SDO FSM

dbuff
: pointer to a ul_dbuff structure to store received/transmitted data

node
: CANopen device node to upload from

index
: uploaded object index

subindex
: uploaded object subindex

srvcli_cob_id
: port to use for server->client communication (default 0x850 used if `srvcli_cob_id`==0)

clisrv_cob_id
: port to use for client->server communication (default 0x600 used if `clisrv_cob_id`==0)

Return

not 0 if `fsm`->out_msg contains CAN message to sent

Header

vcasdo_fsm.h

Name

vcasdo_fsm_taste_msg — try to process msg in FSM

Synopsis

```
int vcasdo_fsm_taste_msg (fsm,
                          msg);
vcasdo_fsm_t *        fsm;
const canmsg_t *      msg;
```

Arguments

fsm
: fsm to process msg

msg
: tasted msg

Return

0 if msg is not eatable for FSM, -1 if message has correct CobID but cann't be processed in current FSM state, 1 if message is processed,

Header

vcasdo_fsm.h

Name

vcasdo_abort_msg — translates SDO abort_code to the string message

Synopsis

```
const char* vcasdo_abort_msg (abort_code);
uint32_t      abort_code;
```

Arguments

abort_code
: abort code

Header

`vcasdo_msg.h`

PDO processing API

Name

struct vcapdo_mapping_t — structure representing mapping of sigle object in PDO

Synopsis

```
struct vcapdo_mapping_t {
  vcaod_object_t * object;
  unsigned char   start;
  unsigned char   len;
  sui_dinfo_t * dinfo;
};
```

Members

object
: pointer to the mapped object

start
: bit offset of object value in PDO

len
: bit length of object value in PDO

dinfo
: pointer to object data source. Every PDO can be read/written through *dinfo* to the OD or to hardware. Actualy there is no other way for PDO object to do that.

Header

vca_pdo.h

Name

struct vcapdolst_object_t — structure representing single PDO object

Synopsis

```
struct vcapdolst_object_t {
  gavl_node_t my_node;
  struct vcaPDOProcessor_t * pdo_processor;
  unsigned long cob_id;
  unsigned char transmition_type;
  unsigned flags;
  unsigned char sync_every;
  unsigned char sync_counter;
  uint16_t inhibit_time;
  uint16_t event_timer;
  unsigned char * pdo_buff;
  int mapped_cnt;
```

```
  vcapdo_mapping_t * mapped_objects;
  evc_rx_hub_t rx_hub;
};
```

Members

my_node
: structure necessary for storing node in GAVL tree

pdo_processor
: pointer to PDO processor servicing this PDO

cob_id
: COB ID of PDO

transmition_type
: type of PDO transmission according to DS301 table 55

flags
: PDO characteristics and parsed transmission_type

sync_every
: synchronous PDO will be processed every n-th SYNC message

sync_counter
: auxiliary variable for `sync_every`

inhibit_time
: minimum gap between two PDO transmissions (multiples of 100 us)

event_timer
: if nonzero, PDO is transmitted every *`event_timer`* ms. Valid only in transmission modes 254, 255. (!vcapdoFlagSynchronous && !vcapdoFlagRTROnly)

pdo_buff
: buffer for received/transmitted PDO

mapped_cnt
: number of mapped objects in OD

mapped_objects
: array to structures describing mapping details for all mapped objects

rx_hub
: If PDO communication is event driven, appropriate events are connected to this hub

See also

GAVL usage (`ul_gavlchk`.c)

Header

vca_pdo.h

Name

struct vcapdolst_root_t — structure representing root of OD

Synopsis

```
struct vcapdolst_root_t {
  gavl_node_t * my_root;
};
```

Members

my_root
: object dictionary GAVL tree root

See also

GAVL usage (`ul_gavlchk`.c)

Header

vca_pdo.h

Name

struct vcaPDOProcessor_t — structure used for PDO communication

Synopsis

```
struct vcaPDOProcessor_t {
  vcapdolst_root_t pdolst_root;
  // TODO send_to_can_fnc: remove this hack and add queue of outcoming
CAN messages// to make this library thread safe.// At present
send_to_can_fnc should be thread safe.vcapdo_send_to_can_fnc_t *;
  vcaod_root_t * od_root;
  //vcaDinfoManager_t * dinfo_mgr;
  int node_id;
};
```

Members

pdolst_root
: GAVL containing all defined &vcapdolst_object_t structures

send_to_can_fnc
: PDOProcessor should use this function if it needs to send CAN message during processing

od_root
: pointer to used OD (necessary for PDOs creation and initialization in `vcaPDOProcessor_createPDOLIst`)

dinfo_mgr
: pointer to used DinfoManager (providing HW dinfos during initialization)

node_id
: Node number, optional parameter, if it is specified, default PDO COB-IDs can be assigned if they are not specified in EDS. If *`node_id`* is 0, then it is ignored.

Description

vcaPDOProcessor is responsible for all PDO related tasks in CANopen device

Header

vca_pdo.h

Name

vcaPDOProcessor_init — vcaPDOProcessor constructor

Synopsis

```
void vcaPDOProcessor_init (proc);
vcaPDOProcessor_t * proc;
```

Arguments

proc
: pointer to PDO processor to work with

Header

vca_pdo.h

Name

vcaPDOProcessor_destroy — vcaPDOProcessor destructor

Synopsis

```
void vcaPDOProcessor_destroy (proc);
vcaPDOProcessor_t * proc;
```

Arguments

proc
: pointer to PDO processor to work with

Description

It releases all PDO objects

Header

vca_pdo.h

Name

vcaPDOProcessor_setOD — assign OD to PDOProcessor

Synopsis

```
void vcaPDOProcessor_setOD (proc,
                            od_root);
vcaPDOProcessor_t *   proc;
vcaod_root_t *        od_root;
```

Arguments

proc
: pointer to PDO processor to work with

od_root
: assigned root of Object Dictionary

Header

vca_pdo.h

Name

vcaPDOProcessor_createPDOList — scans OD and creates all valid PDO structures.

Synopsis

```
int vcaPDOProcessor_createPDOList (proc);
vcaPDOProcessor_t *   proc;
```

Arguments

proc
: pointer to PDO processor to work with

Description

It also deletes previously created PDO structures (if any).

Return

0 or negative number in case of an error

Header

vca_pdo.h

Name

_vcaPDOProcessor_disconnectDinfoLinks — disconnect all PDOs and their dinfo structures

Synopsis

```
void _vcaPDOProcessor_disconnectDinfoLinks (proc);
vcaPDOProcessor_t * proc;
```

Arguments

proc
: pointer to PDO processor to work with

Description

Actualy it only decrements RefCnt, so only dinfos with RefCnt==1 will be deleted

Note

this function is internal and it is not a part of VCA PDO public interface.

Header

vca_pdo.h

Name

vcaPDOProcessor_connectDinfoLinks — scans defined PDOs and makes necessary data links from PDOs to OD and HW

Synopsis

```
void vcaPDOProcessor_connectDinfoLinks (proc);
vcaPDOProcessor_t * proc;
```

Arguments

proc
: pointer to PDO processor to work with

Description

Disconnect all connected dinfos. For each mapped object tries to find appropriate dinfo asking DinfoManager. If DinfoManager returns NULL, thats means, that no HW is connected to this object. In such case function creates dbuff_dinfo for data stored in OD and connect it to mapped PDO.

Header

vca_pdo.h

Name

vcaPDOProcessor_processMsg — tries to process *msg*

Synopsis

`int` **`vcaPDOProcessor_processMsg`** `(`*proc*, *msg*`);`

vcaPDOProcessor_t * *proc*;

canmsg_t * *msg*;

Arguments

proc
: pointer to PDO processor to work with

msg
: CAN msg to proceed

Return

zero if msg is processed

Header

vca_pdo.h

OD access API

Name

struct vcaod_root_t — structure representing root of OD

Synopsis

```
struct vcaod_root_t {
  gsa_array_field_t my_root;
};
```

Members

my_root
: object dictionary GAVL tree root

Header

vca_od.h

Name

struct vcaod_object_t — structure representing single object in OD

Synopsis

```
struct vcaod_object_t {
  #ifndef CONFIG_OD_GSAgavl_node_t my_node;
  #endifunsigned index;
  int subindex;
  unsigned char data_type;
  unsigned object_type;
  int access;
  unsigned flags;
  char * name;
  struct vcaod_object_t * subobjects;
  int subcnt;
  vcaod_dbuff_t value;
  sui_dinfo_t * dinfo;
};
```

Members

my_node
: structure neccessary for storing node in GAVL tree, is NULL for subindicies

index
: index of object

subindex
: subindex of subobject or -1 if object is not subobject

data_type
: can be one of (BOOLEAN, INTEGER8, ...)

object_type
: type of object (DOMAIN=2, DEFTYPE=5, DEFSTRUCT=6, VAR=7, ARRAY=8, RECORD=9)

access
: access attributes (RW, WO, RO, CONST)

flags
: flags can be: `VCAOD_OBJECT_FLAG_MANDATORY` object is mandatory/optional, `VCAOD_OBJECT_FLAG_PDO_MAPPING` object is supposed to be PDO mapped, `VCAOD_OBJECT_FLAG_WEAK_DINFO` *`dinfo`* is weak pointer

name
: textual name of object

subobjects
: pointer to array of subobjects (definition==DEFSTRUCT, RECORD) or NULL

subcnt
: number of subobjects

value
: object values (definition==ARRAY) or single value (other definitions). If definition==ARRAY all values have the same length and they are stored sequently in `value`

dinfo
: Reference to dinfo associated with current object. There are couple of reasons for such a association. 1. Object is PDO mapped but its value doesn't come from HW dinfo (it is not tecnological value) - in such a case dbuff dinfo is created and referenced from that OD object. 2. Object is PDO mapped and its value comes from HW dinfo (it is tecnological value) - in such a case only weak reference is in OD object. When HW module is unloaded or dinfo will be destroyed from any reason, also weak reference to it will be cleared to NULL. 3. Object is not PDO mapped but its value comes from HW dinfo - in such a case even SDO communication sholud read that dinfo to get the propper object value.

Header

vca_od.h

Name

vcaod_find_object — finds object in OD. This function is not a part of the SDO API

Synopsis

```
vcaod_object_t* vcaod_find_object (odroot,
                                   ix,
                                   subix,
                                   abort_code);
```

vcaod_root_t *	*odroot*;
unsigned	*ix*;
unsigned	*subix*;
uint32_t *	*abort_code*;

Arguments

odroot
: object dictionary

ix
: object index

subix
: object subindex, ignored if object does not have subobjects

abort_code

Pointer to the abort code in case of an ERROR. It can be NULL, than it is ignored. Abort codes are defined in CANopen standart 301 and can be translated to text calling `vcasdo_abort_msg`.

Return

found object or NULL

Header

vca_od.h

Name

vcaod_get_value — reads object value from Object Dictionary and copies them to caller buffer

Synopsis

```
int vcaod_get_value (object,
                     array_index,
                     buff,
                     len,
                     abort_code);
```

const vcaod_object_t*	*object*;
int	*array_index*;
void *	*buff*;
int	*len*;
uint32_t *	*abort_code*;

Arguments

object
: object from dictionary, see. `vcaod_find_object`

array_index
: if object is an array `array_index` specifies which index to get, othervise it is ignored.

buff
: buffer to write requested data

len
: length of the buffer

abort_code
: Pointer to the abort code in case of an ERROR. It can be NULL, than it is ignored. Abort codes are defined in CANopen standart 301 and can be translated to text calling `vcasdo_abort_msg`.

Return

number of read bytes negative value in case of an error

Header

vca_od.h

Name

vcaod_set_value — copies object value from caller's buffer to Object Dictionary

Synopsis

```
int vcaod_set_value (object,
                     array_index,
                     buff,
                     len,
                     abort_code);
```

vcaod_object_t*	*object*;
int	*array_index*;
const void *	*buff*;
int	*len*;
uint32_t *	*abort_code*;

Arguments

`object`
: object from dictionary, see. `vcaod_find_object`

`array_index`
: if object is an array, array_index, tells which item to get, in other case it is simply ignored.

`buff`
: buffer containing written data

`len`
: length of the data

`abort_code`
: area to fill the abort code in case of an ERROR. It can be NULL, than it is ignored. Abort codes are defined in CANopen standart 301 and can be translated to text calling `vcasdo_abort_msg`.

Description

Function sets whole buffer to zeros before it starts to copy object data to it, even if buffer is larger than data.

Return

number of stored data bytes negative value in case of an error

Header

vca_od.h

Name

vcaod_get_object_data_size — get size of object in bytes

Synopsis

`int` **`vcaod_get_object_data_size`** `(`*object*, *abort_code*`);`

const vcaod_object_t *	*object*;
uint32_t *	*abort_code*;

Arguments

`object`
: object from dictionary, see. `vcaod_find_object`

`abort_code`
: area to fill the abort code in case of an ERROR. It can be NULL, than it is ignored. Abort codes are defined in CANopen standart 301 and can be translated to text calling `vcasdo_abort_msg`.

Return

number of stored data bytes negative value in case of an error

Header

vca_od.h

Name

od_item_set_value_as_str — set object value from its string representation.

Synopsis

`int` **`od_item_set_value_as_str`** `(`*item*, *valstr*`);`

vcaod_object_t *	*item*;
const char*	*valstr*;

Arguments

item
: object to set

valstr
: string representation of object value

Return

negative value in case of an error

Header

vca_od.h

Name

vcaod_od_free — release all OD memory

Synopsis

`void` **`vcaod_od_free`** `(`*odroot*`);`
vcaod_root_t * `odroot`;

Arguments

odroot
: pointer to the object dictionary root

Header

vca_od.h

Name

vcaod_dump_od — debug function, dumps OD to log

Synopsis

`void` **`vcaod_dump_od`** `(`*odroot*`);`
vcaod_root_t * `odroot`;

Arguments

odroot
: root, which contains OD

Header

vca_od.h

Name

vcaod_get_dinfo_ref — returns reference to dinfo corresponting to *obj*

Synopsis

`sui_dinfo_t * vcaod_get_dinfo_ref (obj, create_weak);`

vcaod_object_t *	*obj*;
int	*create_weak*;

Arguments

obj
: object from OD

create_weak
: if there is no HW dinfo for object, creates temporary dbuff dinfo

Description

If *obj* allready has its &dinfo assigned `vcaod_get_dinfo_ref` returns this pointer, if it is not function creates new &dinfo object.

Return

pointer to associated dinfo with reference count increased or NULL if creation fails

Header

vca_od.h

libulut API

Name

ul_dbuff_init — init memory allocated for dynamic buffer

Synopsis

`int ul_dbuff_init (buf, flags);`

ul_dbuff_t *	*buf*;
int	*flags*;

Arguments

buf
: buffer structure

flags
: flags describing behaviour of the buffer only UL_DBUFF_IS_STATIC flag is supported. in this case buffer use unly static array sbuf

Description

Returns capacity of initialised buffer

Name

ul_dbuff_destroy — frees all resources allocated by buf

Synopsis

```
void ul_dbuff_destroy (buf);
ul_dbuff_t * buf;
```

Arguments

buf
: buffer structure

Name

ul_dbuff_prep — sets a new len and capacity of the buffer

Synopsis

```
int ul_dbuff_prep (buf,
                   new_len);
ul_dbuff_t * buf;
int          new_len;
```

Arguments

buf
: buffer structure

new_len
: new desired buffer length

Description

Returns new buffer length

Name

struct ul_dbuff_t — Generic Buffer for Dynamic Data

Synopsis

```
struct ul_dbuff_t {
  unsigned long len;
  unsigned long capacity;
  int flags;
  unsigned char * data;
  unsigned char * sbuff;
};
```

Members

len
: actual length of stored data

capacity
: capacity of allocated buffer

flags
: only one flag (`UL_DBUFF_IS_STATIC`) used now

data
: pointer to dynamically allocated buffer

sbuff
: static buffer for small data sizes

Name

ul_dbuff_set_capacity — change capacity of buffer to at least *`new_capacity`*

Synopsis

`int` **`ul_dbuff_set_capacity`** `(`*buf*, *new_capacity*`);`

ul_dbuff_t * *buf*;

int *new_capacity*;

Arguments

`buf`
: buffer structure

`new_capacity`
: new capacity

Description

Returns real capacity of reallocated buffer

Name

ul_dbuff_set_len — sets a new len of the buffer, change the capacity if neccessary

Synopsis

```
int ul_dbuff_set_len (buf,
                      new_len);
ul_dbuff_t * buf;
int          new_len;
```

Arguments

buf
: buffer structure

new_len
: new desired buffer length

Description

Returns new buffer length

Name

ul_dbuff_set — copies bytes to buffer and change its capacity if neccessary like memset

Synopsis

```
int ul_dbuff_set (buf,
                  b,
                  n);
ul_dbuff_t * buf;
byte         b;
int          n;
```

Arguments

buf
: buffer structure

b
: appended bytes

n
: number of apended bytes

Returns

length of buffer

Name

ul_dbuff_cpy — copies bytes to buffer and change its capacity if neccessary

Synopsis

```
int ul_dbuff_cpy (buf,
                  b,
                  n);
ul_dbuff_t *  buf;
const void *  b;
int           n;
```

Arguments

buf
: buffer structure

b
: appended bytes

n
: number of apended bytes

Returns

length of buffer

Name

ul_dbuff_cat — appends bytes at end of buffer and change its capacity if neccessary

Synopsis

```
int ul_dbuff_cat (buf,
                  b,
                  n);
ul_dbuff_t *  buf;
const void *  b;
int           n;
```

Arguments

buf
: buffer structure

b

appended bytes

n

number of apended bytes

Returns

length of buffer

Name

ul_dbuff_strcat — appends str at dhe end of buffer and change its capacity if neccessary

Synopsis

`int` **`ul_dbuff_strcat`** `(`*buf*, *str*`)`;

ul_dbuff_t * *buf*;
const char * *str*;

Arguments

buf

buffer structure

str

string to append

Description

Returns number length of buffer (including terminating '\0')

Name

ul_dbuff_strcpy — copy str to the buffer and change its capacity if neccessary

Synopsis

`int` **`ul_dbuff_strcpy`** `(`*buf*, *str*`)`;

ul_dbuff_t * *buf*;
const char * *str*;

Arguments

buf

buffer structure

str

string to copy

Description

Returns number length of buffer (including terminating '\0')

Name

ul_dbuff_append_byte — appends byte at dhe end of buffer and change its capacity if neccessary

Synopsis

```
int ul_dbuff_append_byte (buf,
                          b);
```

ul_dbuff_t * *buf*;
unsigned char *b*;

Arguments

buf
: buffer structure

b
: appended byte

Description

Returns number length of buffer (including terminating '\0')

Name

ul_dbuff_ltrim — remove all white space characters from the left

Synopsis

```
int ul_dbuff_ltrim (buf);
```

ul_dbuff_t * `buf`;

Arguments

buf
: buffer structure

Return

new length of buffer

Name

ul_dbuff_rtrim — remove all white space characters from the right

Synopsis

`int` **`ul_dbuff_rtrim`** `(`*buf*`);`
ul_dbuff_t * `buf`;

Arguments

`buf`
: buffer structure

Description

if buffer is terminated by '\0', than is also terminated after rtrim

Return

new length of buffer

Name

ul_dbuff_trim — remove all white space characters from the right and from the left

Synopsis

`int` **`ul_dbuff_trim`** `(`*buf*`);`
ul_dbuff_t * `buf`;

Arguments

`buf`
: buffer structure

Description

Returns number length of buffer (including terminating '\0')

Name

ul_dbuff_cpos — searches string for char

Synopsis

`int` **`ul_dbuff_cpos`** `(`*buf*,
what,
quote`);`

const ul_dbuff_t * *buf*;
unsigned char *what*;
unsigned char *quote*;

Arguments

buf
: searched dbuff

what
: char to find

quote
: skip str areas quoted in quote chars
 If you want to ignore quotes assign '\0' to quote in function call

Return

position of what char or negative value

Name

ul_str_cpos — searches string for char

Synopsis

```
int ul_str_cpos (str,
                 what,
                 quote);
const unsigned char *  str;
unsigned char          what;
unsigned char          quote;
```

Arguments

str
: zero terminated string

what
: char to find

quote
: skip str areas quoted in quote chars If you want to ignore quotes assign '\0' to quote in function call

Return

position of what char or negative value

Name

ul_str_pos — searches string for substring

Synopsis

```
int ul_str_pos (str,
                what,
```

quote);

const unsigned char * *str*;
const unsigned char * *what*;
unsigned char *quote*;

Arguments

str
: zero terminated string

what
: string to find

quote
: skip str areas quoted in quote chars If you want to ignore quotes assign '\0' to quote in function call

Return

position of what string or negative value

Name

ul_str_ncpy — copies string to the buffer

Synopsis

```
int ul_str_ncpy (to,
                 from,
                 buff_size);
unsigned char *       to;
const unsigned char * from;
int                   buff_size;
```

Arguments

to
: buffer where to copy str

from
: zero terminated string

buff_size
: size of the *to* buffer (including terminating zero)

Description

Standard strncpy function have some disadvatages (ie. do not append term. zero if copied string doesn't fit in to buffer, fills whole rest of buffer with zeros)

Returns strlen(to) or negative value in case of error

Name

ul_dbuff_log_hex — writes content of dbuff to log

Synopsis

```
void ul_dbuff_log_hex (buf, log_level);
ul_dbuff_t * buf;
int log_level;
```

Arguments

buf
: buffer structure

log_level
: logging level

Name

ul_dbuff_cut_pos — cut first *n* bytes from *fromdb* and copies it to *todb*.

Synopsis

```
void ul_dbuff_cut_pos (fromdb, todb, n);
ul_dbuff_t * fromdb;
ul_dbuff_t * todb;
int n;
```

Arguments

fromdb
: buffer to cut from

todb
: buffer to copy to

n
: position where to cut

Description

If *n* is greater than fromdb.len whole *fromdb* is copied to *todb*. If *n* is negative position to cut is counted from the end of *fromdb*. If *n* is zero *fromdb* stays unchanged and todb is resized to len equal zero.

Name

ul_dbuff_cut_delimited — cuts bytes before delimiter + delimiter char from *fromdb* and copies tham to the *todb*

Synopsis

```
void ul_dbuff_cut_delimited (fromdb,
                             todb,
                             delimiter,
                             quote);
ul_dbuff_t * fromdb;
ul_dbuff_t * todb;
char         delimiter;
char         quote;
```

Arguments

fromdb
: buffer to cut from

todb
: buffer to copy to

delimiter
: delimiter char

quote
: quoted delimiters are ignored, *quote* can be '\0', than it is ignored.

Description

If *fromdb* doesn't contain delimiter *todb* is trimmed to zero length.

Name

ul_dbuff_cut_token — cuts not whitespaces from `fromdb` to `todb`.

Synopsis

```
void ul_dbuff_cut_token (fromdb,
                         todb);
ul_dbuff_t * fromdb;
ul_dbuff_t * todb;
```

Arguments

fromdb
: buffer to cut from

todb
: buffer to copy to

Description

Leading whitespaces are ignored. Cut string is trimmed.

Name

evc_link_init — Initialize Event Connector Link

Synopsis

```
int evc_link_init (link);
evc_link_t * link;
```

Arguments

link
: pointer to the link

Description

Link reference count is set to 1 by this function

Return Value

negative value informs about failure.

Name

evc_link_new — Allocates New Event Connector Link

Synopsis

```
evc_link_t * evc_link_new (void);
void;
```

Arguments

void
: no arguments

Description

Link reference count is set to 1 by this function

Return Value

pointer to the new link or `NULL`.

Name

evc_link_connect — Connects Link between Two Hubs

Synopsis

```
int evc_link_connect (link,
                      src,
                      dst,
                      prop);
evc_link_t *       link;
evc_tx_hub_t *     src;
evc_rx_hub_t *     dst;
evc_prop_fnc_t *   prop;
```

Arguments

link
: pointer to the non-connected initialized link

src
: pointer to the source hub of type &evc_tx_hub_t

dst
: pointer to the destination hub of type &evc_rx_hub_t

prop
: propagation function corresponding to source and destination expected event arguments

Description

If ready flag is not set, link state is set to ready and reference count is increased.

Return Value

negative return value indicates fail.

Name

evc_link_init_standalone — Initialize Standalone Link

Synopsis

```
int evc_link_init_standalone (link,
                              rx_fnc,
                              context);
evc_link_t *   link;
```

evc_rx_fnc_t * *rx_fnc*;
void * *context*;

Arguments

link
: pointer to the link

rx_fnc
: pointer to the function invoked by event reception

context
: context for the `rx_fnc` function invocation

Description

Link reference count is set to 1 by this function

Return Value

negative value informs about failure.

Name

evc_link_new_standalone — Allocates New Standalone Link

Synopsis

```
evc_link_t * evc_link_new_standalone (rx_fnc,
                                      context);
```

evc_rx_fnc_t * *rx_fnc*;
void * *context*;

Arguments

rx_fnc
: callback function invoked if event is delivered

context
: context provided to the callback function

Description

Link reference count is set to 1 by this function

Return Value

pointer to the new link or `NULL`.

Name

evc_link_connect_standalone — Connects Standalone Link to Source Hubs

Synopsis

```
int evc_link_connect_standalone (link,
                                 src,
                                 prop);
```

evc_link_t * *link*;
evc_tx_hub_t * *src*;
evc_prop_fnc_t * *prop*;

Arguments

link
: pointer to the non-connected initialized link

src
: pointer to the source hub of type &evc_tx_hub_t

prop
: propagation function corresponding to hub source and standalone `rx_fnc` expected event arguments

Description

If ready flag is not set, link state is set to ready and reference count is increased.

Return Value

negative return value indicates failure.

Name

evc_link_delete — Deletes Link from Hubs Lists

Synopsis

```
int evc_link_delete (link);
```

evc_link_t * `link`;

Arguments

link
: pointer to the possibly connected initialized link

Description

If ready flag is set, link ready flag is cleared and reference count is decreased. This could lead to link disappear, if nobody is holding reference.

Return Value

positive return value indicates immediate delete, zero return value informs about delayed delete.

Name

evc_link_dispose — Disposes Link

Synopsis

`void` **`evc_link_dispose`** `(`*link*`);`
evc_link_t * `link`;

Arguments

`link`
: pointer to the possibly connected initialized link

Description

Deletes link from hubs, marks it as dead, calls final death `propagate` for the link and if link is *`malloced`*, releases link occupied memory.

Name

evc_tx_hub_init — Initializes Event Transmition Hub

Synopsis

`int` **`evc_tx_hub_init`** `(`*hub*`);`
evc_tx_hub_t * `hub`;

Arguments

`hub`
: pointer to the &evc_tx_hub_t type hub

Return Value

negative return value indicates failure.

Name

evc_tx_hub_done — Initializes Event Transmition Hub

Synopsis

`void` **`evc_tx_hub_done`** `(`*hub*`);`
evc_tx_hub_t * `hub`;

Arguments

hub
: pointer to the &evc_tx_hub_t type hub

Name

evc_tx_hub_propagate — Propagate Event to Links Destinations

Synopsis

```
void evc_tx_hub_propagate (hub,
                           args);
evc_tx_hub_t * hub;
va_list        args;
```

Arguments

hub
: pointer to the &evc_tx_hub_t type hub

args
: pointer to the variable arguments list

Description

The function propagates event to the connected links, it skips links marked as *dead*, *blocked* or *delete_pend*. If the link is not marked as *recursive*, it ensures, that link is not called twice.

Name

evc_tx_hub_emit — Emits Event to Hub

Synopsis

```
void evc_tx_hub_emit (hub,
                      ...);
evc_tx_hub_t * hub;
...            ...;
```

Arguments

hub
: pointer to the &evc_tx_hub_t type hub

...
: variable arguments

Description

The function hands over arguments to `evc_tx_hub_propagate` as &va_list.

Name

evc_rx_hub_init — Initializes Event Receiption Hub

Synopsis

```
int evc_rx_hub_init (hub,
                     rx_fnc,
                     context);
evc_rx_hub_t *  hub;
evc_rx_fnc_t *  rx_fnc;
void *          context;
```

Arguments

hub
: pointer to the &evc_rx_hub_t type hub

rx_fnc
: pointer to the function invoked by event reception

context
: context for the `rx_fnc` function invocation

Return Value

negative return value indicates failure.

Name

evc_rx_hub_done — Finalize Event Receiption Hub

Synopsis

```
void evc_rx_hub_done (hub);
evc_rx_hub_t * hub;
```

Arguments

hub
: pointer to the &evc_rx_hub_t type hub

Name

struct evc_link — Event Connector Link

Synopsis

```
struct evc_link {
  struct dst;
  evc_prop_fnc_t * propagate;
  int refcnt;
  unsigned recursive:1;
  unsigned blocked:1;
  unsigned ready:1;
  unsigned dead:1;
  unsigned delete_pend:1;
  unsigned malloced:1;
  unsigned standalone:1;
  unsigned tx_full_hub:1;
  unsigned rx_full_hub:1;
  short taken;
};
```

Members

dst
: determines destination of the event, it can be *`standalone`* `rx_fnc` function with with *`context`* or &evc_tx_hub_t in the *`multi`* case

propagate
: pointer to the arguments propagation function,

refcnt
: link reference counter

recursive
: link can propagate could be invoked recursively, else recursive events are ignored by link

blocked
: event propagation is blocked for the link, can be used by application

ready
: link is ready and has purpose to live - it connects two active entities

dead
: link is dead and cannot propagate events

delete_pend
: link is being deleted, but it is taken simultaneously, delete has to wait for finish of the propagate and to moving to the next link

malloced
: link has been malloced and should be automatically freed when referenc counts drop to zero

standalone
: link is used for standalone function invocation

tx_full_hub
: *`src`* points to the full hub structure

rx_full_hub
: *`dst`* points to the full hub structure

taken
: link is in middle of the propagation process

Description

The link delivers events from the source to the destination. The link specific function `propagate` is called for each link leading from the hub activated by `evc_tx_hub_emit` and `evc_tx_hub_propagate`. The propagate function is responsible for parameters transformation before invocation of standalone or destination hub `rx_fnc` function.

Name

struct evc_tx_hub — Event Transmit Hub

Synopsis

```
struct evc_tx_hub {
  ul_list_head_t links;
};
```

Members

links
: list of links outgoing from the hub

Name

struct evc_rx_hub — Event Receiving Hub

Synopsis

```
struct evc_rx_hub {
  ul_list_head_t links;
  evc_rx_fnc_t * rx_fnc;
  void * context;
};
```

Members

links
: list of links incoming to the hub

rx_fnc
: function invoked when event arives

context
: context for `rx_fnc`

Name

evc_link_inc_refcnt — Increment Link Reference Count

Synopsis

`void` **`evc_link_inc_refcnt`** `(`*link*`);`
evc_link_t * *link*;

Arguments

`link`
: pointer to link

Name

evc_link_dec_refcnt — Decrement Link Reference Count

Synopsis

`void` **`evc_link_dec_refcnt`** `(`*link*`);`
evc_link_t * *link*;

Arguments

`link`
: pointer to link

Description

if the link reference count drops to 0, link is deleted from hubs by `evc_link_dispose` function and if *`malloced`* is sed, link memory is disposed by `free`. Special handlink can be achieved if `propagate` returns non-zero value if called with *ded* link.

Name

gavl_first_node — Returns First Node of GAVL Tree

Synopsis

`gavl_node_t *` **`gavl_first_node`** `(`*root*`);`
const gavl_root_t * *root*;

Arguments

`root`
: GAVL tree root

Return Value

pointer to the first node of tree according to ordering

Name

gavl_last_node — Returns Last Node of GAVL Tree

Synopsis

`gavl_node_t *` **`gavl_last_node`** `(`*root*`);`
const gavl_root_t * *root*;

Arguments

`root`
GAVL tree root

Return Value

pointer to the last node of tree according to ordering

Name

gavl_is_empty — Check for Empty GAVL Tree

Synopsis

`int` **`gavl_is_empty`** `(`*root*`);`
const gavl_root_t * *root*;

Arguments

`root`
GAVL tree root

Return Value

returns non-zero value if there is no node in the tree

Name

gavl_search_node — Search for Node or Place for Node by Key

Synopsis

`int` **`gavl_search_node`** `(`*root*,

key,
mode,
nodep);

const gavl_root_t * *root*;

const void * *key*;
int *mode*;
gavl_node_t ** *nodep*;

Arguments

root
GAVL tree root

key
key value searched for

mode
mode of the search operation

nodep
pointer to place for storing of pointer to found node or pointer to node which should be parent of inserted node

Description

Core search routine for GAVL trees searches in tree starting at *root* for node of item with value of item field at offset *key_off* equal to provided **key* value. Values are compared by function pointed by **cmp_fnc* field in the tree *root*. Integer *mode* modifies search algorithm: GAVL_FANY .. finds node of any item with field value **key*, GAVL_FFIRST .. finds node of first item with **key*, GAVL_FAFTER .. node points after last item with **key* value, reworded - index points at first item with higher value of field or after last item

Return Value

Return of nonzero value indicates match found. If the *mode* is ored with GAVL_FCMP, result of last compare is returned.

Name

gavl_find — Find Item for Provided Key

Synopsis

`void *` **`gavl_find`** (*root*,
key);

const gavl_root_t * *root*;

const void * *key*;

Arguments

root
: GAVL tree root

key
: key value searched for

Return Value

pointer to item associated to key value.

Name

gavl_find_first — Find the First Item with Provided Key Value

Synopsis

`void * gavl_find_first (root, key);`

const gavl_root_t * *root*;

const void * *key*;

Arguments

root
: GAVL tree root

key
: key value searched for

Description

same as above, but first matching item is found.

Return Value

pointer to the first item associated to key value.

Name

gavl_find_after — Find the First Item with Higher Key Value

Synopsis

`void * gavl_find_after (root, key);`

const gavl_root_t * *root*;

const void * *key*;

Arguments

root
: GAVL tree root

key
: key value searched for

Description

same as above, but points to item with first key value above searched *key*.

Return Value

pointer to the first item associated to key value.

Name

gavl_insert_node_at — Insert Existing Node to Already Computed Place into GAVL Tree

Synopsis

```
int gavl_insert_node_at (root,
                         node,
                         where,
                         leftright);
```

gavl_root_t *	*root*;
gavl_node_t *	*node*;
gavl_node_t *	*where*;
int	*leftright*;

Arguments

root
: GAVL tree root

node
: pointer to inserted node

where
: pointer to found parent node

leftright
: left (1) or right (0) branch

Return Value

positive value informs about success

Name

gavl_insert_node — Insert Existing Node into GAVL Tree

Synopsis

```
int gavl_insert_node (root,
                      node,
                      mode);
gavl_root_t *   root;
gavl_node_t *   node;
int             mode;
```

Arguments

root
: GAVL tree root

node
: pointer to inserted node

mode
: if mode is `GAVL_FAFTER`, multiple items with same key can be used, else strict ordering is required

Return Value

positive value informs about success

Name

gavl_insert — Insert New Item into GAVL Tree

Synopsis

```
int gavl_insert (root,
                 item,
                 mode);
gavl_root_t *   root;
void *          item;
int             mode;
```

Arguments

root
: GAVL tree root

item
: pointer to inserted item

mode

if mode is GAVL_FAFTER, multiple items with same key can be used, else strict ordering is required

Return Value

positive value informs about success, negative value indicates malloc fail or attempt to insert item with already defined key.

Name

gavl_delete_node — Deletes/Unlinks Node from GAVL Tree

Synopsis

```
int gavl_delete_node (root,
                      node);
gavl_root_t * root;
gavl_node_t * node;
```

Arguments

root
: GAVL tree root

node
: pointer to deleted node

Return Value

positive value informs about success.

Name

gavl_delete — Delete/Unlink Item from GAVL Tree

Synopsis

```
int gavl_delete (root,
                 item);
gavl_root_t * root;
void * item;
```

Arguments

root
: GAVL tree root

item
: pointer to deleted item

Return Value

positive value informs about success, negative value indicates that item is not found in tree defined by root

---

Name

gavl_delete_and_next_node — Delete/Unlink Item from GAVL Tree

Synopsis

```
gavl_node_t * gavl_delete_and_next_node (root, node);
gavl_root_t * root;
gavl_node_t * node;
```

Arguments

root
: GAVL tree root

node
: pointer to actual node which is unlinked from tree after function call, it can be unalocated or reused by application code after this call.

Description

This function can be used after call `gavl_first_node` for destructive traversal through the tree, it cannot be combined with `gavl_next_node` or `gavl_prev_node` and root is emptied after the end of traversal. If the tree is used after unsuccessful/unfinished traversal, it must be balanced again. The height differences are inconsistent in other case. If traversal could be interrupted, the function `gavl_cut_first` could be better choice.

Return Value

pointer to next node or NULL, when all nodes are deleted

Name

gavl_cut_first — Cut First Item from Tree

Synopsis

```
void * gavl_cut_first (root);
gavl_root_t * root;
```

Arguments

root
: GAVL tree root

Description

This enables fast delete of the first item without tree balancing. The resulting tree is degraded but height differences are kept consistent. Use of this function can result in height of tree maximally one greater the tree managed by optimal AVL functions.

Return Value

returns the first item or NULL if the tree is empty

Name

struct gavl_node — Structure Representing Node of Generic AVL Tree

Synopsis

```
struct gavl_node {
  struct gavl_node * left;
  struct gavl_node * right;
  struct gavl_node * parent;
  int hdiff;
};
```

Members

left
: pointer to left child or NULL

right
: pointer to right child or NULL

parent
: pointer to parent node, NULL for root

hdiff
: difference of height between left and right child

Description

This structure represents one node in the tree and links *left* and *right* to nodes with lower and higher value of order criterion. Each tree is built from one type of items defined by user. User can decide to include node structure inside item representation or GAVL can malloc node structures for each inserted item. The GAVL allocates memory space with capacity sizeof(gavl_node_t)+sizeof (void*) in the second case. The item pointer is stored following node structure (void**)(node+1);

Name

struct gavl_root — Structure Representing Root of Generic AVL Tree

Synopsis

```
struct gavl_root {
  gavl_node_t * root_node;
  int node_offs;
  int key_offs;
  gavl_cmp_fnc_t * cmp_fnc;
};
```

Members

root_node
: pointer to root node of GAVL tree

node_offs
: offset between start of user defined item representation and included GAVL node structure. If negative value is stored there, user item does not contain node structure and GAVL manages standalone ones with item pointers.

key_offs
: offset to compared (ordered) fields in the item representation

cmp_fnc
: function defining order of items by comparing fields at offset *key_offs*.

Name

gavl_node2item — Conversion from GAVL Tree Node to User Defined Item

Synopsis

void * **gavl_node2item** (*root*, *node*);

const gavl_root_t * *root*;
const gavl_node_t * *node*;

Arguments

root
: GAVL tree root

node
: node belonging to *root* GAVL tree

Return Value

pointer to item corresponding to node

Name

gavl_node2item_safe — Conversion from GAVL Tree Node to User Defined Item

Synopsis

```
void * gavl_node2item_safe (root,
                            node);
const gavl_root_t *  root;
const gavl_node_t *  node;
```

Arguments

root
: GAVL tree root

node
: node belonging to *root* GAVL tree

Return Value

pointer to item corresponding to node

Name

gavl_node2key — Conversion from GAVL Tree Node to Ordering Key

Synopsis

```
void * gavl_node2key (root,
                      node);
const gavl_root_t *  root;
const gavl_node_t *  node;
```

Arguments

root
: GAVL tree root

node
: node belonging to *root* GAVL tree

Return Value

pointer to key corresponding to node

Name

gavl_next_node — Returns Next Node of GAVL Tree

Synopsis

```
gavl_node_t * gavl_next_node (node);
const gavl_node_t * node;
```

Arguments

node
: node for which accessor is looked for

Return Value

pointer to next node of tree according to ordering

Name

gavl_prev_node — Returns Previous Node of GAVL Tree

Synopsis

```
gavl_node_t * gavl_prev_node (node);
const gavl_node_t * node;
```

Arguments

node
: node for which predecessor is looked for

Return Value

pointer to previous node of tree according to ordering

Name

gavl_balance_one — Balance One Node to Enhance Balance Factor

Synopsis

```
int gavl_balance_one (subtree);
gavl_node_t ** subtree;
```

Arguments

subtree
: pointer to pointer to node for which balance is enhanced

Return Value

returns nonzero value if height of subtree is lowered by one

Name

gavl_insert_primitive_at — Low Lewel Routine to Insert Node into Tree

Synopsis

```
int gavl_insert_primitive_at (root_nodep,
                              node,
                              where,
                              leftright);
```

gavl_node_t **	*root_nodep*;
gavl_node_t *	*node*;
gavl_node_t *	*where*;
int	*leftright*;

Arguments

root_nodep
: pointer to pointer to GAVL tree root node

node
: pointer to inserted node

where
: pointer to found parent node

leftright
: left (>=1) or right (<=0) branch

Description

This function can be used for implementing AVL trees with custom root definition. The value of the selected `left` or `right` pointer of provided *node* has to be NULL before insert operation, i.e. node has to be end node in the selected direction.

Return Value

positive value informs about success

Name

gavl_delete_primitive — Low Lewel Deletes/Unlinks Node from GAVL Tree

Synopsis

```
int gavl_delete_primitive (root_nodep,
```

node);

gavl_node_t ** *root_nodep*;
gavl_node_t * *node*;

Arguments

root_nodep
: pointer to pointer to GAVL tree root node

node
: pointer to deleted node

Return Value

positive value informs about success.

Name

gavl_cut_first_primitive — Low Lewel Routine to Cut First Node from Tree

Synopsis

`gavl_node_t *` **`gavl_cut_first_primitive`** `(`*root_nodep*`);`

gavl_node_t ** *root_nodep*;

Arguments

root_nodep
: pointer to pointer to GAVL tree root node

Description

This enables fast delete of the first node without tree balancing. The resulting tree is degraded but height differences are kept consistent. Use of this function can result in height of tree maximally one greater the tree managed by optimal AVL functions.

Return Value

returns the first node or NULL if the tree is empty

Name

gsa_struct_init — Initialize GSA Structure

Synopsis

```
void gsa_struct_init (array,
                      key_offs,
                      cmp_fnc);
gsa_array_t *     array;
int               key_offs;
gsa_cmp_fnc_t *   cmp_fnc;
```

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

key_offs
: offset to the order controlling field obtained by `UL_OFFSETOF`

cmp_fnc
: function defining order of items by comparing fields at offset *key_offs*.

Name

gsa_delete_all — Delete Pointers to the All Items in the Array

Synopsis

```
void gsa_delete_all (array);
gsa_array_t * array;
```

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

Description

This function releases all internally allocated memory, but does not release memory of the *array* structure

---

Name

gsa_bsearch_indx — Search for Item or Place for Item by Key

Synopsis

```
int gsa_bsearch_indx (array,
                      key,
                      key_offs,
                      cmp_fnc,
                      mode,
                      indx);
```

gsa_array_t * *array*;
void * *key*;
int *key_offs*;
gsa_cmp_fnc_t * *cmp_fnc*;
int *mode*;
unsigned * *indx*;

Arguments

`array`
: pointer to the array structure declared through `GSA_ARRAY_FOR`

`key`
: key value searched for

`key_offs`
: offset to the order controlling field obtained by `UL_OFFSETOF`

`cmp_fnc`
: function defining order of items by comparing fields

`mode`
: mode of the search operation

`indx`
: pointer to place, where store value of found item array index or index where new item should be inserted

Description

Core search routine for GSA arrays binary searches for item with field at offset `key_off` equal to `key` value Values are compared by function pointed by *`cmp_fnc` field in the array structure `array`. Integer `mode` modifies search algorithm: `GSA_FANY` .. finds item with field value *`key`, `GSA_FFIRST` .. finds the first item with field value *`key`, `GSA_FAFTER` .. index points after last item with *`key` value, reworded - index points at first item with higher value of field or after last item

Return Value

Return of nonzero value indicates match found.

Name

gsa_find — Find Item for Provided Key

Synopsis

```
void * gsa_find (array,
                 key);
```

gsa_array_t * *array*;
void * *key*;

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

key
: key value searched for

Return Value

pointer to item associated to key value or `NULL`.

Name

gsa_find_first — Find the First Item for Provided Key

Synopsis

`void * `**`gsa_find_first`**` (`*array*,
key`);`

gsa_array_t * *array*;
void * *key*;

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

key
: key value searched for

Description

same as above, but first matching item is found.

Return Value

pointer to the first item associated to key value or `NULL`.

Name

gsa_find_indx — Find the First Item with Key Value and Return Its Index

Synopsis

`void * `**`gsa_find_indx`**` (`*array*,
key,
indx`);`

gsa_array_t * *array*;
void * *key*;
int * *indx*;

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

key
: key value searched for

indx
: pointer to place for index, at which new item should be inserted

Description

same as above, but additionally stores item index value.

Return Value

pointer to the first item associated to key value or `NULL`.

Name

gsa_insert_at — Insert Existing Item to the Specified Array Index

Synopsis

```
int gsa_insert_at (array,
                   item,
                   where);
gsa_array_t *  array;
void *         item;
unsigned       where;
```

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

item
: pointer to inserted Item

where
: at which index should be *item* inserted

Return Value

positive or zero value informs about success

Name

gsa_insert — Insert Existing into Ordered Item Array

Synopsis

```
int gsa_insert (array,
                item,
                mode);
gsa_array_t * array;
void *        item;
int           mode;
```

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

item
: pointer to inserted Item

mode
: if mode is `GSA_FAFTER`, multiple items with same key can be stored into array, else strict ordering is required

Return Value

positive or zero value informs about success

Name

gsa_delete_at — Delete Item from the Specified Array Index

Synopsis

```
int gsa_delete_at (array,
                   indx);
gsa_array_t * array;
unsigned      indx;
```

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

indx
: index of deleted item

Return Value

positive or zero value informs about success

Name

gsa_delete — Delete Item from the Array

Synopsis

```
int gsa_delete (array,
                item);
gsa_array_t *  array;
void *         item;
```

Arguments

array
: pointer to the array structure declared through GSA_ARRAY_FOR

item
: pointer to deleted Item

Return Value

positive or zero value informs about success

Name

gsa_resort_buble — Sort Again Array If Sorting Criteria Are Changed

Synopsis

```
int gsa_resort_buble (array,
                      key_offs,
                      cmp_fnc);
gsa_array_t *    array;
int              key_offs;
gsa_cmp_fnc_t *  cmp_fnc;
```

Arguments

array
: pointer to the array structure declared through GSA_ARRAY_FOR

key_offs
: offset to the order controlling field obtained by UL_OFFSETOF

cmp_fnc
: function defining order of items by comparing fields

Return Value

non-zero value informs, that resorting changed order

Name

gsa_setsort — Change Array Sorting Criterion

Synopsis

```
int gsa_setsort (array,
                 key_offs,
                 cmp_fnc);
```

gsa_array_t * *array*;
int *key_offs*;
gsa_cmp_fnc_t * *cmp_fnc*;

Arguments

array
: pointer to the array structure declared through `GSA_ARRAY_FOR`

key_offs
: new value of offset to the order controlling field

cmp_fnc
: new function defining order of items by comparing fields at offset *key_offs*

Return Value

non-zero value informs, that resorting changed order

Name

struct gsa_array_field_t — Structure Representing Anchor of ustom GSA Array

Synopsis

```
struct gsa_array_field_t {
  void ** items;
  unsigned count;
  unsigned alloc_count;
};
```

Members

items
: pointer to array of pointers to individual items

count
: number of items in the sorted array

alloc_count
: allocated pointer array capacity

Name

struct ul_htim_node — Timer queue entry base structure

Synopsis

```
struct ul_htim_node {
  #elseul_hpt_node_t node;
  #elseul_hpt_node_t node;
  #endiful_htim_time_t expires;
};
```

Members

node
: regular GAVL node structure for insertion into

node
: regular GAVL node structure for insertion into

expires
: time to trigger timer in &ul_htim_time_t type defined resolution

Description

This is basic type useful to define more complete timer types

---

Name

struct ul_htim_queue — Timer queue head/root base structure

Synopsis

```
struct ul_htim_queue {
  #elseul_hpt_root_field_t timers;
  #elseul_hpt_root_field_t timers;
  #endifint first_changed;
};
```

Members

timers
: root of FLES GAVL tree of timer entries

timers
: root of FLES GAVL tree of timer entries

first_changed
: flag, which is set after each add, detach operation which concerning of firsts scheduled timer

Description

This is basic type useful to define more complete timer queues types

---

Name

struct ul_htimer — Standard timer entry with callback function

Synopsis

```
struct ul_htimer {
  ul_htim_node_t htim;
  ul_htimer_fnc_t * function;
  unsigned long data;
};
```

Members

htim
: basic timer queue entry

function
: user provided function to call at trigger time

data
: user selected data

Description

This is standard timer type, which requires *`data`* casting in many cases. The type of *`function`* field has to be declared in “ul_htimdefs.h” header file.

Name

struct ul_htimer_queue — Standard timer queue

Synopsis

```
struct ul_htimer_queue {
  ul_htim_queue_t htim_queue;
};
```

Members

htim_queue
: the structure wraps &ul_htim_queue structure

Description

This is standard timer type, which requires *`data`* casting in many cases

---

Name

list_add — add a new entry

Synopsis

```
void list_add (new,
               head);
```

struct list_head * *new*;

struct list_head * *head*;

Arguments

new
: new entry to be added

head
: list head to add it after

Description

Insert a new entry after the specified head. This is good for implementing stacks.

Name

list_add_tail — add a new entry

Synopsis

`void` **`list_add_tail`** `(`*new*, *head*`);`

struct list_head * *new*;

struct list_head * *head*;

Arguments

new
: new entry to be added

head
: list head to add it before

Description

Insert a new entry before the specified head. This is useful for implementing queues.

Name

list_del — deletes entry from list.

Synopsis

`void` **`list_del`** `(`*entry*`);`
struct list_head * `entry`;

Arguments

entry
: the element to delete from the list.

Note

list_empty on entry does not return true after this, the entry is in an undefined state.

Name

list_del_init — deletes entry from list and reinitialize it.

Synopsis

```
void list_del_init (entry);
struct list_head * entry;
```

Arguments

entry
: the element to delete from the list.

Name

list_move — delete from one list and add as another's head

Synopsis

```
void list_move (list,
                head);
struct list_head * list;
struct list_head * head;
```

Arguments

list
: the entry to move

head
: the head that will precede our entry

Name

list_move_tail — delete from one list and add as another's tail

Synopsis

```
void list_move_tail (list, head);
struct list_head * list;
struct list_head * head;
```

Arguments

list
: the entry to move

head
: the head that will follow our entry

Name

list_empty — tests whether a list is empty

Synopsis

```
int list_empty (head);
struct list_head * head;
```

Arguments

head
: the list to test.

Name

list_splice — join two lists

Synopsis

```
void list_splice (list, head);
struct list_head * list;
struct list_head * head;
```

Arguments

list
: the new list to add.

head
: the place to add it in the first list.

Name

list_splice_init — join two lists and reinitialise the emptied list.

Synopsis

```
void list_splice_init (list,
                       head);
struct list_head * list;
struct list_head * head;
```

Arguments

list
: the new list to add.

head
: the place to add it in the first list.

Description

The list at *list* is reinitialised

Name

list_entry — get the struct for this entry

Synopsis

```
list_entry (ptr, type, member);
ptr;
type;
member;
```

Arguments

ptr
: the &struct list_head pointer.

type
: the type of the struct this is embedded in.

member
: the name of the list_struct within the struct.

Name

list_for_each — iterate over a list

Synopsis

list_for_each (*pos*, *head*);
pos;
head;

Arguments

pos
: the &struct list_head to use as a loop counter.

head
: the head for your list.

Name

__list_for_each — iterate over a list

Synopsis

__list_for_each (*pos*, *head*);
pos;
head;

Arguments

pos
: the &struct list_head to use as a loop counter.

head
: the head for your list.

Description

This variant differs from `list_for_each` in that it's the simplest possible list iteration code, no prefetching is done. Use this for code that knows the list to be very short (empty or 1 entry) most of the time.

Name

list_for_each_prev — iterate over a list backwards

Synopsis

list_for_each_prev (*pos*, *head*);
pos;
head;

Arguments

pos

the &struct list_head to use as a loop counter.

head

the head for your list.

Name

list_for_each_safe — iterate over a list safe against removal of list entry

Synopsis

`list_for_each_safe` (*pos*, *n*, *head*);

`pos`;
`n`;
`head`;

Arguments

pos

the &struct list_head to use as a loop counter.

n

another &struct list_head to use as temporary storage

head

the head for your list.

Name

list_for_each_entry — iterate over list of given type

Synopsis

`list_for_each_entry` (*pos*, *head*, *member*);

`pos`;
`head`;
`member`;

Arguments

pos

the type * to use as a loop counter.

head

the head for your list.

member

the name of the list_struct within the struct.

Name

list_for_each_entry_reverse — iterate backwards over list of given type.

Synopsis

`list_for_each_entry_reverse` (*pos*,
head,
member);

pos;
head;
member;

Arguments

`pos`
: the type * to use as a loop counter.

`head`
: the head for your list.

`member`
: the name of the list_struct within the struct.

Name

list_for_each_entry_safe — iterate over list of given type safe against removal of list entry

Synopsis

`list_for_each_entry_safe` (*pos*, *n*, *head*, *member*);

`pos`;
`n`;
`head`;
`member`;

Arguments

`pos`
: the type * to use as a loop counter.

`n`
: another type * to use as temporary storage

`head`
: the head for your list.

`member`
: the name of the list_struct within the struct.

libsuiut API

Name

sui_dinfo_inc_refcnt — Increase reference count of DINFO

Synopsis

`void` **`sui_dinfo_inc_refcnt`** `(`*datai*`);`
sui_dinfo_t * *datai*;

Arguments

`datai`
Pointer to dinfo structure.

File

sui_dinfo.c

Name

sui_dinfo_dec_refcnt — Decrease reference count of DINFO

Synopsis

`void` **`sui_dinfo_dec_refcnt`** `(`*datai*`);`
sui_dinfo_t * *datai*;

Arguments

`datai`
Pointer to dinfo structure.

Description

If the reference count reaches zero, DINFO starts to be destroyed. The event `SUEV_COMMAND` with command `SUCM_DONE` is sent to dinfo, next event `SUEV_FREE` is emmited or direct `free` is called the `SUEV_FREE` is disabled.

File

sui_dinfo.c

Name

sui_create_dinfo — Creates new dynamic DINFO

Synopsis

`sui_dinfo_t *` **`sui_create_dinfo`** `(`*adata*,
afdig,
amin,
amax,
ainfo,
rd,

wr);

void *	*adata*;
int	*afdig*;
long	*amin*;
long	*amax*;
long	*ainfo*;
sui_datai_rdfnc_t *	*rd*;
sui_datai_wrfnc_t *	*wr*;

Arguments

adata
: DINFO type specific pointer to the data

afdig
: Number of fractional digits if the fixed decimal point format is used

amin
: The minimal allowed value

amax
: The maximal allowed value

ainfo
: DINFO type specific pointer

rd
: Pointer to the read processing function

wr
: Pointer to the write processing function

Return Value

Pointer to newly created DINFO.

File

sui_dinfo.c

Name

sui_create_dinfo_int — Creates DINFO for signed integer or fixed point data

Synopsis

```
sui_dinfo_t * sui_create_dinfo_int (adata,
                                    aidxsize,
                                    asize);
```

void *	*adata*;
long	*aidxsize*;
int	*asize*;

Arguments

adata
: Pointer to the signed char, short, int, long or fixed point data

aidxsize
: Allowed range of indexes form 0 to *aidxsize*-1, if zero, then no check

asize
: The size of the integer type representation returned by `sizeof`

Return Value

Pointer to newly created DINFO.

File

sui_dinfo.c

Name

sui_create_dinfo_uint — Creates DINFO for unsigned integer or fixed point data

Synopsis

```
sui_dinfo_t * sui_create_dinfo_uint (adata,
                                     aidxsize,
                                     asize);
```

void * *adata*;
long *aidxsize*;
int *asize*;

Arguments

adata
: Pointer to the unsigned char, short, int, long or fixed point data

aidxsize
: Allowed range of indexes form 0 to *aidxsize*-1, if zero, then no check

asize
: The size of the integer type representation returned by `sizeof`

Return Value

Pointer to newly created DINFO.

File

sui_dinfo.c

Name

sui_rd_long — Reads long integer data from specified DINFO

Synopsis

```
int sui_rd_long (datai,
                 idx,
                 buf);
```

sui_dinfo_t * *datai*;
long *idx*;
long * *buf*;

Arguments

datai
: Pointer to the DIONFO

idx
: Index of read data inside DINFO.

buf
: Pointer to where the read value is stored

Return Value

Operation result code, `SUDI_DATA_OK` in the case of success.

File

sui_dinfo.c

Name

sui_wr_long — Writes long integer data to specifies DINFO

Synopsis

```
int sui_wr_long (datai,
                 idx,
                 buf);
```

sui_dinfo_t * *datai*;
long *idx*;
const long * *buf*;

Arguments

datai
: Pointer to the DIONFO

idx
: Index of read data inside DINFO.

buf

Pointer to the new data value

Return Value

Operation result code, `SUDI_DATA_OK` in the case of success.

File

sui_dinfo.c

Name

dinfo_scale_proxy — Creates value scale proxy DINFO

Synopsis

```
sui_dinfo_t * dinfo_scale_proxy (dfrom,
                                 ainfo,
                                 amultiply,
                                 adivide);
```

sui_dinfo_t *	*dfrom*;
long	*ainfo*;
long	*amultiply*;
long	*adivide*;

Arguments

dfrom
: Pointer to the underlying DINFO

ainfo
: The local DINFO specific parameter

amultiply
: Multiply factor

adivide
: Divide factor

Description

Creates scaling proxy DINFO. Read value is multiplied by *`amultiply`* factor and then divided by *`adivide`* factor. The long integer overflow is not checked. If the full checking is required use `sui_lintrans_proxy` instead which works with wider numbers representations and checks for all overflow cases.

Return Value

Pointer to newly created DINFO.

File

sui_dinfo.c

Name

dinfo_simple_proxy — Creates simple proxy DINFO

Synopsis

```
sui_dinfo_t * dinfo_simple_proxy (dfrom,
                                  ainfo);
sui_dinfo_t * dfrom;
long          ainfo;
```

Arguments

dfrom
: Pointer to the underlying DINFO

ainfo
: The local DINFO specific parameter which specifies index value for calling of underlying DINFO

Return Value

Pointer to newly created DINFO.

File

sui_dinfo.c

Name

sui_dinfo_dbuff_create — Creates DINFO for ul_dbuff structure

Synopsis

```
sui_dinfo_t * sui_dinfo_dbuff_create (db,
                                      aidxsize);
ul_dbuff_t * db;
long         aidxsize;
```

Arguments

db
: Pointer to the dbuff

aidxsize
: Allowed range of indexes form 0 to *aidxsize*-1, if zero then no check

Returns

Pointer to newly created DINFO.

File

sui_dinfo_dbuff.c

Name

sui_dinfo_dbuff_rd_dbuff — Reads ul_dbuff data from specified DINFO

Synopsis

```
int sui_dinfo_dbuff_rd_dbuff (di,
                              idx,
                              dbuf);
```

sui_dinfo_t * *di*;
long *idx*;
ul_dbuff_t * *dbuf*;

Arguments

di
: Pointer to the DIONFO

idx
: Index of read data inside DINFO.

dbuf
: Pointer to where the read value is stored

Return Value

Operation result code, `SUDI_DATA_OK` in the case of success.

File

sui_dinfo_dbuff.c

Name

sui_dinfo_dbuff_wr_dbuff — Writes ul_dbuff data to specifies DINFO

Synopsis

```
int sui_dinfo_dbuff_wr_dbuff (di,
                              idx,
                              dbuf);
```

sui_dinfo_t * *di*;
long *idx*;
const ul_dbuff_t * *dbuf*;

Arguments

di
: Pointer to the DIONFO

idx
: Index of read data inside DINFO.

dbuf
: Pointer to the dbuff

Return Value

Operation result code, `SUDI_DATA_OK` in the case of success.

File

sui_dinfo_dbuff.c

Name

sui_dinfo_dbuff_rd_long — Reads long integer data from specified dbuff DINFO

Synopsis

int **sui_dinfo_dbuff_rd_long** (*di*, *idx*, *buf*);

sui_dinfo_t *	*di*;
long	*idx*;
long *	*buf*;

Arguments

di
: Pointer to the DIONFO

idx
: Index of read data inside DINFO.

buf
: Pointer to the dbuff

Return Value

Operation result code, `SUDI_DATA_OK` in the case of success.

File

sui_dinfo_dbuff.c

Name

sui_dinfo_dbuff_wr_long — Writes long integer data to specified dbuff DINFO

Synopsis

```
int sui_dinfo_dbuff_wr_long (di,
                             idx,
                             buf);
sui_dinfo_t *  di;
long           idx;
const long *   buf;
```

Arguments

di
: Pointer to the DIONFO

idx
: Index of read data inside DINFO.

buf
: Pointer to the dbuff

Return Value

Operation result code, `SUDI_DATA_OK` in the case of success.

File

sui_dinfo_dbuff.c

Name

sui_dtree_lookup — Find dinfo in the named dinfo database

Synopsis

```
int sui_dtree_lookup (from_dir,
                      path,
                      found_dir,
                      datai);
sui_dtree_dir_t *   from_dir;
const char *        path;
sui_dtree_dir_t **  found_dir;
sui_dinfo_t **      datai;
```

Arguments

from_dir
: the directory to start from

path

path from directory to dinfo or directory

found_dir
: the optional pointer to space that would hold pointer to directory of found dinfo

datai
: optional pointer to store the found dinfo

Return Value

SUI_DTREE_FOUND, SUI_DTREE_DIR, SUI_DTREE_NOPATH, SUI_DTREE_ERROR

File

sui_dtree.c

Name

sui_dtree_mem_lookup — Find dinfo in the named dinfo database

Synopsis

```
int sui_dtree_mem_lookup (from_dir,
                          path,
                          consumed,
                          found_dir,
                          datai);
```

sui_dtree_dir_t *	*from_dir*;
const char *	*path*;
int *	*consumed*;
sui_dtree_dir_t **	*found_dir*;
sui_dinfo_t **	*datai*;

Arguments

from_dir
: the directory to start from

path
: path from directory to dinfo or directory

consumed
: pointer to location for numeber of consumed characters from path

found_dir
: the optional pointer to space that would hold pointer to directory of found dinfo

datai
: optional pointer to store the found dinfo

Return Value

SUI_DTREE_FOUND, SUI_DTREE_DIR, SUI_DTREE_NOPATH, SUI_DTREE_ERROR

File

sui_dtreemem.c

Name

struct sui_dtree_memdir_t — Ancestor of sui_dtree_dir_t which containing sui_dtree_memnode_t GAVL list .

Synopsis

```
struct sui_dtree_memdir_t {
  sui_dtree_dir_t dir;
  gavl_cust_root_field_t name_root;
};
```

Members

dir
: base struct (Container_of technology). Containing dir needs it.

name_root
: GAVL with children of type &sui_dtree_memnode_t

Header

sui_dtreemem.h

Name

struct sui_dtree_memnode_t — structure representing single node in memtree.

Synopsis

```
struct sui_dtree_memnode_t {
  char * name;
  int node_type;
  gavl_node_t name_node;
  union ptr;
  void * dll_handle;
};
```

Members

name
: structure neccessary for storing node in GAVL tree, is NULL for subindicies

node_type

type of node contens (dir or dinfo)

name_node
: the structure can be stored in GAVL tree thanks to that field

ptr
: pointer to dinfo or directory that this node contains.

dll_handle
: if memnode is one imported from DLL, DLLs handle is stored here. (else it is 0)

Description

Node can contain `dinfo` or `directory` (&sui_dtree_dir_t).

Header

sui_dtreemem.h.h

Name

struct sui_event — Common suitk event structure

Synopsis

```
struct sui_event {
  unsigned short what;
};
```

Members

what
: Code of event.(See 'event_code' enum with 'SUEV_' prefix)

File

sui_base.h

Name

enum event_code — Code of SUITK events ['SUEV_' prefix]

Synopsis

```
enum event_code {
  SUEV_MDOWN,
  SUEV_MUP,
  SUEV_MMOVE,
  SUEV_MAUTO,
  SUEV_KDOWN,
  SUEV_KUP,
  SUEV_DRAW,
  SUEV_REDRAW,
```

```
  SUEV_COMMAND,
  SUEV_BROADCAST,
  SUEV_SIGNAL,
  SUEV_GLOBAL,
  SUEV_FREE,
  SUEV_NOTHING,
  SUEV_MOUSE,
  SUEV_KEYBOARD,
  SUEV_MESSAGE,
  SUEV_DEFMASK,
  SUEV_GRPMASK
};
```

Constants

SUEV_MDOWN
: Mouse button is down.

SUEV_MUP
: Mouse button is up.

SUEV_MMOVE
: Mouse is in move.

SUEV_MAUTO

SUEV_KDOWN
: Key is down.

SUEV_KUP
: Key is up.

SUEV_DRAW
: Draw widget.

SUEV_REDRAW
: Redraw widget.

SUEV_COMMAND
: Command event.

SUEV_BROADCAST
: Bradcast event.

SUEV_SIGNAL
: ?

SUEV_GLOBAL
: ?

SUEV_FREE
: ?

SUEV_NOTHING
: ?

SUEV_MOUSE
: ?

SUEV_KEYBOARD
: ?

SUEV_MESSAGE
: ?

SUEV_DEFMASK
: ?

SUEV_GRPMASK
?

File

sui_base.h

Name

enum command_event — Command codes for command event ['SUCM_' prefix]

Synopsis

```
enum command_event {
  SUCM_VALID,
  SUCM_QUIT,
  SUCM_ERROR,
  SUCM_MENU,
  SUCM_CLOSE,
  SUCM_ZOOM,
  SUCM_RESIZE,
  SUCM_NEXT,
  SUCM_PREV,
  SUCM_HELP,
  SUCM_OK,
  SUCM_CANCEL,
  SUCM_YES,
  SUCM_NO,
  SUCM_DEFAULT,
  SUCM_FOCUSASK,
  SUCM_FOCUSSET,
  SUCM_FOCUSREL,
  SUCM_INIT,
  SUCM_DONE,
  SUCM_NEWDISPLAY,
  SUCM_DISPNUMB,
  SUCM_CHANGE_STBAR,
  SUCM_NEXT_GROUP,
  SUCM_PREV_GROUP,
  SUCM_EVC_LINK_TO
};
```

Constants

SUCM_VALID
VALID command event.
SUCM_QUIT
QUIT command event.
SUCM_ERROR
ERROR command event.
SUCM_MENU
MENU command event. Open, select, close, ... menu.

SUCM_CLOSE
    CLOSE command event.
SUCM_ZOOM
    ZOOM command event.
SUCM_RESIZE
    RESIZE command event.
SUCM_NEXT
    NEXT command event. Mainly for change widget focus by pressing TAB key.
SUCM_PREV
    PREV command event. Mainly for change widget focus by pressing SHIFT+TAB key.
SUCM_HELP
    HELP command event.
SUCM_OK
    OK button pressed.
SUCM_CANCEL
    CANCEL button pressed.
SUCM_YES
    YES button pressed.
SUCM_NO
    NO button pressed.
SUCM_DEFAULT
    DEFAULT button pressed.
SUCM_FOCUSASK
    Which widget has focus ?
SUCM_FOCUSSET
    Set focus to the widget.
SUCM_FOCUSREL
    Release focus from the widget.
SUCM_INIT
    Initialize widget.
SUCM_DONE
    Done widget - decrement reference counter, deallocate widget data.
SUCM_NEWDISPLAY
    Create new screen from pointer to screen.
SUCM_DISPNUMB
    Create new screen from number to screen.
SUCM_CHANGE_STBAR
    Status bar is changed.
SUCM_NEXT_GROUP
    Change focus between groups (like as ALT+TAB in Windows).
SUCM_PREV_GROUP
    Change focus between groups.
SUCM_EVC_LINK_TO
    Only Pavel Pisa knows :))

File

sui_base.h