WP10 -D10.7 User 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 by Pierre Morel
6) chapter missing: performances by all
7) chapter still missing: Adding a new component: Kconfig and makefiles explained by Pierre Morel
8) Re-reading and corrections by Ocera members

Table of Contents

Illustration Index

PART I

The OCERA Framework

1) Overview

By Pierre Morel - MNIS

This is the OCERA User Guide. We will try through this document to help you understanding how to use OCERA.
In this first chapter we will explain the goal of this guide and introduce OCERA to you, so that you will be familiar with the concepts used all along the guide.

1.1) Presentation

1.1.1) Short description of OCERA

OCERA is a European project, started in April 2002, with the goal to provide the European industry with an Open Source Real-Time system following Industry standards. OCERA stands for **Open Components for Embedded Realtime Applications** and we defined as component:
"pieces of software bringing functionalities for Scheduling, Quality Of Service, Fault Tolerance or Communication."

The components are basically developed for Linux and RTLinux-GPL but may be portable to other Realtime systems or other time sharing systems to bring to them some realtime feature.
The component are all Open Source and most of the components licences are LGPL[1] or GPL.

1.1.2) Intended *Audience*

This guide is intended for
software engineers who plan to build a real time embedded application for **commercial** or **educational** use.
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 exemples.

1.1.3) 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, a basic knowledge of TCP/IP network architectures.
To develop applications using OCERA you will need some skills in a development language, either C, C++ or Ada 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.

1 Actually, the licence of RTLinux-GPL as the licence of Linux are GPL but most of the Linux Libraries licences are LGPL (Less GPL), which means that they give more freedom in the usage. If you intend to make a commercial usage of your development using OCERA, you will have a great interest in reading the chapter on Licensing at the end of the guide.

1.2) What will you find in the User's Guide

The User's guide will present you

- an overview of the different components of OCERA.
- interfaces between the realtime system and the time sharing system.
- the development tools, compiler and debuggers.
- how to build an embedded system with OCERA.

If you need details on the programing interface for hard and soft real-time applications you will have to refer to the *OCERA PROGRAMMER'S GUIDE*.
The Guide will provide you the informations you need to develop a Real-Time application and to integrate it on an embedded system using OCERA.
After the present chapter introducing OCERA, you will find the following chapters in this guide:

- **Architecture:** more details on OCERA project and the architecture of the OCERA system.
- **Getting Started:** How to get the sources and the documentation.
- **Defining the framework**: How to generate a complete embedded system.
- **Development:** How to develop a dedicated application and how to add this application to the OCERA system.
- **Integration:**: How to generate an image, install this image on an embedded system or on a training workstation and how to boot on this embedded system or on the workstation?
- **Debugging:** How to debug and test the complete system.

At the end of the Guide you will also find examples of sample applications:

- control command with CAN provided by UniControl
- real time Ethernet provided by the University Prague
- a robotic application provided by the CEA
- streaming video provided by Visual Tools

2) Architecture of the OCERA System

By Pierre Morel - MNIS

2.1) History of the OCERA project

OCERA is a European project, started in April 2002, with the goal to provide the European industry with an Open Source Real-Time system following Industry standards. Therefor, OCERA, using existing technology like Linux and the RTLinux patch, started to redesign the Real Time part of Linux and RTLinux to offer:

- POSIX compliant interface.
- Communication with an industrial bus: CAN
- Quality Of Service, allowing hard and soft real time to co exist on the same system.
- Real Time Exchange over Ethernet
- and Fault Tolerance.

You will need OCERA if you need one or more items of the following list:

- A good hard real-time scheduling latency and response time for embedded systems.

- A soft real-time environment with a huge choice of applications.
- Quality of service with Bandwidth reservation
- Fault tolerance and reconfiguration
- A real time Ethernet communication layer
- A POSIX compliant programming interface
- Avionic certified real-time system
- An OpenSource GPL licensed real-time system
- A single generation interface for the complete embedded system, kernel, drivers, tools, libraries and application

To achieve these goals, the OCERA system uses the LINUX/RTLinux combination, which already exists and provide a good startpoint in term of latency and in term of standard POSIX interfaces and enhanced it the following way:

- rewriting of some of the RTLinux primitives to correct it or to make them POSIX compliant.
- adding new POSIX components to RTLinux, barriers, timers
- adding new realtime scheduling possibilities with EDF and CBS schedulers and an application defined scheduler.
- enhancing the Linux scheduler to add CBS scheduling
- adding POSIX traces for the real time components
- adding Quality Of Service primitives to both the hard real-time (RTLinux) and the soft-real-time (LINUX with real time extensions)
- integrating Components, RTLinux and Linux configuration and generation in a single graphical tool.

2.2) Technical choices

2.2.1) The goals

The goal we follow is to have a maximum realtime efficiancy while allowing the greater possible number of standard applications without realtime expectations to run on our system.
We had at the beginning the idea of using a aystem architecture based on a separation of the functionalities, the realtime functionalities being provided by a RealTime Operating System and the non realtime functionalities provided by a standard, non realtime, operating system.
The study of existing real-time operating systems lead us to define what would be of interest to have in a optimal real-time operating system. See deliverable D1.1 of the ocera project at the OCERA main web site:
http://www.ocera.org/archives/deliverables/WP1/D1.1.pdf
Since our project is based on a cooperation between Universities and industrial companies and has a duration of 2 and half year, we needed to start with a system already used in the industry and enhance it.

2.2.2) The choices

We looked at what existed and studied two open source real-time system that could be appropriate:
RTLinux-GPL and the derivative, RTAI.
We choosed RTLinux-GPL which design is closer to our goals. You can read the details in the document RTLINUX versus RTAI that you must be able to download from OCERA web site.
We choose the linux kernel 2.4.18 as a start for the developpement, as it was the most stable kernel as we began the project.

2.2.3) The drawbacks

a) Rapidity versus Security

The choices we made to achieve our goals lead to some drawbacks. The first one is that we choose rapidity with oposition to security.
In OCERA, all realtime threads and all drivers share the same address space as the Linux kernel. Only linux user's processes are protected against bad memory accesses.
This can lead to serious security problem which can be eliminated by a serious and structured software development. The usage of UML to design and of Ada to developp applications is a way to reduce it by providing a secured development framework.
For drivers and internal realtime kernel and Linux kernel development, the security is the entire responsibility of the developper.
Other realtime operating systems provide driver's security or/and realtime task security like L4Linux, Jaluna or RTLinux-Pro.

b) GPL and commercial usage

The kernel of the developments are RTLINUX-GPL and LINUX. Both are using the GNU General Public Licence, the GNU-GPL also simply called GPL.
Following the General Public Licence has the following implications:
All development made by the OCERA consortium within Linux or RTLinux must be GPL.
All code developped and linked to these sofware must be GPL. The RTLinux-GPL applications threads are linked with RTLinux-GPL libraries and must be GPL.
On the other hand ORTE, the Fault tolerance system and the Quality Of Service, as used in the Linux User's environment may use the licence scheme they want and use LGPL , Less GPL Licence, allowing an application linked to these libraries to use the licence type the developper want to use. Thus allowing commercial applications to use the interfaces.

2.3) Quick overview of the architecture

2.3.1) 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 iddle 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.

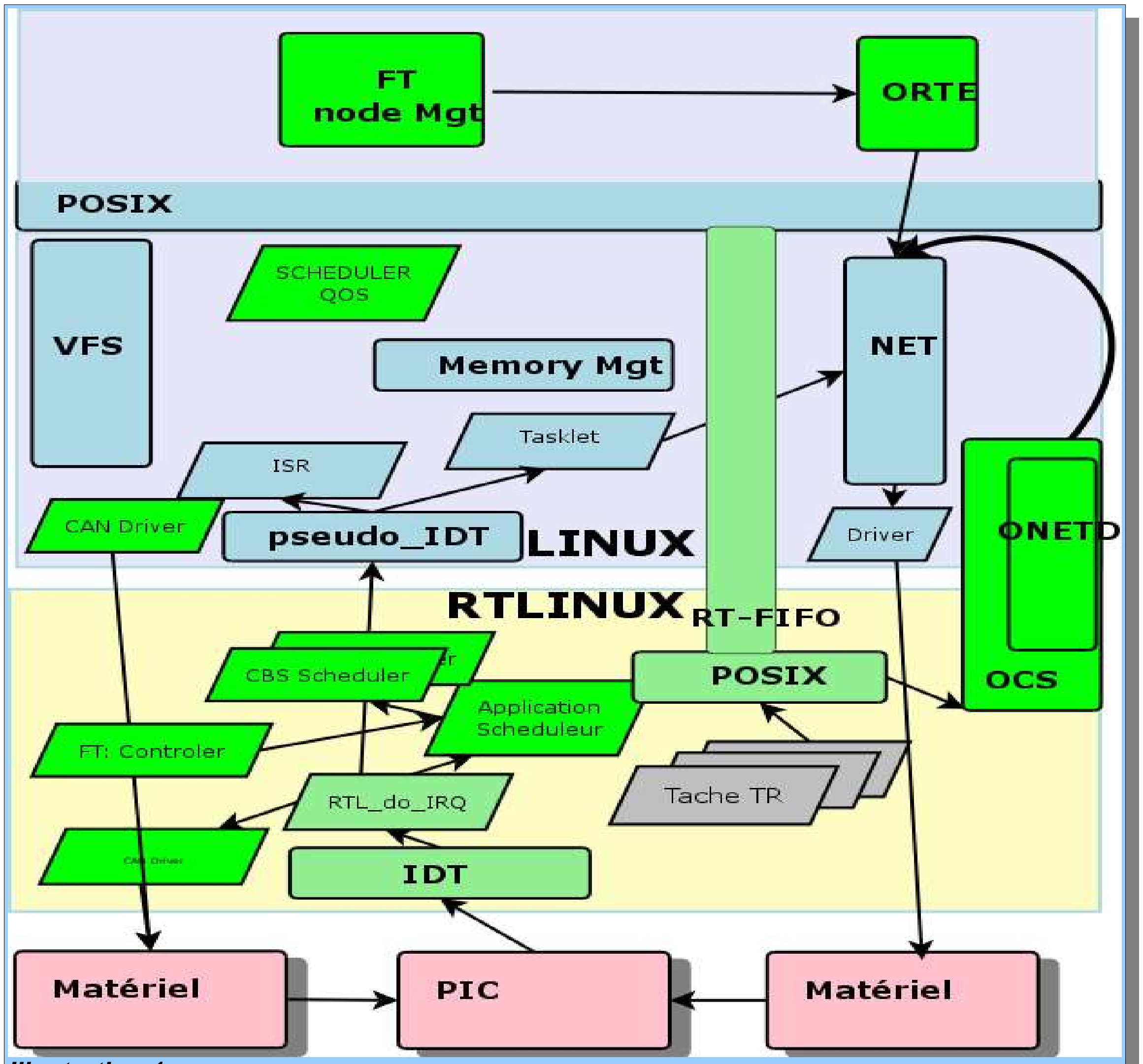


Illustration 1
Drawing 1OCERA Frameworf

2.3.2) 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:

a) 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.

b) 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

c) 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.

d) 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.

2.4) Future developments

The OCERA consortium released OCERA 1.0 in mai 2004 and OCERA-1.1 in february 2005 based on Linux 2.4.18 and has a nomber of sub-projects being developped but not actually included in the stable release.

- SA-RTLinux: a stand alone RTLinux, providing a little and fast realtime kernel.
- RTLide: a realtime aware IDE driver.
- RTLfs: a realtime file system to access the disk from within RTLinux
- RTLwIP: a realtime UDP stack providing complete TCP/IP stack within RTLinux-GPL
- JAVA: port of a JAVA machine inside RTLinux-GPL
- C++: port of a C++ runtime in RTLinux-GPL

Other projects are under design like:

- RTST: RealTime Secured Threads, giving to special RTLinux-GPL applications a memory protection using unused LDT in Linux and the application scheduler from RTLinux-GPL
- RTSD: RealTime Secured Drivers, allowing to write a Driver for RTLinux with memory protection.

These two project will have an important side effect, as the application or driver will use a segmented environment to run inside, they will use an interrupt mechanism to access to RTLinux-GPL resources.
The interrupt mechanism being considered to break the GPL license, we will be able to provide a way to develop commercial applications with OCERA.
The drawback of this implementation will be a lost in speed when communicating with other tasks and by task switch.

2.4.1) The documentation on the CD-ROM

Another advantage of the OCERA CD-ROM is that you get all the documentation on the CD-ROM in pdf format.

2.5) 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 these in the chapter on “uploading the target”.

PART II

OCERA components

3) Posix Components

By
Patricia Balbastre

3.1) Posix Signals

Signals are an important part of multitasking in the UNIX/POSIX environment. Signals are used for many purposes, including: exception handling, task notification of asynchronous event occurrence (timer expiration,...), emulation of multitasking or 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.

POSIX define two types of signals: basic signals and real-time signals extensions. This component provides only the basic signal mechanisms. The real-time POSIX signals extension is far more complex (they are closer to message queues than to interrupts) and require complex data structures.

POSIX signals were designed to be used in weight-processes systems, where each process has its own signal handlers, signal mask and status. But in RTLinux, as well as most embedded RTOS, the programming model is based on lightweight processes (threads). The standard is not as clear and unambiguous as it should be. We had to extend the semantics of the signals API to thread.

RTLinux-3.1 code already had partial signal support, some systems signals are supported but user signals and the facility to define signal handlers are not supported. We completed the signal support to be fully UNIX compatible.

Configuration options: The selection of this component sets RTL_OC_PSIGNALS to true and allows you to send signals (RTL_SIGUSR1, RTL_SIGUSR2 and from signal RTL_SIGRTMIN to RTL_SIGRTMAX) to threads (pthread_kill), blocking signals (pthread_sigmask), suspend a thread waiting for a signal to arrive (sigsuspend) and install signal handlers, among other things.

3.2) Posix Timers

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. Although RTLinux has good and accurate timing facilities, it do not provides general timer functionality. RTLinux defines only one timer for each thread, which is used to implement the periodic behaviour of the thread. This component implements the POSIX real-time extensions.

Configuration options: This component depends on Posix Signals. If Posix Timers is selected, then RTL_OC_PTIMERS variable is set to true and the funciotns defined in kernel/rtlinux/include/rtl_timer.h are available.

3.3) Posix Barriers

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. Threads using a barrier must wait at a specific point until all have finished before any of them can continue.

POSIX barriers are a relatively new feature and are not supported on all systems.

Configuration options: Select this option to have defined RTL_OC_PBARRIERS variable. Then, barriers API will be available in file kernel/rtlinux/include/rtl_barrier.h

3.4) Posix Trace

This component adds (most of) the tracing support defined in the POSIX Trace standard to RTLinux. The POSIX Trace standard defines a set of portable interfaces for tracing applications.

This standard was recently approved (September 2000) and now can be found integrated with the rest of POSIX 1003.1 standards in a single document issued in 2001 after the approval of both the IEEE-SA Standards Board and the Open Group.

Configuration options: To enable this option (RTL_OC_PTRACE) you need "Shared Memory Driver" (Memory Management->Shared Memory Support)

3.5) Posix Message Queues

This component implement The POSIX message queues facility between RTLinux threads. POSIX Message Queues is a message passing facility that relies only on services that are already available or that are going to be incorporated by other components to the RTLinux core. As they do not require any modification of the RTLinux, they can be located at the High-Level RTLinux layer.
Configuration options: RTL_OC_PMQUEUE is set when you enable this option. Then, the API implemented in directory /kernel/rtlinux/pmqueue is available.

4) Quality Of Service

Giuseppe Lipari - SSSA
Luigi Palopoli - SSSA
Luca Marzario - SSSA

4.1) Resource Reservation Scheduling module

4.1.1) Summary

- Name: **qres**
- Description: It is a dynamically loadable module for the Linux kernel, that provides a resource reservation scheduler for soft real-time tasks in user space.
- Authors: Luca Abeni (luca@sssup.it), Luca Marzario (lukesky@gandalf.sssup.it), Claudio Scordino (scordino@gandalf.sssup.it)
- Reviewer
- Layer High Level Linux Component
- Version 1.0

- Status Beta
- Dependencies Depends on the Generic Scheduler Patch (gen_sched: see Chapter). It may use the High Resolution Timer Patch. It may use the Preemption Compatibility Patch (pcomp: see Chapter)
- Release date: Aprile 2003 (MS2)

4.1.2) Description

This component implements a resource reservation scheduler. The algorithm is based on the Constant Bandwidth Server (CBS) algorithm [Abe98]. However, we modified and extended the original algorithm to take into account several practical issues. In our implementation, a server can handle more than one task; an automatic reclamation mechanism [Lip00] can be optionally activated; bandwidths can be tightly bounded through a self suspension mechanism. All these features are currently available on a single software module and they can be enabled/disabled through conditional compilation (see Section for more details on how to compile and install the component). In the next version (second phase of this workpackage), we will provide different custom modules available separately.

The complete algorithm is described in the following.

Common definitions and assumptions

A CBS S_i is described by: *the server budget* Q_i and the *server period* P_i . The *server bandwidth* $U_i = Q_i / P_i$ is the fraction of the CPU bandwidth reserved to S_i . To avoid inconsistencies and overload situations, the following condition must hold at all times:

$$\sum_{i}^{n} U_i \leq 1$$

Therefore, a variation in the parameters (like for example the budget) is allowed only at replenishment time, and only if the above condition is respected.

Dynamically, each server updates two variables (q_i, d_i) . Variable q_i is the *current budget* and keeps track of the consumed bandwidth. Variable d_i is the server's *scheduling deadline*. A server is *active* if any of its tasks has a pending instance and the current budget is greater than 0. If there a is pending instance and the current budget is 0, the server is *suspended* until the current budget is recharged. If at time *t* there is no pending task's instance and the current budget is greater than 0, the server is still *active* if $t<d_i-q_i P_i/Q_i$, otherwise it is *inactive.*

The system consists of *n* servers and a global scheduler based on the Earliest Deadline First (EDF) priority assignment. At each instant, the active server with the earliest scheduling deadline is selected and the corresponding task is dispatched to execute.

A task corresponds to a Linux thread. A task instance (or job) is activated (or *arrives*) just after it has been created with a fork() or with a pthread_create(), and when it is *unblocked*. A task instance finishes when the thread is *blocked*. In this version of the scheduler we do not distinguish between different blocking situations; every time a thread is blocked, the scheduler interprets it as the finishing of the job.

Basic CBS Algorithm

The algorithm's rules are the following.

1. Initially, $q_i=0$, $d_i=0$ and the server is *inactive*.
2. When a task is activated at time *t*, if the server is *inactive*, then $q_i=Q_i$ and $d_i=t+P_i$, and the server becomes *active*. If the server is already active, then q_i and d_i remain unchanged.
3. At any time *t*, the global scheduling algorithm selects the active server with the earliest deadline d_i . When the server is selected, it executes the first task in its ready queue (which may be order according to any policy).
4. If some of its tasks is executing for *x* units of time, the server's current budget q_i is decremented by the same amount *x*.
5. The global scheduler can *preempt* the server for executing another server: in this case, the current budget q_i is no longer decremented.
6. If $q_i=0$ and some of the server tasks has not yet finished, then the server is *suspended* until time d_i ; at time d_i , q_i is recharged to Q_i , d_i is set to $d_i=d_i+P_i$ and the server can execute again.

7. When, at time *t*, the last task has finished executing and there is no other pending task in the server, the server yields to another server. Moreover, if $t \leqslant d_i - q_i P_i / Q_i$, the server becomes *inactive*; otherwise it remains *active*, and it will become *inactive* at time $d_i - q_i Q_i / P_i$, unless another task is activated before.

Reclamation

An optional reclamation mechanism has been implemented. We choose Algorithm GRUB [Lip00], which does a greedy reclamation of the unused bandwidth, i.e. it gives all free bandwidth to the currently executing task. For this reason GRUB is not a fair algorithm. However, it can be implemented with little additional overhead, and it can also be used as a voltage scheduling algorithm, allowing energy saving.

The GRUB algorithm keeps track of a global variable $U_{act}(t)$ called *Total Bandwidth,* which is the sum of the bandwidth of all active servers at time *t*. Therefore, when a server becomes active (rule 2) its bandwidth is added to $U_{act}(t)$; when a server becomes inactive (rule 7), its bandwidth is decremented from $U_{act}(t)$. Finally, rule 4 is modified to take into account the reclamation:

4'. If If some of its tasks is executing for *x* units of time, the server's current budget q_i is decremented by $x U_{act}$.

A proof of the correctness of the GRUB algorithm can be found in [Lip00].

Energy saving (voltage scheduling)

The same algorithm can be used as a voltage scheduling algorithm. Suppose that the processor supports two different voltages, which results in two different processor speeds, the nominal speed s_{nom} and the low speed s_{low}*, and* suppose that the processor normally works at speed s_{nom}. We define a minimum bandwidth and two thresholds:

$$U_{low} = \frac{s_{low}}{s_{nom}} \qquad U_{th1} < U_{th2} < U_{low}$$

Then, we add two additional rules:

8. if U_{act} is below the first threshold U_{th1}, the speed of the processor is set to s_{low};

9. if U_{act} is greater than the second threshold U_{th2} , the speed of the processor is set to s_{nom} .

These additional rules are currently under implementation and test on a version of Linux that runs on a Intel PXA250 processor. Therefore, this feature it is only available as an alpha release.

4.1.3) API / Compatibility

The provided module does not introduce any new primitive. However, it modifies an existing primitive, the **sched_setscheduler**() standard Linux primitive, by allowing a new kind of scheduler to be selected by the user.

The standard interface of the **sched_setscheduler** primitive is the following:

```
int sched_setscheduler(pit_t pid, int policy,
                       const struct sched_param *p);
```

where pid is the id of the process for which the new scheduling strategy can be set; policy can be one of **SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER**; **p** is a pointer to a structure **sched_parms** that contains the scheduling parameters. The CBS module provides a new scheduling policy, called **SCHED_CBS**.

Thanks to the use the **gensched** patch (see Chapter), **sched_parm** has an additional field **sched_p**, of type **(void *)**, which can point to a dedicated structure that holds our scheduling parameters for the CBS. The size of this structure must be put into another field of **sched_parms**, called **sched_size**.

The CBS dedicated structure is listed below:

```
struct cbs_param {
  unsigned long int signature;
  int period;
  int max_budget;
};
```

To set the CBS scheduler for a certain process, it is necessary to write the following lines of code:

```
struct sched_param sp;
struct cbs_param cs;
```

```
    sp.sched_size = sizeof(struct cbs_param);
    sp.sched_p = &cs;
    cs.signature = CBS_SIGNATURE;
    cs.max_budget = Q;
    cs.period = T;

    res = sched_setscheduler(getpid(), SCHED_CBS, &sp);
```

Notice that only the superuser can call the **sched_setscheduler()** primitive.

PROC interface

Also, an interface through the standard **/proc** file system is available. Currently, the interface is only meant for debugging purposes. It is possible to analyse the status of the scheduler by reading all important scheduling parameters for each task. The CBS module can optionally provide the following file in the standard Linux virtual file system:

```
    /proc/cbs/scheduler
```

which can be read as a normal text file and contains one line for each task served by a CBS, in which all sensitive informations are listed. We are evaluating the overhead of using such interface to see if it would be possible to send basic commands to the scheduler through such interface.

4.1.4) Implementation issues

The **qres** module uses the generic scheduler patch to provide its services. In particular, when the module is loaded (see function **init_module** in **src/init.c**), the hook function pointers defined in the generic scheduler patch are assigned particular functions provided by the **qres** module. All the rules presented in Section are implemented in such functions.

The CBS implements the EDF ready queue internally. Every task descriptor has a pointer to a structure (called **cbs_struct**, in file src/include/cbs.h) where the CBS puts its own task's related data. Among these data is the current budget, the period, the absolute deadline, and so on.

If the **MULTI** option is defined (see Section), each CBS can handle more than one task. How these tasks are scheduled by Linux is up to the standard Linux scheduler.

4.1.5) Tests

The **qres** module is currently distributed together with a test program that can be used to verify the correct behavior of the module trough a graphical trace of the scheduling process.

The test program is made up of two executable file: *Schedtest* and *Filter*.

The *Schedtest* program run a time consuming CBS task with a specified budget and period. The syntax is:

```
schedtest Q T seconds > tracefile
```

where *Q* is the budget of the server (in micro-sec), *T* is the period of the server (in micro-sec), *seconds* is the execution time of the test and *tracefile* is the name of the file for saving the trace. You can run more than one *Schedtest* program at the same time with different budget and period to see the scheduling execution of CBS tasks.

Filter extracts some information from the output of *Schedtest*, the *tracefile*, and creates a graphic trace file that can be visualized using the program *xfig*. The syntax is:

```
filter tracefile_1 ... tracefile_n > trace.fig
```

where *tracefile_1* ... *tracefile_n* are the file previous created by the *Schedtest* program.

For example, to view the scheduling trace of two CBS tasks, the first with a budget of 20000 and a period of 60000 micro-sec, the second with a budget of 40000 micro-sec and a period of 600000 micro-sec, for 30 seconds, just type:

```
./schedtest 200000 600000 30 > trace1 & ./schedtest
400000 600000 30 > trace2
```

This command runs two tests and creates two output files, called trace1 and trace2.

To create a grafic trace file, type:

```
    ./filter trace1 trace2 > trace.fig
```

This command creates a file called *c.fig*, that can be viewed using the *xfig* program.

4.1.6) Examples

The **qres** module is currently distributed together with two examples that can be used as utilities.

Wrapper is a program that can be used to schedule a task with CBS scheduler. The syntax is the following:

```
    wrapper Q T COMMAND
```

where Q is the budget of the server (in micro-sec), T is the period of the server (in micro-sec), and COMMAND is the name of the command to start (with its absolute path). For example, to run the program *top* with a budget of 20000 and a period of 60000 micro-sec, just type

```
    ./wrapper 20000 60000 /usr/bin/top
```

Cbser is a program that transforms a task running on the Linux system in a CBS task; its syntax is the following:

```
    cbser Q T PID
```

where *Q* is the budget of the server (in micro-sec), *T* is the period of the server (in micro-sec), and *PID* is the Process ID of the task (to know the pid of the task is possible to use the command *top*). For example, if the program *find* is running with the PID 2001, and you want to run it with a budget of 20000 and a period of 60000 micro-sec, just type:

```
    ./cbser 20000 60000 2001
```

4.1.7) Installation instructions

CBS is a loadable Kernel Module that can be inserted on a Linux platform to distribute the CPU time to the system tasks. Before using this feature, the following steps are needed:

- Patching the Linux kernel with the 'Generic Scheduler Patch'
- Compiling the **qres** module
- Installing the **qres** module

Patching the kernel

Locate the directory of the kernel on your Linux system (usually /usr/src/linux). Be sure that the release of the kernel used on your system is 2.4.18 (to verify the kernel release run the command 'uname -r'). If there isn't any kernel directory on the system, or if the kernel release is wrong, download the kernel from *http://www.kernel.org*, and extract the content of the file on a directory of your choice. Copy the 'Generic Scheduler Patch' on the kernel directory, and type

```
patch -p1 < generic_scheduler.patch
```

Now the kernel is patched. To continue, it's necessary to compile and install the patched kernel. Compile the kernel using the following commands:

- make dep
- make bzImage
- make modules
- make modules_install

Restart your system.

Compiling the qres module

If the kernel directory isn't '/usr/src/linux', set the environment variable KERNEL_DIR typing the command 'export KERNEL_DIR=your_kernel_directory'. Go in the directory containing the CBS code (usually qres/src). This directory should contain a file called Makefile. There are many options that can be set during the compilation. The easiest way to compile the module is to type 'make'. This will create a module with the basic features. However, it's possible to specify the following options (which should be appended to the 'make' command):

DEBUG =1 The module prints some warnings during the execution. To read the warnings printed, type 'dmesg'

MULTI=1 Gives a bandwidth of 10% to all Linux tasks. In this way these tasks can execute during the execution of the tasks scheduled by CBS scheduler. In addition this option lets you assign an amount of bandwidth to a set of tasks.

HRT=1 Use this option if you patched the kernel with 'High Resolution Timers' patch. This gives to the system a good precision (below 10msec).

PRECISE_ALLOC=1 Any task scheduled by CBS scheduler executes exactly as specified by its bandwidth (even if there is only one CBS task on the system). This is a possible solution to solve some problems that the CBS algorithm encounters when used to schedule very long jobs.

LOG=... During debugging often happens that the kernel is unable to correctly log all the warning messages printed by the module. This option specifies the directory of the logger to be used, and is usually set when the option DEBUG is set too.

GRUB=1 The module uses the algorithm GRUB instead of CBS (GRUB is capable of reclaiming the unused bandwidth).

GRUB_DEBUG=1 Works like the option 'DEBUG', and should be used when the option 'GRUB' is set.

GRUB_EXTRA_BANDWIDTH=1 When a task blocks, its remaining budget is given to next running task.

GRUB_PWR=1 Adds the feature of power saving to the GRUB scheduler, and must be used only when the option 'GRUB' is set. By now, it is possible to use this feature only on Intel PXA250 processors (see the option 'PXA250').

ARM=1 Enables a cross compiling for ARM processors. It requires the 'arm-linux' utilities (i.e. Arm-linux-gcc).

PXA250=1 Enables a cross compiling for Intel PXA250 processor.

PROC=1 Enables the /proc FileSystem support for CBS scheduler.

QMGR=1 Support the qmgr module. You must enable this modules if you want to use the qos manager module.

Installing the qres module

To install the module in the system, type '**make load**' (equivalent to **insmod**). To remove the module from the system, type '**make unload**' (equivalent to **rmmod**).

4.2) Quality of Service Manager

4.2.1) Summary

- Name **qmgr**
- Description: it is a dynamically loadable module for the Linux kernel that provides QoS management services.
- Author : Luca Marzario (lukesky@gandalf.sssup.it)
- Reviewer:
- Layer: Linux High Level.
- Version 1.0
- Status alpha
- Dependencies: the module depends on the generic scheduler patch and on the qres component. It takes advantage of the High Resolution Patch and of the Preemption Patch [PRK].
- Release date (milestone) April 2003.

4.2.2) Description

The use of computer based solutions for time-sensitive applications is increasingly popular. Important examples include multimedia streaming, video-conference, CD/DVD burning and so forth. Such applications are certainly time-sensitive but classical hard real-time techniques prove unsuitable to support them.

Soft real-time scheduling solutions are commonly regarded as a better solution since they provide temporal isolation, thus allowing for individual task timing guarantees. Moreover, they approximate a fluid allocation of resources, which is certainly a desirable feature to have, for instance, in multimedia applications. However, the problem of finding a correct allocation of bandwidth to the different tasks is still to be considered as a tough problem.

This problem is addressed in this software components and it essentially amounts to finding an appropriate allocation of resources to competing activities. To this regard, a static policy, in our evaluation, confronts unfavorably to a dynamic one. The main argument to support this point of view is that there is often a structure in the dynamic variations of resource requirements for tasks. For instance a typical movie alternates “"fast scenes" (which induce a heavy computation workload) to "slow" dialogues between the characters (which on the contrary require a light computation). To take advantage of this structure, we propose a dynamic adjustment of the bandwidth, in which a software module complementary to the scheduler collects QoS measures for the executing tasks and varies the assigned bandwidth accordingly.

The design of this module is largely based on concepts borrowed from control theory. The application of control theory to scheduling problem has been explored in the recent years. In our context, the design of a feedback controller is greatly aided by the availability of a a dynamic model for CPU reservations [Abe02] combining accuracy with analytically tractability. This result has opened up an interesting research field on appropriate design of a feedback controller. A first result of this kind is shown in [Pal03] and is the theoretical foundation for the algorithms used in the package. However, the exploration of different alternatives is under way.

4.2.3) API / Compatibility

As in the **qres** scheduler component, the only way of setting the parameters for the feedback scheduler is through the **sched_setscheduler**() standard API. The **qmgr** module defines a new data structure:

```
    struct qmgr_param {
       unsigned long int signature;
       unsigned long int cbs_period;
       unsigned long long int qmgr_period;
       unsigned long int qmgr_max_budget;
```

```
        unsigned int delta;
        unsigned long int h;
        unsigned long int H;
        unsigned long int ei;
        unsigned long int Ei;
        unsigned long int e;
        unsigned long int E;
    };
```

The parameters have the following meaning:

- **signature** identifies the structure and must be assigned the value QMGR_SIGNATURE.
- **cbs_period** is the same as the CBS parameter (see Section).
- **qmgr_period** is the period of the task. It may be different from the period of the CBS server. Usually, the period of the task is a multiple of the period of the server.
- **qmgr_max_budget** is the maximum possible budget that can be assigned to the server. The CBS inizial max_budget is inizialized with this value.
- **delta** is the maximum estimated variation of the execution time between two consecutive instances, and it is expressed in microseconds.
- **H** and **h** are the maximum and minimum values, respectively, of each task's instance computation time, expressed in microseconds.
- **Ei** and **ei** are the maximum and minimum values, respectively, of the desired scheduling error.
- **E** and **e** are the maximum and minimum value, respectively, of the guaranteed scheduling error.

To specify that a task has to be served by a CBS server with a feedback scheduler, we have to invoke the **sched_setscheduler** in the following way:

```
    struct qmgr_param parms;
    struct sched_parms sp;

    parms.signature = QMGR_SIGNATURE;
    parms.max_budget = ...;
    // set all parameters...

    sp.sched_size = sizeof(struct qmgr_param);
    sp.sched_p = &parms;

    sched_setscheduler(getpid(), SCHED_QMGR, &sp);
```

4.2.4) Implementation issues

The **qmgr** module requires the presence of the **qres** module, as it completely relies on it for task scheduling. When the **qmgr** module is loaded, it substitutes all the generic scheduler hook with its own functions. When the **sched_setscheduler**() is called, it checks if the signature is equal to QMGR_SIGNATURE. If it is not, it invokes the original hook (i.e. the **qres** function). Otherwise, it simply sets the parameters to the internal data structures, and then invokes the original **qres** hook, passing the correct data structure.

When a task is blocked, the **qmgr** hook is called: it computes the new bandwidth to assign to the CBS task and updates the corresponding **max_budget** field.

When a task forks, the new task by default is not handled by the **qmgr** module to not compromise the execution of the parent.

When a task ends, the relative **qmgr_struct** is freed.

4.2.5) Tests and validation

A test suite has been devised and evaluated, but its application on the module is under way upon the release of this document.

4.2.6) Examples

The **qmgr** module is currently distributed together with one examples that can be used as utilities.

Qmgr_wrapper is a program that can be used to schedule a task with the qos-manager. The syntax is

```
    qmgr_wrapper Qmax Tcbs Tqmgr h H ei Ei e E <COMMAND>
```

where *Qmax* is the maximum budget that can be assigned to the CBS server (in microsec), *Tcbs* is the period of the CBS server (in microsec), *Tqmgr* is the period of the task, *h, H, ei ,Ei, e E* are the execution time's profile of the task (see Section 5.3), and *COMMAND* is the name of the command or of the program to start (with its absoulte path). For example, for a task with a period of 32000 microsec (i.e. an mpegplayer):

```
./qmgr_wrapper 10000 20000 32000 5000 90000  15000 0
20000     12000 /usr/bin/mpegplayer my_video.mpeg
```

Simulplay is a program that simulate the execution of a generic periodic task.

This program can be used together with *qmgr_wrapper* to verify the correctness of the qmgr component (NB: you need a kernel with high resolution timer to use this program).

In more details, the program read the execution time (expressed in usec) of each instance from an input file and waste cpu time for the read time.

The syntax is:

```
./simulplay <file>
```

where *file* is the file of a previous traced of execution times.

In the same directory of simulplay there is a trace of an mpeg video that can be used as a trace file for simulplay (file *mpeg_trace.txt*).

To run an example that executes *qmgr_wrapper* using simulplay as program and *mpeg_trace.txt* as trace file, just load the qres module and type in the utility directory the following script:

```
./run_example.sh
```

4.2.7) Installation instructions

The **qmgr** module is based on the CBS module, so if you want to install this module, the following steps are needed:

- Installing the qres module (see Section 4.7)
- Compiling the **qmgr** module

- Installing the **qmgr** module

Compiling the qmgr module

If the kernel directory isn't '/usr/src/linux', set the environment variable KERNEL_DIR typing the command 'export KERNEL_DIR=your_kernel_directory'. Go in the directory containing the qmgr code (usually qmgr/src) and type 'make'. This will create a module called qmgr.o.

Installing the qmgr module

To install the module in the system, type 'make load' (it's equivalent to the command 'insmod'). To remove the module from the system, type 'make unload' (it's equivalent to the command 'rmmod').

5) CAN/CANopen user guide

Frantisek Vacek – CTU
Pavel Pisa – CTU

Introduction

The CAN/CANopen component consists of four main parts.

- LinCAN driver
- Virtual CAN API (VCA) and libvca
- CAN device
- CAN monitor

Virtual CAN API (VCA) and libvca

5.1.1) Description and implementation issues

The libvca consists of five parts.

- VCA base (vca_base.c)
- Object dictionary acces (vca_od.c)
- SDO processing (vcasdo_fsm.c)
- PDO processing (vca_pdo.c)
- Miscelaneous and utility functions

The main idea of VCA is to have only one interface between application/library and CAN driver (LinCAN). The access to the CAN driver is different in RTLinux and Linux user space. While the function calls are used in RTLinux, the user space application uses /dev/can device. This is why we need VCA.

Next figures shows how to incorporate all the parts of libvca to work together in both spaces.

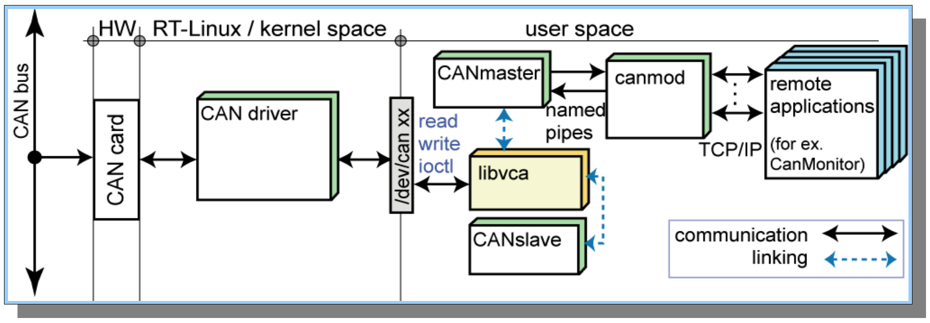


Figure 1.1. Usage of libvca in the Linux user space

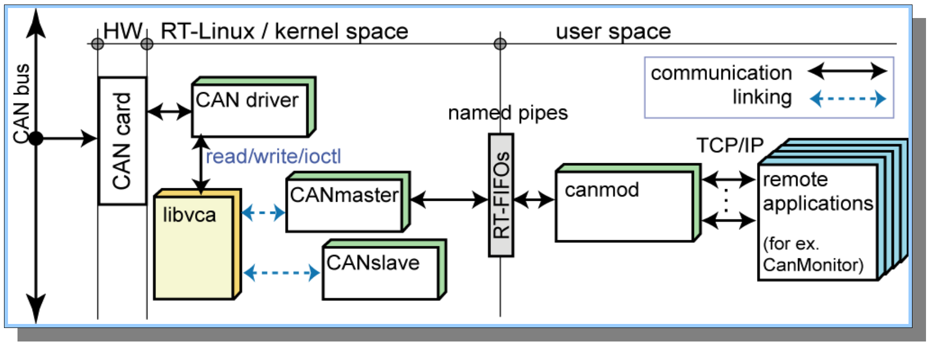


Figure 1.2. Usage of libvca in the RTLinux space

a) VCA base

VCA base is primarily a set of primitive functions used open/close CAN driver and send/receive CAN message. Where are also couple od IOCTLs shielded by VCA base. Most of VCA base is implemented in `vca_base.c`. One part of VCA base is also logging support used by whole CAN/CANopen component. Log support is implemented in `vca_log.c`.

Next code fragment shows simple usage of VCA primitives.

```
vca_handle_t canhandle;
const char *candev = "/dev/can0";
printf("Opening %s\n", candev);
if (vca_open_handle(&canhandle, candev, NULL, 0) != VCA_OK) {
    perror("open");
    exit(1);
}

while (1) {
    struct canmsg_t readmsg;
    int ret = vca_rec_msg_seq(canhandle, &readmsg, 1);
    if(ret < 0) {
        vca_log("cantest", LOG_ERR, "Error reading message from '%'s\n", candev);
    }
    else {
        printf("Received message #%lu: id:%lx data:[",i,readmsg.id);
        for(n=0 ; n<readmsg.length ; n++) {
            if(n > 0) printf(" ");
            printf("%.2x", (unsigned char)readmsg.data[n]);
        }
        printf("]\n");
        i++;
    }
}
```

b) Object dictionary acces

The Object Dictionary (OD) is implemented as a GAVL tree of vcaod_object_t objects. It can be also a GSA array for embeded devices with small amount of memory, but this feature is not implemented yet.

There are three main functions for access to objects in OD.

- vcaod_find_object
- vcaod_get_value
- vcaod_set_value

Using this function one can read or change objects in OD.

```
vca_handle_t canhandle;
const char *candev = "/dev/can0";
printf("Opening %s\n", candev);
if (vca_open_handle(&canhandle, candev, NULL, 0) != VCA_OK) {
   perror("open");
   exit(1);
}

while (1) {
   struct canmsg_t readmsg;
   int ret = vca_rec_msg_seq(canhandle, &readmsg, 1);
   if(ret < 0) {
      vca_log("cantest", LOG_ERR, "Error reading message from '%'s\n", candev);
   }
   else {
      printf("Received message #%lu: id:%lx data:[",i,readmsg.id);
      for(n=0 ; n<readmsg.length ; n++) {
         if(n > 0) printf(" ");
         printf("%.2x", (unsigned char)readmsg.data[n]);
      }
      printf("]\n");
      i++;
   }
}
```

c) SDO processing

The core structure for the SDO processing is a vcasdo_fsm_t. This structure holds all the status information about current SDO handshake and also other information like SDO COB IDs, node number etc. SDO processing library do not contain any synchronous call like select(), read(), write() etc. This aproach gives it independancy on used comunication model. Next code example showes, how to deploy SDO library. Fragment is taken from `canslave.c`.

```
// slave SDO communication loop
vcasdo_fsm_t fsm;
// use default SDO COB IDs
vcasdo_init_fsm(&fsm, 0, 0, node);
while(1) {
   // read CAN driver loop
   struct canmsg_t readmsg;
   int ret = vca_rec_msg_seq(canhandle, &readmsg, 1); // 1.
   if(ret <= 0) continue;

   if(fsm->state == sdofsmIdle) { // 2.
```

```
        // init communicated data in fsm for new communication
        // load object data and prepare FSM for new communication handshake
        do { // 3.
            int cmd;
            vcasdo_read_multiplexor(readmsg.data + 1, &fsm->index, &fsm->subindex);
            cmd = VCA_SDO_GET_COMMAND(readmsg.data[0]);
            if(cmd == VCA_SDO_INIT_UPLOAD_R) {
                uint32_t abort_code;
                ul_dbuff_t *db = &fsm->data;
                vcaod_object_t *odo;
                int l;
                // load data from OD
                odo = vcaod_find_object(&od_root, fsm->index, fsm->subindex, &abort_code);
                if(!odo) {
                    mylog(LOG_ERR, "[%04x:%02x] not found, ABORTING transfer\n", fsm->index,
fsm->subindex);
                    vcasdo_fsm_abort(fsm, abort_code);
                    break;
                    // further function calls returns check are omitted for the example brevity.
                }
                l = vcaod_get_object_data_size(odo, &abort_code);
                ul_dbuff_set_len(db, l);
                // because object can be an array, wee should set parameter array_index to fsm-
>subindex
                l = vcaod_get_value(odo, fsm->subindex, db->data, db->len, &abort_code);
                // FSM is prepared, make it run
                vcasdo_fsm_run(fsm);
            }
            else if(cmd == VCA_SDO_INIT_DOWNLOAD_R) {
                // in case of download nothing special should be done with FSM
                vcasdo_fsm_run(fsm);
            }
        } while(0);
    }

    // now run new communication or continue in previous one (segmented or block transfer)
    // vcasdo_fsm_taste_msg() generate answer CAN message for incoming CAN message
    // according to state of fsm
    // if(fsm->state != sdofsmRun) vcasdo_fsm_taste_msg() returns -1 and it does not do
anything more.
    if(vcasdo_fsm_taste_msg(fsm, &readmsg) == 0) { // 4.
        // bad cobid
        mylog(LOG_DEB, "message REFUSED\n");
    }
    else { // 5.
        if(fsm->state == sdofsmDone) {
            do {
                // SDO transfer complete
                mylog(LOG_INF, "SDO transfer done\n");
                if(!fsm->is_uploader) {
                    // store downloaded data to OD
                    uint32_t abort_code;if
                    ul_dbuff_t *db = &fsm->data;
                    int l;
                    // store downloaded data to OD
                    vcaod_object_t *odo;
                    odo = vcaod_find_object(&od_root, fsm->index, fsm->subindex, &abort_code);
                    l = vcaod_set_value(odo, fsm->subindex, db->data, db->len, &abort_code);
                    // transfer is done, no answer will be sent to the CAN
                    ul_dbuff_set_len(&fsm->data, 0);
                }
            } while(0);
        }
        else if(fsm->state == sdofsmAbort) {
            // SDO transfer aborted
            mylog(LOG_MSG, "SDO transfer ABORTED: error %x '%s'\n", fsm->err_no,
vcasdo_abort_msg(fsm->err_no));
        }
        else if(fsm->state == sdofsmError) {
            // SDO transfer error
            mylog(LOG_MSG, "SDO transfer ERROR: error %x '%s'\n", fsm->err_no,
vcasdo_abort_msg(fsm->err_no));
        }
        else if(fsm->state == sdofsmRun) {
            mylog(LOG_DEB, "SDO transfer RUNNING\n");
        }
        else {
            // unexpected state
            mylog(LOG_ERR, "SDO FSM unexpected state: %i\n", fsm->state);
            fsm->out_msg.length = 0;
        }

        if(fsm->out_msg.length > 0) { // 7.
            // fsm->out_msg.length > 0 signals that message should be sent to CAN
            vca_send_msg_seq(canhandle, &fsm->out_msg, 1);
        }
        // if fsm is not still running reinit communication after all errors, aborting or successfull
handshake
        if(fsm->state != sdofsmRun) {
            vcasdo_fsm_idle(fsm);
        }
    }
}
vcasdo_destroy_fsm(&fsm);
```

The example above is long but lot of code is OD object getting/setting and extraordinary states logging. The main idea is following.

1. get one CAN message
2. check FSM state, if it is currently serving SDO communication or if it is idle
3. if FSM is idle, parse message, get SDO command and get data from OD in case of upload, than make FSM run.
4. give CAN message to the FSM's vcasdo_fsm_taste_msg() function, it ether process message (and change FSM state in appropriate way) or simply refuses it.
5. if message is not refused, check FSM state again.
6. if fsm->out_msg contains data, send data to CAN
7. go to 1. again

CANopen master SDO communication uses the same library in a very similar manner (see `canmaster.c`). The diference is only in fact that master initialize communication (starts with send) while slave allways starts with CAN message read. If the master wants to start SDO communication it should init SDO FSM for upload or download calling `vcasdo_fsm_upload1()` or `vcasdo_fsm_download1()`.

d) 2.1.4. PDO processing

PDO processing is made using a structure vcaPDOProcessor_t. PDO prosessor knows which OD objects are PDO mapped (because it is written in EDS) and it can store/retrieve them to/from OD automaticaly. Core function is `vcaPDOProcessor_processMsg()`. If one call this function with a message just read from CAN, PDO processor check if it is PDO object and it takes care about appropriate behaviour. For more details see `canslave.c`.

e) Miscelaneous and utility functions

This is set of help function to parse text, convert it to number or serialize CAN messages to human readable form.

5.2) CAN device

5.2.1) CAN slave

a) Description and implementation issues

Can slave consist of two main parts

- CANslave core
- HW module

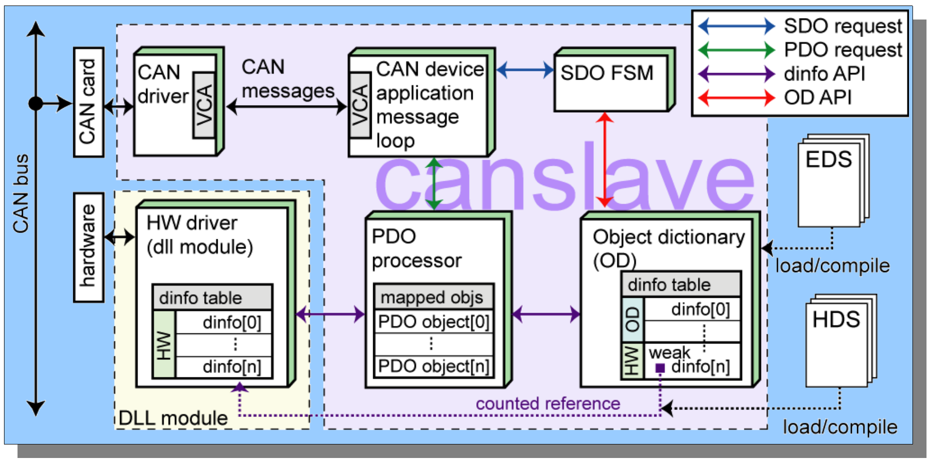


Figure 1.3. CAN Slave block schema

The core is a canslave part which is allways the same. It is responsible for PDO & SDO communication and OD storage. Slave message loop takes CAN messages from CAN. Every message is passed to PDO processor, if it is not processed here than it is passed to SDO FSM (Service Data Object Finite State Machine). SDO FSM process message, if it is part of SDO communication frame or refuse it. If it is SDO, FSM makes all neccessary actions (mainly OD exchange) and prepare CAN answer message. Slave message loop sends such a message back to CAN bus. CAN slave core has also timer (timer is missing in figure below), which is responsible for trigering of synchronous PDOs.

When one starts CAN slave, he should provide EDS and HDS file to it. Slave reads EDS file and it build OD tree according to its contens. In future work will be possible to compile parsed EDS directly to CAN slave core.

EDS Electronic Data Sheet file is a text file describing all objects in the slave object dictionary and its mapping into the PDOs. It has normalized form according to CiA Draft Standard 301.

HDS file contains information which CANopen object in OD is linked to which dinfo in HW module. It is a simple text file with following structure.

```
6000:01 /nascanhw/input01
6200:01 /nascanhw/output01
```

Every line of HDS contains OD object index and subindex and dinfo name to be connected to. Dinfos can be stored in a arbitrary tree like files in directories. First directory on dinfo path is name of DLL which contains this dinfo. For example `/nascanhw/input01` is in `libnascanhw.so`. When HDS file is parsed all needed DLLs are dynamicaly loaded. Thats mean that HW module can consist of more than one *.so file.

What is dinfo? Dinfo is generic structure for passing arbitrary data type among canslave components. Every process value has to have its dinfo in HW module. Every dinfo has getter and setter functions for the primitive data types. At present only the long int and ul_dbuff_t is supported.

You can see the dinfo table on figure inside the OD and also inside the HW module. During CAN device initialization some dinfo structures are allocated. There are two kind of them. HW dinfos resiststing in the driver module are initialized when module is loaded. Every object mentioned in HDS file has also HW dinfo reference in OD. When some object, that do not have HW dinfo (not connected to the hardware), is PDO mapped the fake dinfo is created for that object in OD because the PDO processor allways use only dinfo API for access to any PDO processed object data wheather it comes from HW module or not. All dinfo structures are reference counted, so they are destroyed automatically when they are not needed anymore.

b) Testing

See “CanMonitor testing”.

5.2.2) CAN master

a) Description and implementation issues

CAN master is very similar to the CAN slave. Big difference is in ability of CANmaster to communicate with hierarchically higher application via named pipes. This gives an application opportunity to communicate with CANmaster placed in the user space (via named pipes) or in the RT-linux space (via `/dev/rtfxx`).

For more information see `canmaster.c`.

b) Testing

See CanMonitor testing”.

5.3) CAN monitor

5.3.1) Description and implementation issues

Can monitor component consists of two parts. CAN proxy - **canmond** and Java GUI canmond client **CanMonitor**.

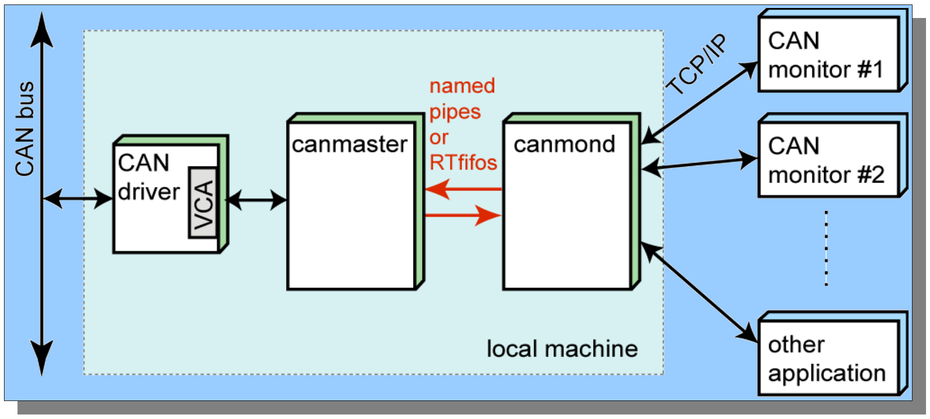


Figure 1.4. CAN monitor component

Canmond is the heard of component. It works like CAN proxy, it is connected using named pipes to the **canmaster** and resends CANopen objects to the all connected applications. Actualy it is a TCP server listenning on port 10001. TCP connection allows clients to be placed wherever on Internet. One can for example read/send CAN messages using a Java applet on his HTML browser. Canmond uses named pipes for communication with canmaster because canmaster can be placed in kernel space (using `/dev/rtfxx`) or in user space and use arbitrary couple of named pipes. This decomposition gives us oportuniti place canmaster in every memory space.

CanMonitor is a GUI Java based application connected to the canmond using UNIX TCP socket. One can send/monitor CAN messages using it. If one has slave EDS (Electronic Data Sheet), he can read/write device OD (Object Dictionary) just by clicking on the mouse.

5.3.2) CanMonitor testing

Program from this package does not need special installation. They can run from any directory. Just type **`make`** in `can/canmon` directory. And copy desired files from `can/_compiled` directory. If you want to compile only one component, type **`make`** in the component's directory.

Restrictions on versions of GNU C or glibc are not known in this stage of project.

Java SDK ver. 1.4 or above is recommended.

Component was tested with real CANopen device WAGO 750-307.

All VCA sources were compiled by GNU C ver. 3.2 and linked with glibc ver. 2.2.5.

All components were also tested with **canmaster** and **canslave** components. In following example is written how.

a) Example 1 - connecting to real CANopen device

Make sure, that CAN driver and the CAN monitor component is installed and works properly. Check that real CANopen device is connected to your CAN card.

Open two terminal windows. In first window launch **canmaster**.

You should see something like this

```
[fanda@mandrake bin]$ ./canmaster
```

```
CANMASTER - CANopen master
canmaster: entering state STATE_INITIALIZING
canmaster: entering state STATE_PREOPERATIONAL
canmaster: entering state STATE_OPERATIONAL
```

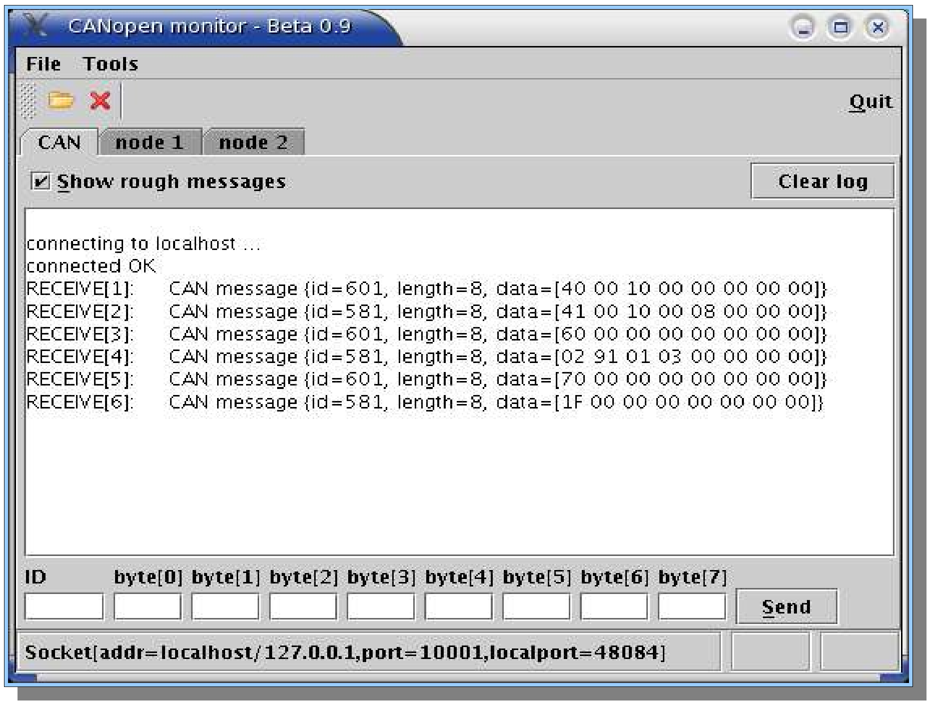
Than you should launch **canmond** on the same machine.

```
[fanda@mandrake bin]$ ./canmond
CANMOND - CAN monitor server
```

If you have a graphical environment with Java installed, you can launch **CanMonitor** with CANopen device EDS file issuing:

```
[fanda@mandrake bin]$ canmonitor -e nascan.eds
loading config from '/home/fanda/.canmonitor/CanMonitor.conf.xml'
connecting to localhost/127.0.0.1
connected OK
```

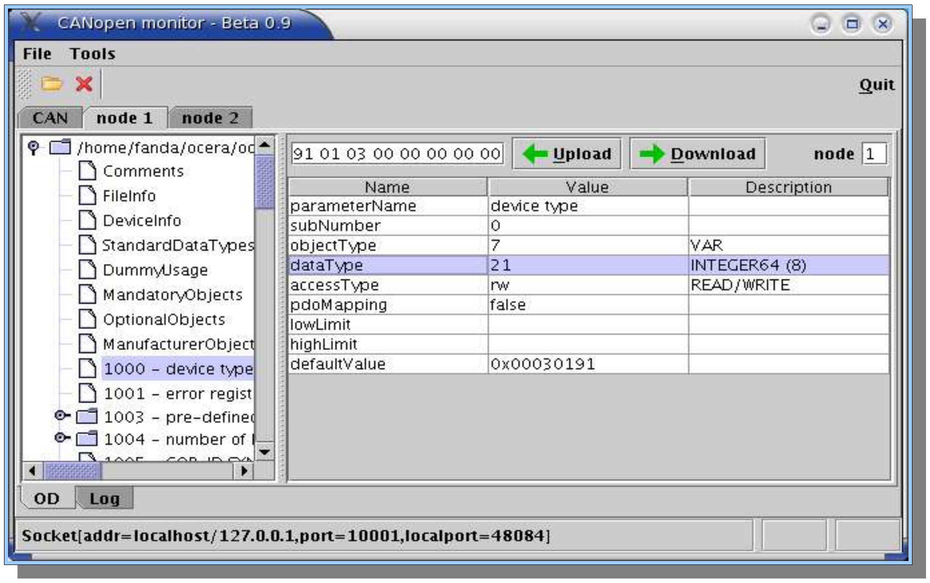
If everything works right, you should see Java application window.


CANopen monitor - Beta 0.9
File Tools
Quit
CAN node 1 node 2
Show rough messages
Clear log
ID byte[0] byte[1] byte[2] byte[3] byte[4] byte[5] byte[6] byte[7]
Send

```
connecting to localhost ...
connected OK
RECEIVE[1]:     CAN message {id=601, length=8, data=[40 00 10 00 00 00 00 00]}
RECEIVE[2]:     CAN message {id=581, length=8, data=[41 00 10 00 08 00 00 00]}
RECEIVE[3]:     CAN message {id=601, length=8, data=[60 00 00 00 00 00 00 00]}
RECEIVE[4]:     CAN message {id=581, length=8, data=[02 91 01 03 00 00 00 00]}
RECEIVE[5]:     CAN message {id=601, length=8, data=[70 00 00 00 00 00 00 00]}
RECEIVE[6]:     CAN message {id=581, length=8, data=[1F 00 00 00 00 00 00 00]}
```

```
Socket[addr=localhost/127.0.0.1,port=10001,localport=48084]
```

Figure 1.5. CanMonitor message window



91 01 03 00 00 00 00 00

Name	Value	Description
parameterName	device type	
subNumber	0	
objectType	7	VAR
dataType	21	INTEGER64 (8)
accessType	rw	READ/WRITE
pdoMapping	false	
lowLimit		
highLimit		
defaultValue	0x00030191	

Socket[addr=localhost/127.0.0.1,port=10001,localport=48084]

Figure 1.6. CanMonitor EDS node window

Now you can load device EDS file and upload/download CANopen objects.

b) Example 2 - connecting to canslave

In this example **canslave** is tested, that means that you do not need any real CANopen device. Tested canslave can resist on same computer as canmaster on can be on other computer connected by CAN bus. If both programs resist on same computer make sure that CAN driver **lincan** was configured to send CAN messages to all other who have open CAN driver on same computer.

Do all steps from example 1. Open terminal window and launch **canslave**. You can launch more canslaves with different node numbers. Do not forget introduce *.EDS file name after -e switch in command line. You should see something like this

```
[fanda@mandrake bin]$ canslave -e nascan.eds
CANSLAVE - CAN slave
canslave: Opening CAN driver: /dev/can0
```

```
canslave: Opening EDS: nascan.eds
canslave: entering state STATE_INITIALIZING
canslave: SYNC COB_ID: 0, SYNC period: 0
canslave: entering state STATE_PREOPERATIONAL
canslave: entering state STATE_OPERATIONAL
```

Than you can load to the running **CanMonitor** next EDS file and work with canslave OD or scan the CAN bus traffic.

5.4) 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).

6) OCERA Real-Time Ethernet

Petr Smolik - 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.

6.1) ORTE Summary

6.1.1) Summary

Name of the component
    OCERA Real-Time Ethernet (ORTE)
Author
    Petr Smolik
ORTE Internet resources
    http://www.ocera.org OCERA project home page
    http://sourceforge.net/projects/ocera OCERA SourceForge project page. The OCERA CVS relative path to ORTE driver sources is `ocera/components/comm/eth/orte`.
    http://www.rti.com Real-Time Innovation home page
Reviewer
    not validated
Layer
    High-level
Version
    orte-0.2.3
Status
    Beta
Dependencies
    Ethernet adapter with a UDP stack.
    Multi-threaded operating system OS
    Memory allocator (functions malloc, free)
Supported OS
- Unix - Linux, FreeBSD, Solaris, MacOS
- Windows
- RTAI with RTNet

Release date
    September 2004

6.2) ORTE Description

6.2.1) Introduction

The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. This protocol is being to submit to IETF as an informational RFC and has been adopted by the IDA group.

6.2.2) The Publish-Subscribe Architecture

The publish-subscribe architecture is designed to simplify one-to-many data-distribution requirements. In this model, an application "publishes" data and "subscribes" to data. Publishers and subscribers are decoupled from each other too. That is:

- Publishers simply send data anonymously, they do not need any knowledge of the number or network location of subscribers.
- Subscribers simply receive data anonymously, they do not need any knowledge of the number or network location of the publisher.

An application can be a publisher, subscriber, or both a publisher and a subscriber.

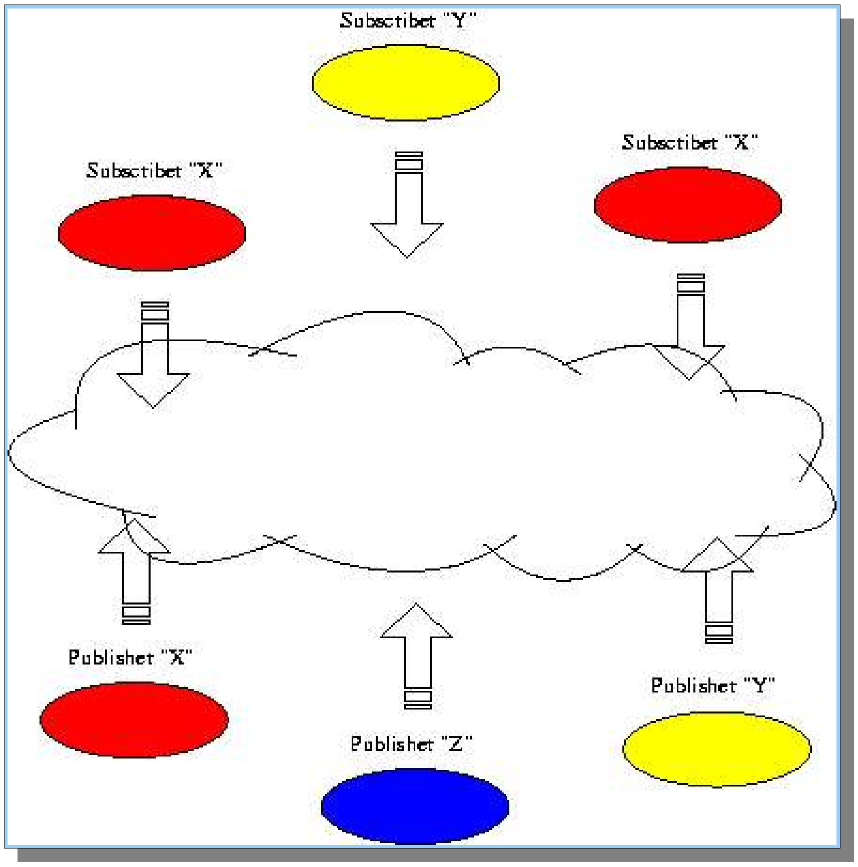


Figure 1-1. Publish-Subscribe Architecture

Publish-subscribe supports anonymous, event-driven transfer between many nodes. The developer simply writes the application to send or receive the data.

Publish-subscribe architectures are best-suited to distributed applications with complex data flows. The primary advantages of publish-subscribe to applications developers are:

- Publish-subscribe applications are modular and scalable. The data flow is easy to manage regardless of the number of publishers and subscribers.

- The application subscribes to the data by name rather than to a specific publisher or publisher location. It can thus accommodate configuration changes without disrupting the data flow.
- Redundant publishers and subscribers can be supported, allowing programs to be replicated (e.g. multiple control stations) and moved transparently.
- Publish-subscribe is much more efficient, especially over client-server, with bandwidth utilization.

Publish-subscribe architectures are not good at sporadic request/response traffic, such as file transfers. However, this architecture offers practical advantages for applications with repetitive, time-critical data flows.

a) The Publish-Subscribe Model

Publish-subscribe (PS) data distribution is gaining popularity in many distributed applications, such as financial communications, command and control systems. PS popularity can be attributed to the dramatically reduced system development, deployment and maintenance effort and the performance advantages for applications with one-to-many and many-to-many data flows.

Several main features characterize all publish-subscribe architectures:

Distinct declaration and delivery. Communications occur in three simple steps:

- Publisher declares intent to publish a publication.
- Subscriber declares interest in a publication.
- Publisher sends a publication issue.

The publish-subscribe services are typically made available to applications through middleware that sits on top of the operating system s network interface and presents an application programming interface.

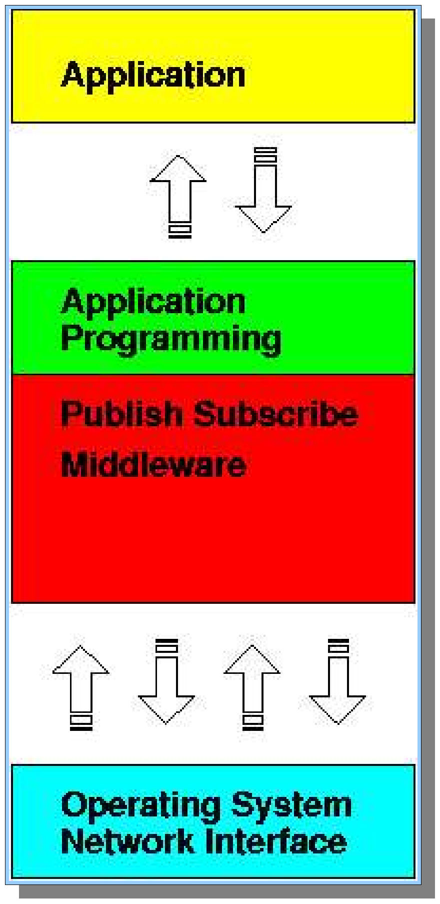

Application
Application Programming
Publish Subscribe Middleware
Operating System Network Interface

Figure 1-2. Generic Publish-Subscribe Architecture

Publish-subscribe is typically implemented through middleware that sits on top of the operating system s network interface. The middleware presents a publishsubscribe API so that applications make just a few simple calls to send and receive publications. The middleware performs the many and complex network functions that physically distribute the data.

The middleware handles three basic programming chores:

- Maintain the database that maps publishers to subscribers resulting in logical data channels for each publication between publishers and subscribers.

- Serialize (also called marshal) and deserialize (demarshal) the data on its way to and from the network to reconcile publisher and subscriber platform differences.
- Deliver the data when it is published.

b) Publish-Subscribe in Real Time

Publish-subscribe offers some clear advantages for real-time applications:

- Because it is very efficient in both bandwidth and latency for periodic data exchange, PS offers the best transport for distributing data quickly.
- Because it provides many-to-many connectivity, PS is ideal for applications in which publishers and subscribers are added and removed dynamically.

Real-time applications require more functionality than what is provided by desktop and Internet publish-subscribe semantics. For instance, real-time applications often require:

- **Delivery timing control:** Real-time subscribers are concerned with timing; for example, when the data is delivered and how long it remains valid.
- **Reliability control:** Reliable delivery conflicts with deterministic timing. Each subscriber typically requires the ability to specify its own reliability characteristics.
- **Request-reply semantics:** Complex real-time applications often have one-time requests for actions or data. These do not fit well into the PS semantics.
- **Flexible delivery bandwidth:** Typical real-time applications include both real-time and non-realtime subscribers. Each subscriber s bandwidth requirements - even for the same publication - can be different.
- **Fault tolerance:** Real-time applications often require "hot standby" publishers and/or subscribers.
- **Thread priority awareness:** Real-time communications often must work without affecting publisher or subscriber threads.
- **Robustness:** The communications layer should not introduce any single-node points-of-failure to the application.

- **Efficiency:** Real-time systems require efficient data collection and delivery. Only minimal delays should be introduced into the critical data-transfer path.

6.2.3) The Real-Time Publish-Subscribe Model

The Real-Time Publish-Subscribe (RTPS) communications model was developed to address these limitations of PS. RTPS adds publication and subscription timing parameters and properties so the developer can control the different types of data flows and achieve their application s performance and reliability goals.

a) Publication Parameters

Each publication is characterized by four parameters: topic, type, strength and persistence. The topic is the label that identifies each data flow. The type describes the data format. The strength indicates a publisher s weight relative to other publishers of the same topic. The persistence indicates how long each publication issue is valid. Next figure illustrates how a subscriber arbitrates among publications using the strength and persistence properties.

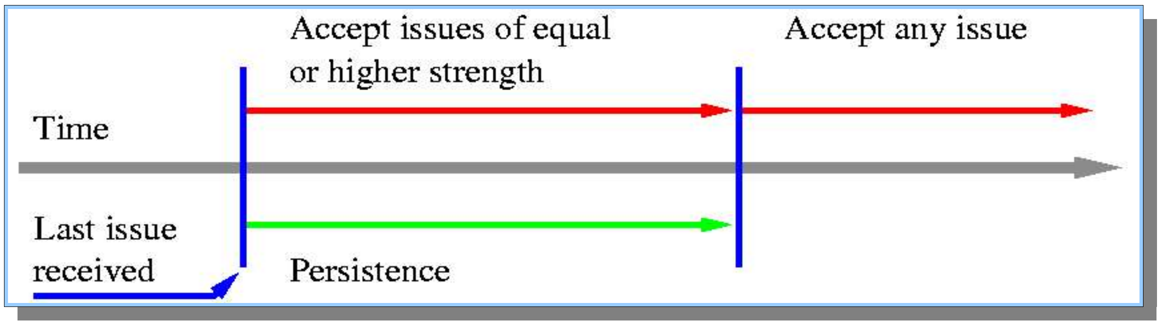


Figure 1-3. Publication Arbitration

Fault tolerant applications use redundant publishers sending publications with the same topic to ensure continuous operation. Subscribers arbitrate among the publications on an issue-by-issue basis based on the strength and persistence of each issue.

When there are multiple publishers sending the same publication, the subscriber accepts the issue if its strength is greater than the last-received issue or if the last issue s persistence has expired. Typically, a publisher that sends issues with a period of length T will set its persistence to some time Tp where Tp > T. Thus, while the strongest publisher is functional, its issues will take precedence over publication issues of lesser strength. Should the strongest publisher stop sending issues (willingly or due to a failure), other publisher(s) sending issues for the same publication will take over after Tp elapses. This mechanism establishes an inherently robust, quasi-stateless communications channel between the then-strongest publisher of a publication and all its subscribers.

b) Subscription Paramters

Subscriptions are identified by four parameters: topic, type, minimum separation and deadline. The topic the label that identifies the data flow, and type describes the data format (same as the publication properties). Minimum separation defines a period during which no new issues are accepted for that subscription. The deadline specifies how long the subscriber is willing to wait for the next issue. Next figure illustrates the use of these parameters.

Figure 1-4. Subscription Issue Separation

Once the subscriber has received an issue, it will not receive another issue for at least the minimum separation time. If a new issue does not arrive by the deadline, the application is notified.

The minimum separation protects a slow subscriber against publishers that are publishing too fast. The deadline provides a guaranteed wait time that can be used to take appropriate action in case of communication delays.

c) Reliability and Time-Determinism

Publish-subscribe can support a variety of message delivery reliability models, not all of which are suitable to real-time applications. The RTPS reliability model recognizes that the optimal balance between time determinism and data-delivery reliability varies between real-time applications, and often among different subscriptions within the same application. For example, signal subscribers will want only the most up-to-date issues and will not care about missed issues. Command subscribers, on the other hand, must get every issue in sequence. Therefore, RTPS provides a mechanism for the application to customize the determinism versus reliability trade-off on a per subscription basis.

The RTPS determinism vs. reliability model is subscriber-driven. Publishers simply send publication issues. However, to provide message delivery reliability, publishers must be prepared to resend missed issues to subscriptions that require reliable delivery.

The RTPS reliability model uses publication buffers publisher and subscriber and retries to ensure that subscribers who need each issue receive them in the proper sequence. In addition, the publisher applies sequence number to each publication issue.

The publisher uses the publication buffer to store history of the most recently sent issues. The subscriber uses its publication buffer to cache the most recently received issues. The subscriber acknowledges issues received in order and sends a request for the missing issue when the most recent issue s sequence number out of order. The publisher responds by sending the missed update again.

Publishers remove an issue from their history buffers in two cases: the issue has been acknowledged by all reliable subscribers or the publisher overflows the history buffer space. Flow control can be implemented by setting high and low watermarks for the buffer. These publication-specific parameters let the publisher balance the subscribers need for issues against its need to maintain a set publication rate.

6.3) 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

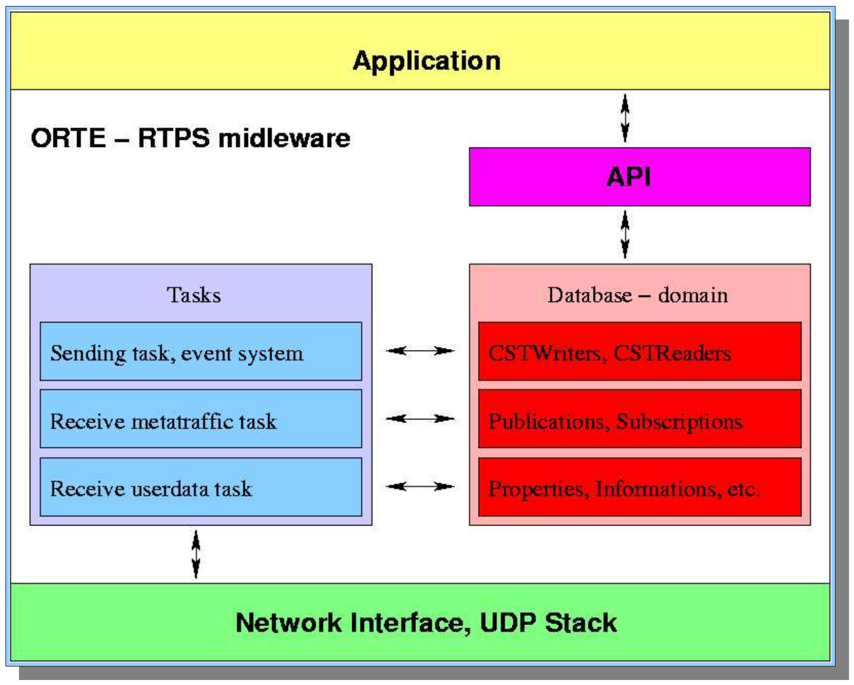


Figure 1-5. ORTE Architecture

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.

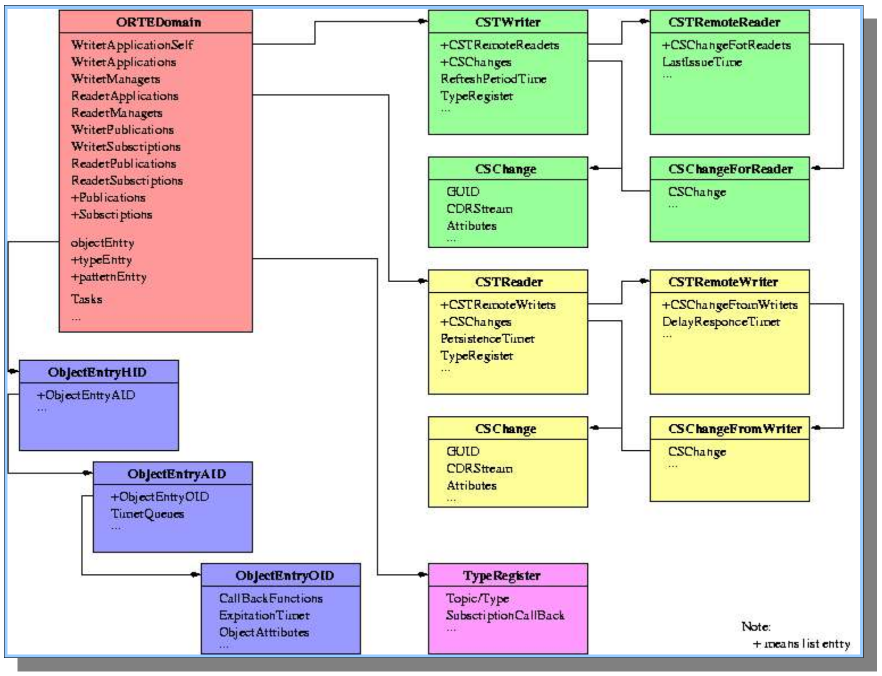


Figure 1-6. ORTE Internal Attributes

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.

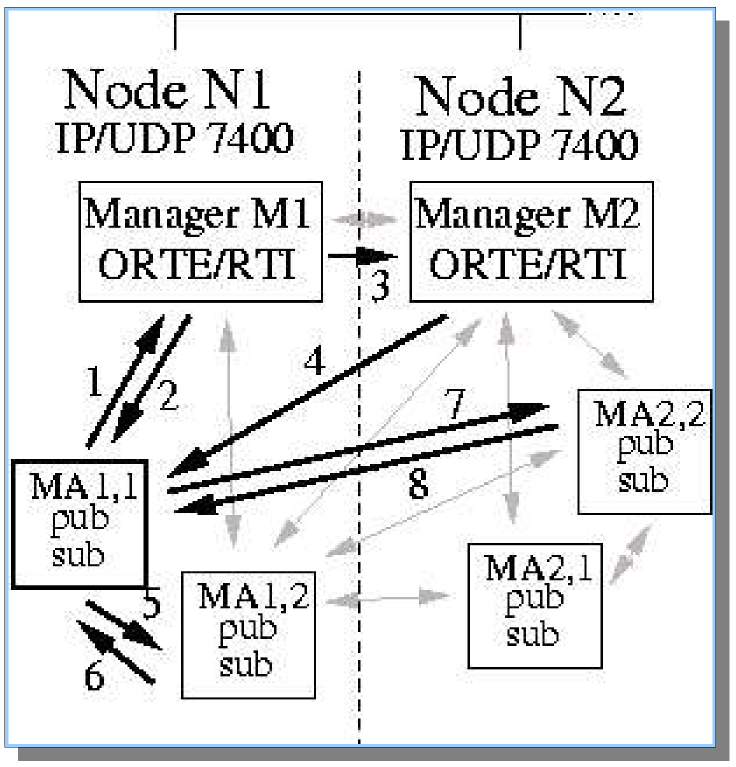


Figure 1-7. RTPS Communication among Network Objects

6.4) 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, ...).

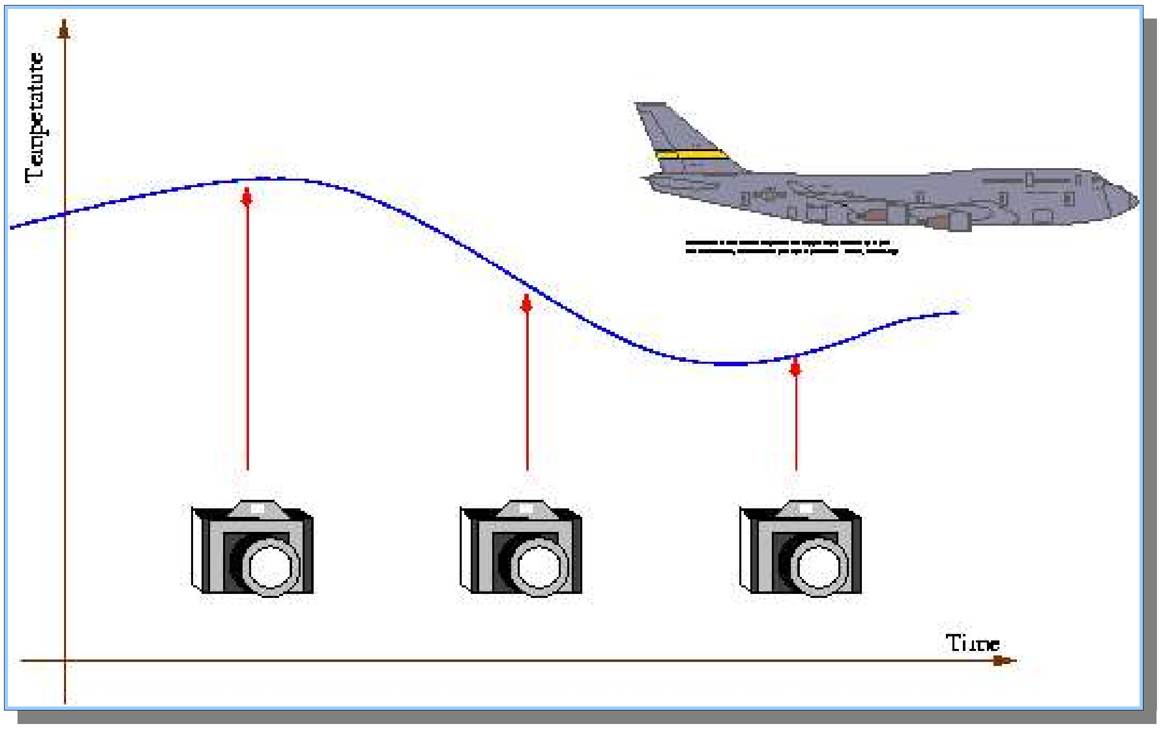


Figure 1-8. Periodic Snapshots of a BestEffort Publisher

- **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.

6.4.1) 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.

6.4.2) 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.

6.4.3) 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`.

6.4.4) 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.

6.5) ORTE Usage Information

6.5.1) 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.

a) 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!

b) 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`.

c) 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.

d) 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.

6.6) 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.

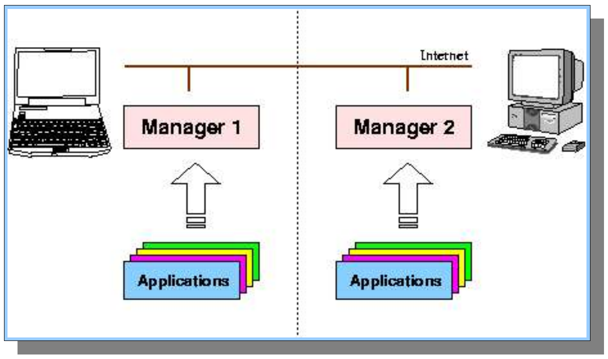


Figure 1-9. Position of Managers in RTPS communication

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.

6.6.1) 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.

6.6.2) 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.

7) Fault-tolerance components

By
A. Lanusse - CEA
P. Vanuxeem - CEA

The main objective of the fault-tolerant work-package in OCERA is to provide two types of facilities : degraded mode management in mono-node applications and redundancy management in distributed applications. The next sections describe briefly the two corresponding frameworks. For more complete descriptions the user can refer to documents D6.1 , D6.2_rep (related to Degraded Mode Management and D6.3_rep and D6.4_rep for Redundancy Management

7.1) Degraded Mode Management.

In this section we describe how to use the fault-tolerance (from now on FT) facilities provided by OCERA V1.0 which concern degraded mode management support for real-time embedded applications.

The objective is to insure, as far as possible, continuity of service (even if degraded) in spite of errors or faults. Errors considered are either timing errors or Kill events detected on application threads. When such errors occur, replacement behaviors are activated depending on rules provided by users during design.

The role of FT_components is to provide transparent run-time modules that detect errors and apply replacement strategies according to user's specification. An additional off-line component, the FT-builder provides support for the specification of desired behaviors. Altogether these components constitute a basic framework to handle degraded mode management that will be progressively enriched.

7.1.1) Introduction

a) Design choices

The design choices for these FT facilities have been based on a declarative approach combined with transparent error handling mechanisms. This choice is driven by the fact that we consider fault-tolerance as non-functional requirements that must not interfere with application core coding for two main reasons: first to get a better control over consistency of fault-tolerance related coding and second, to facilitate maintainability since such requirements may be subject to change. Propagation of requirements change must be handled in a consistent manner which is much more complex if fault-tolerance programming is embedded in the user code.

According to these choices, non functional requirements related to fault-tolerance are collected through a design/build tool and used to instantiate the various run-time components in charge of the behavioural control of the application.

b) Main principles

The approach retained for degraded mode management, relies on a specific programming model providing the concepts of ft_task and application_mode (along with the notions of ft_task_behaviour and application_mode_transition); and on two specific run-time components that implement degraded mode management through activation of ft_task_behaviour change and application_mode switching.

The role of these ft_components is to insure a transparent and safe management of such transitions at task and application level. A particular attention has been paid to the overall application logical and temporal consistency and to a clean resource management so that aborting a task does not produce subsequent tasks blocking. The basic principles of degraded mode management according to this approach are the following: when an error is detected at task level, it triggers a task behaviour change to a degraded mode and propagates the notification of abnormal event at the application level where a decision is taken to apply or not an application mode change.

Degraded mode management is thus based on two levels : first a reactive level provides facilities for immediate handling of abnormal events detected at task level (events considered are Kill or deadline_miss on the current thread of the task); second, facilities for global management of event at application level, possibly involving application mode change.

The declarative approach chosen forces the user to specify transition conditions both at application level and at task level to handle properly reactions to abnormal events. These transition conditions are used to instantiate specific error handling hooks.

Practically, the building tool provides the user with means to specify for each task the related temporal constraints, the different possible alternative behaviours (functions to be activated in threads), and the transition conditions for switching behaviour. It permits also the definition of application modes and of transition conditions for application mode switch. The modelling of the application relies on the task model described in the next section.

7.1.2) Assumptions on application characteristics

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. We list hereunder the main characteristics of applications handled.

a) Task model

- An application consists of a set of periodic tasks.
- The tasks are considered as independent thanks to a specific synchronization model for communication.
- Communication between tasks are restricted to data exchange on a cyclic basis (data are updated at each end of execution cycle of writer and made available to other tasks at each start of new cycle). There is only one writer, the owning task for a data. Client tasks read the data elaborated during previous period.
- No other synchronization is defined between tasks.
- At design the user associates several possible behaviors to a task.

  For the moment two behaviors definitions are expected : a normal behavior and a degraded behavior. These behaviors are actually the routines that will be executed depending on the current task behavior which may be one of : NOT_STARTED, NORMAL, DEGRADED, TERMINATED. In the current implementation, once a task has become DEGRADED, it cannot recover and become NORMAL again.

b) Application model

- The application is defined as having possibly several modes of execution. These are predefined at design and specify the possible degraded modes of functioning. Choice similar to OSEK/VDX[13]
- An application mode defines a specific configuration of active ft_tasks. That is, the specification of the tasks that must be active in the mode and the relevant behavior for each one. This results in a list of pairs (task,behavior).
- The init mode is the mode in which the application will be started (initial configuration).

- Transitions between modes are triggered by the detection of specific events. In the current implementation, the possible triggering events are Kill Event and deadline_miss Event.

c) Events leading to application mode changes

- KILL Event results from the detection of a thread abortion by the kernel due to a software error.
- DEADLINE_MISS Event results from the detection of a deadline_miss by the OCERA EDF scheduler.

  It is planned that User Events can also be triggering conditions for mode change.

d) Application mode transition

On detection of one of the above abnormal events, the application mode can be automatically shifted to an other mode. This transition is defined at design time by the application developer.

- Application mode transition is defined by a triggering event, the task on which the event occurs, the initial application mode, the target application mode.
- When fired an application mode transition has the following effect :
  - the termination of all tasks that are specified TERMINATED in the target application mode;
  - the change of behaviors of all the tasks that are present in the target application mode with a different behavior;
  - The start of tasks that were created but NOT_STARTED in the current application mode and that must be active in the target application mode.

7.1.3) Degraded Mode Management specific entities

In this section, we describe practically what are the entities introduced in order to offer a programming model usable for RTLinux developers. This programming model relies mainly on four entities.

a) FT_task

A FT_task is an encapsulation of a RTLinux real-time periodic task, it actually offers an abstraction for the management of the multi-behaviors of a task.

When a user declares a FT_task, it actually creates an entity that will manage several threads corresponding to the different possible behaviors of the task along with the resources attached to the task. Three primitives have been introduced to manipulate ft_tasks : **ft_task_init()**, **ft_task_create()**, **ft_task_end ()**, these constitute the core of the user API required to program applications.

b) FT_task_behavior and FT_task_routine

The FT_task behavior can be one of NOT_STARTED, NORMAL, DEGRADED or TERMINATED. For each ft_task, two ft_task_routines are defined corresponding to the code to be run within the threads related to its NORMAL and DEGRADED behaviors. These routines are standard RTLinux routine.

c) FT_application_mode

The FT_application_mode entity is a data structure that is used by the run-time FT components implementing the degraded mode management facility. This entity is not seen by the developer in its coding. The data structure is populated by the Design/build tool named **Ftbuilder** that permits interactive definition of application modes and application modes transitions. We describe its utilization in the development process section.

d) FT_application_mode_transition

The FT_application_mode_transition entity as FT_application_mode entity is used internally by the FT run_time components, the developer does not manipulate it directly in its coding but defines it during design through the **Ftbuilder**.

7.1.4) How to use FT Degraded Mode Management Framework

One of the major issue in the introduction of FT facilities was to preserve as far as possible user programming habits and thus to keep unchanged the way he writes tasks routines. We have thus introduced a limited number of primitives mainly used at init to declare what we call ft_tasks while the rest of code writing is kept unchanged. The only important thing concerning ft_tasks is that the user has to provide a routine for each possible behavior (actually two in the current implementation: one for the normal behavior and one for the degraded one).

The introduction of mode management at application level implies that additional information is provided to the system in order to handle abnormal situations in a proper way. This information is actually gathered into internal databases within the run-time ft_components. In order to facilitate the initialization of these internal databases, information collected off-line is processed in order to produce specific files used at init to instantiate them. This way the user has not to provide additional code but only to include these files during the compilation of their application.

a) Development process

The development process proposed to the application developer follows three steps :

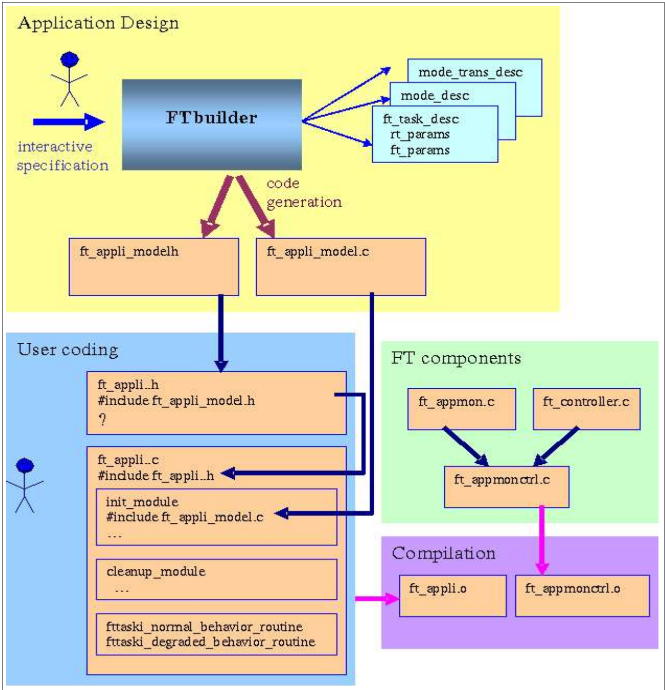


Figure 7.1.1 FT design process

1. **Application design** is achieved interactively using the OCERA **Ftbuilder** tool. The user describes tasks, modes and mode transitions. From these descriptions two files are generated : ***ft_appli_model.c*** and ***ft_appli_model.h***

2. **User Coding** is done manually by the application developer. It consists mainly in writing the code of routines for the application threads identified during previous step.

3. The third step **Compilation** combines files issued by the two previous steps and links it with OCERA **ftcomponents**.

Application design using FT_builder

FT application design covers three main stages which are supported by the FT_builder.

Application Modeling

Application modeling consists in three main aspects :

- identifying the applications tasks and specifying for each one their real_time and FT parameters;
- identifying the applications modes and specifying for each mode the relevant behavior of tasks active in the mode;
- identifying the modes transitions and specifying for each of them : the triggering condition (event and task), the source and destination modes.

Verification

Verification consists in :

- verification of consistency of tasks parameters(real_time and FT)
- verification of transitions consistency

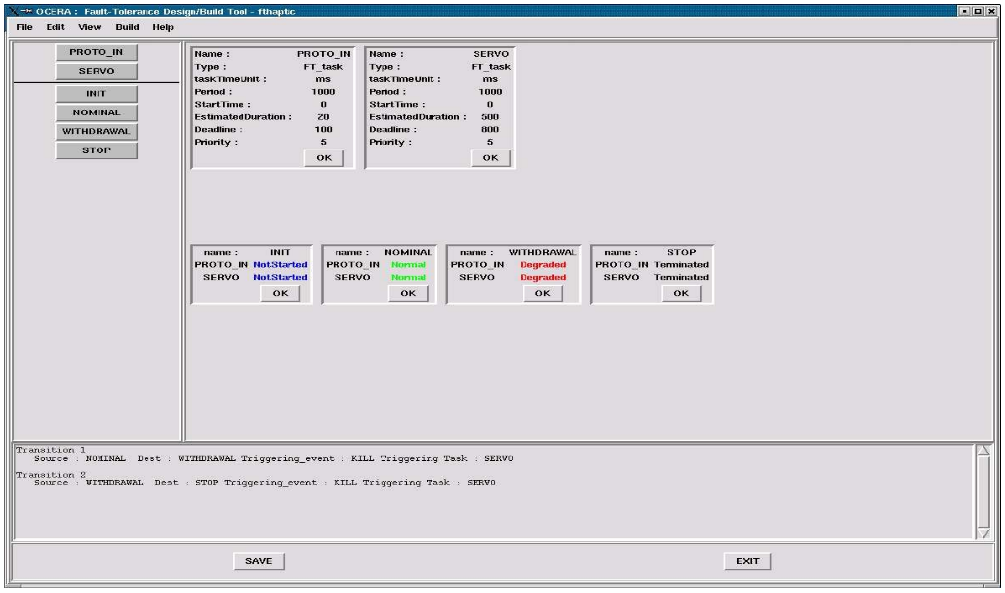
Building (code generation)

Building consists in generating code for application control monitoring. This code consists in two files used to instantiate internal Databases:

- ft_appli_model.h. This file contains taks and modes declarations along with related tables and variables.
- ft_appli_model.c. This file contains calls to init functions that permit to set up two internal DataBases, the AppliModesTable and the AppliControlDataBase.

Ftbuilder Overview

The FT_builder provides various facilities to define tasks, modes, transitions, to edit and view them. It permits also to generate application model files used for application compilation. The following figure shows a general overview of the tool where tasks and modes are displayed.


OCERA : Fault-Tolerance Design/Build Tool - fthaptic
File Edit View Build Help
PROTO_IN
SERVO
INIT
NOMINAL
WITHDRAWAL
STOP
OK
SAVE
EXIT

Name :	PROTO_IN
Type :	FT_task
taskTimeUnit :	ms
Period :	1000
StartTime :	0
EstimatedDuration :	20
Deadline :	100
Priority :	5

Name :	SERVO
Type :	FT_task
taskTimeUnit :	ms
Period :	1000
StartTime :	0
EstimatedDuration :	500
Deadline :	800
Priority :	5

name :	INIT
PROTO_IN	NotStarted
SERVO	NotStarted

name :	NOMINAL
PROTO_IN	Normal
SERVO	Normal

name :	WITHDRAWAL
PROTO_IN	Degraded
SERVO	Degraded

name :	STOP
PROTO_IN	Terminated
SERVO	Terminated

```
Transition 1
   Source : NOMINAL  Dest : WITHDRAWAL Triggering_event : KILL Triggering Task : SERVO

Transition 2
   Source : WITHDRAWAL  Dest : STOP Triggering_event : KILL Triggering Task : SERVO
```

Figure 7.1.2 FTbuilder : global view

Tasks and modes are listed in the left part of the screen while details are displayed on the right part. The bottom part is devoted to the display of messages or to list entities such as tasks, modes and mode transitions (menu Edit/<entity>/List<entity>. Modes are described by the list of tasks and related behaviors that must be applied in the mode. In the next sections we review dedicated acquisition windows for task, modes, and transition specification.

Task specification

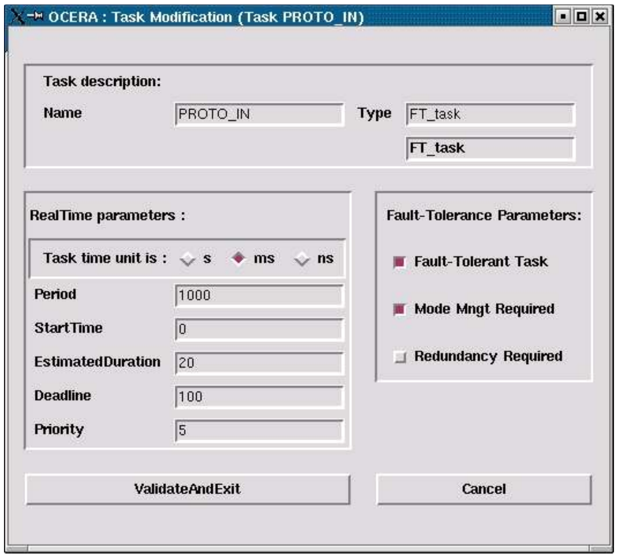
The task specification consists in providing ft_task real_time and ft_task parameters using the FT_builder.

The task specification consists in providing ft_task real_time and ft_task parameters. This is done using the FT_builder NewTask or ModifyTask facility.

The consistency of parameters entered is checked before storing information and it is planned that in the future a global analysis of these parameters will be done (RMA tool). In the current (V1.0) implementation, only FT_tasks are handled.

Real-time parameters associated to a ft_task are :

- Period,
- Start Time,
- Estimated Duration
- Deadline
- Priority


OCERA : Task Modification (Task PROTO_IN)

Task description:

Name	PROTO_IN	Type	FT_task
			FT_task

RealTime parameters :

Task time unit is : s [✓ ms] ns

Period	1000
StartTime	0
EstimatedDuration	20
Deadline	100
Priority	5

Fault-Tolerance Parameters:

[✓ Fault-Tolerant Task]

[✓ Mode Mngt Required]

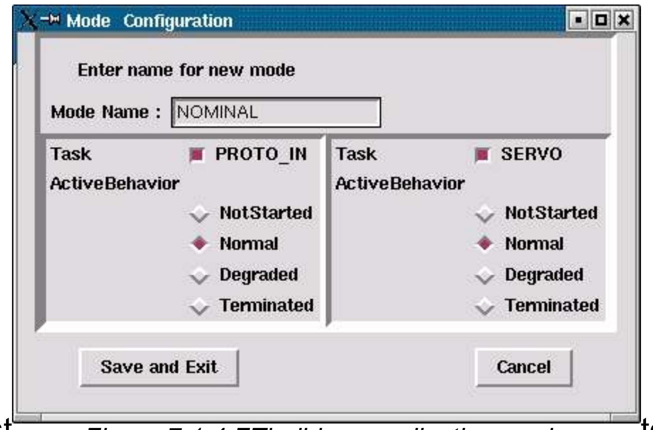
Redundancy Required

ValidateAndExit Cancel

Figure 7.1.3 FTbuilder : task specification

Mode specification

The mode specification consists in selecting for each application task the behavior expected in the mode. This is done using the option NewMode facility.


Mode Configuration

Enter name for new mode

Mode Name : NOMINAL

Task	■ PROTO_IN
ActiveBehavior	NotStarted [✓ Normal] Degraded Terminated

Task	■ SERVO
ActiveBehavior	NotStarted [✓ Normal] Degraded Terminated

Figure 7.1.4 FTbuilder : application mode specification

The user has just to enter a mode name (which must be different from an existing one) and to select for each task, the right behavior to adopt in this mode. Once these choices have been made, just save and exit (or cancel if you wish).

In this figure, the NOMINAL mode is defined. This configuration consists of two active ft_tasks PROTO_IN and SERVO running their normal behavior. Other modes have been defined for the application : namely: INIT, WITHDRAWAL and STOPPED. They appear in the global view of the Ftbuilder .

Mode Transition specification

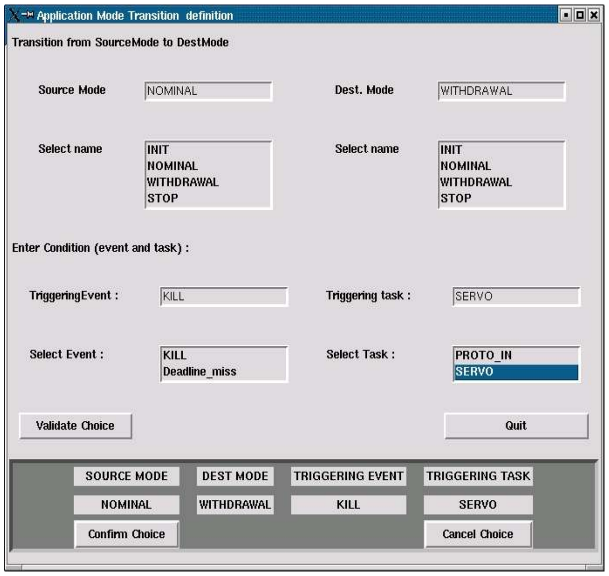
Specifying a mode transition consists in providing :

- a Source Mode

- a Destination Mode
- a Transition Condition

where the transition condition consists of

  - a triggering event (the event can be KILL or deadline Miss)
  - a triggering task (the task that receives the event)



Application Mode Transition definition
Transition from SourceMode to DestMode
Enter Condition (event and task) :
Validate Choice
Quit
Confirm Choice
Cancel Choice

Source Mode: NOMINAL
Dest. Mode: WITHDRAWAL

Select name: INIT, NOMINAL, WITHDRAWAL, STOP
Select name: INIT, NOMINAL, WITHDRAWAL, STOP

TriggeringEvent : KILL
Triggering task : SERVO

Select Event : KILL, Deadline_miss
Select Task : PROTO_IN, SERVO

SOURCE MODE	DEST MODE	TRIGGERING EVENT	TRIGGERING TASK
NOMINAL	WITHDRAWAL	KILL	SERVO

Figure 7.1.5 Ftbuilder: application mode transition specification

In this example, the transition between NORMAL and WITHDRAWAL modes is triggered when a Kill event occurs on SERVO task.

The user must enter the source and dest. Modes, then select the type of event and the triggering task.

Then he can validate his choice, a confirmation step displays a summary of the choices made for the transition, and waits for confirmation or cancellation.

The user can list the existing defined transitions with the Edit/Modes/ListModesTransitions facility.

The transitions are displayed in the bottom part of the main window.

Code generation

Once the desired entities have been defined, the user can generate the corresponding appli_model files using the build menu in the main window. Two files are produced **ft_model.h** and **ft_model.h**. By default they are located under :

<Ftbuilder_dir>/appli_generated_files/<model_name>

User application coding

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 level

```
-> 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

- 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 chapter eight). Static scheduling plan on ft-tasks has to be faisable.
  The following using restriction and recommandation for these real-time parameters are :
    - 1 ms <= period < =100 s
    - 0 <= start_time < period
    - 0 < estimated_duration < periode
    - 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.

headers

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

```
#include "ft_api_appmon_appli.h"  // include api ft_appli_monitor
#include "ft_appli_model.h"          // include modele application model (header file)
```

init_module

The init_module function looks like regular RTLinux init_module except for the initialization of ft_tasks.

Since ft_tasks correspond to an encapsulation of several threads, specific primitives have been introduced for the init, creation and deletion of such ft_tasks (see FT_API section).

Within the init_module, two primitives are used :

- **ft_task_init**

  This primitive initializes data structures related to ft_tasks within internal FT_components. Its arguments are : a name, pointers to normal and degraded routines associated to the ft_task along with arguments and scheduling parameters (for normal and degraded behavior).

- **ft_task_create**

  This primitive creates and starts ft_task threads. Two arguments are provided : the ft_task_id and the behavior to be activated. Usually the Normal behavior is activated which means that two threads are created respectively with normal and degraded routines and the thread with normal behavior is made periodic while the other one (with degraded behavior routine) is suspended.

```
int init_module(void) {                          // init module RTLinux
   #include "ft_appli_model.c"                  // include application model
  FT_sched_param ap_normal_sched_param, ap_degraded_sched_param;
  ...
   for (ap_task_id=1; ap_task_id < APPLI_TASKS_MAX_NB+1; ap_task_id++)
   { // tasks building loop
     // FT_task initialization
     strcpy(&ap_task_name_tab[ap_task_id][0],"PROTO_IN");
     // Initialization of FT scheduling parameters
     ap_normal_sched_param.prio=SERVO_PRIORITY;
     ap_normal_sched_param.period=SERVO_PERIOD;
     ap_normal_sched_param.start_time=SERVO_START_TIME;
     ap_normal_sched_param.deadline=SERVO_DEADLINE;
     ap_normal_sched_param.duration=SERVO_DURATION;
     // Idem for ap_degraded_sched_param
     ...
     ft_task_init(                                    // call to ft_task_init
       &ap_task_name_tab[ap_task_id][0],          // ft_task_name
        ap_normal_behavior_routine,              // normal routine (pointer)
        ap_degraded_behavior_routine,            // degraded routine (pointer)
        ap_normal_behavior_rout_arg,             // normal routine argument
        ap_degraded_behavior-rout_arg,           // degraded routine argument
        ap_normal_sched_param,                   // normal scheduling parameters
//  (prio, period, deadline)
        ap_degraded_sched_param );               // degraded sched.  Parameters
                                      // (prio, period, deadline)
     // FT_task creation with NORMAL behavior
     //     Two threads are created :
     //         - one for NORMAL behavior which is made periodic
     //         - one for Degraded behavior which is suspended
     ap_task_mode=FT_TASK_NORMAL;
     ft_task_create(                          // call to ft_task_create
       ap_task_id,                           // ft_task Id
       ap_task_mode);                         // ft_task_mode (one of
                    //FT_TASK_NORMAL,FT_TASK_DEGRADED,FT_TASK_NOT_STARTED)
   }
}
```

cleanup_module

As the init_module function, the cleanup_module function looks like regular RTLinux cleanup_module except for the deletion of ft_tasks where the developer has to use the ft_task_end primitive to terminate properly the threads related to ft_tasks , free ressources and cleanup data structures.

```
void cleanup_module(void) {                // deleting application module
   for (ap_task_id=1; ap_task_id < APPLI_TASKS_MAX_NB+1; ap_task_id++)
   {          // for each ft_task
   ft_task_end(ap_task_id);                // delete ft_task and free ressources
  ...
   }
}
```

Routine coding for normal and degraded behaviors

For each ft_task, two routines are to be defined , one corresponding to the code to be run for a normal behavior of the task and one corresponding to the code to be run for a degraded behavior of the task. In both cases, the structure of the code is a usual rtlinux periodic task with a main infinite loop and a body starting by a wait for the next period.

normal behavior routine

```
void *ap_normal_behavior_routine(void *arg) {// routine for normal behavior
  ap_task_id = (int) arg;
  while(1) {                        // main loop
    pthread_wait_np();                   // wait for periodic wakeup
    no_cycle++;                        // period
    ... DO SOMETHING                          // Body for normal behavior
    }
  return (void *) 0;
}
```

degraded behavior routine

The degraded behavior routine has exactly the same structure as the normal one. The difference is in the content of the behavior. For instance, we can program a different algorithm, or a smooth stop. Usually, the thread with the degraded mode is suspended until an error occurs on the thread with normal behavior and is resumed then.

```
void *ap_degraded_behavior_routine(void *arg) {// degraded behavior routine
   ap_task_id = (int) arg;
```

```
  rtl_printf("\n\nApplication : thread %d switching to running
            (degraded_behavior_%d)", pthread_self(), ap_task_id);
  while(1) {                // main loop
    pthread_wait_np();       // wait for periodic wakeup
    no_cycle++;             // period
   ... DO SOMETHING          // Body for degraded behavior
  return (void *) 0;
}
```

Application compiling

In order to compile an ft application (here ftappli), 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

Here ftappli may be for instance an application (or example) under the directory either $OCERA_DIR/app/fthaptic or $OCERA_DIR/components/ft/ftcontroller/examples/ftappli directory.

The compilation of the ftappli module, is achieved by applying the following commands at the ocera (or ftappli) directory level :

– Clean the application directory :

```
$ make clean
```

Old ftappli.o file is cleaned up if it exists.

- Compile the application module:

```
$ make all
```

The ftappli.o module is now available under the application directory.

b) Running an application

The ftappli Makefile doesn't install and execute the ftappli module. It only compiles it (produces ftappli.o).

The general compilation and installation procedure for user application is not finalized yet.

So for the moment, use the Makefile in ft/ftcontroller/examples directory that installs and execute both the ft module named ftappmonctrl.o and application example module ftappli.o.

The procedure is the following :

- Go to the application (or example) directory level :

```
$ cd $OCERA_DIR/app/fthaptic
```

or :

```
$ cd $OCERA_DIR/components/ft/ftcontroller/examples/ftappli/
```

- 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
```

or :

```
# make start
...
# make stop
```

- Get the modules execution traces:

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

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

Example of application trace

Two application modes are defined :

Mode NOMINAL : in which all ft_tasks have a normal behavior

Mode WITHDRAWAL : in which ft_tasks have a degraded behavior;

A mode transition is defined from NOMINAL to WITHDRAWAL on occurrence of pthread_kill on .

Execution trace of ft appli example

The execution trace can be consulted in dmesg.

Here we can follow the main steps of this example.

1. Application loading, init and start

```
Dec  7 11:32:46 is002404 kernel: mbuff: kernel shared memory driver v0.7.2 for Linux 2.4.18-
ocera-1.1
Dec  7 11:32:46 is002404 kernel: mbuff: (C) Tomasz Motylewski et al., GPL
Dec  7 11:32:46 is002404 kernel: mbuff: registered as MISC device minor 254
Dec  7 11:32:46 is002404 kernel: RTLinux Extensions Loaded (http://www.fsmlabs.com/)
Dec  7 11:32:49 is002404 kernel:
Dec  7 11:32:49 is002404 kernel: ***************
Dec  7 11:32:49 is002404 kernel:  FT_Controller
Dec  7 11:32:49 is002404 kernel: ***************
Dec  7 11:32:49 is002404 kernel:
Dec  7 11:32:49 is002404 kernel: ******************
Dec  7 11:32:49 is002404 kernel:  FT_Appli_Monitor
Dec  7 11:32:49 is002404 kernel: ******************
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: ******************
Dec  7 11:32:50 is002404 kernel:  FT_Haptic
Dec  7 11:32:50 is002404 kernel: ******************
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: Application :
Dec  7 11:32:50 is002404 kernel: PROTO_IN_EVENT_CYCLE=60
Dec  7 11:32:50 is002404 kernel: SERVO_EVENT_CYCLE=60
Dec  7 11:32:50 is002404 kernel: PROTO_IN_TEST_CYCLE=20
Dec  7 11:32:50 is002404 kernel: SERVO_TEST_CYCLE=20
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: Application : PROTO_IN period=1000000000
Dec  7 11:32:50 is002404 kernel: Application : PROTO_IN deadline=0
Dec  7 11:32:50 is002404 kernel: Application : Function init_module : ft_task_init PROTO_IN
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: Application : SERVO period=1000000000
Dec  7 11:32:50 is002404 kernel: Application : Function init_module : ft_task_init SERVO
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: Application : ap_task_id_tab[1]=1
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: Application : ap_i=1 ap_task_behavior=FT_TASK_NORMAL
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: PROTO_IN normal : ft-task 1, thread -818053120 started,
normal or not started behavior
Dec  7 11:32:50 is002404 kernel: PROTO_IN normal : ft-task 1, thread -818053120 switching
to wait, normal or not started behaviorDec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: PROTO_IN normal : ft-task 1, thread -818053120 switching
to running, normal_behavior
```

ft_task PROTO_IN is started with normal behavior

```
Dec  7 11:32:50 is002404 kernel: PROTO_IN normal : VPr=0
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: Application : ap_task_id_tab[2]=2
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: Application : ap_i=2 ap_task_behavior=FT_TASK_NORMAL
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: SERVO normal : ft-task 2, thread -850526208 started, normal
or not started behavior
Dec  7 11:32:50 is002404 kernel: SERVO normal : ft-task 2, thread -850526208 switching to
wait, normal or not started behavior
Dec  7 11:32:50 is002404 kernel:
Dec  7 11:32:50 is002404 kernel: SERVO normal : ft-task 2, thread -850526208 switching to
running, normal_behavior
```

ft_task SERVO is started with NORMAL behavior. PROTO_IN and SERVO exchange data

```
Dec  7 11:32:50 is002404 kernel: SERVO normal : data=0
Dec  7 11:32:51 is002404 kernel:
Dec  7 11:32:51 is002404 kernel: PROTO_IN normal : VPr=0
Dec  7 11:32:51 is002404 kernel:
Dec  7 11:32:51 is002404 kernel: SERVO normal : data=0
Dec  7 11:32:52 is002404 kernel:
Dec  7 11:32:52 is002404 kernel: PROTO_IN normal : VPr=0
Dec  7 11:32:52 is002404 kernel:
Dec  7 11:32:52 is002404 kernel: SERVO normal : data=0
Dec  7 11:32:53 is002404 kernel:
Dec  7 11:32:53 is002404 kernel: PROTO_IN normal : VPr=1
Dec  7 11:32:53 is002404 kernel:
Dec  7 11:32:53 is002404 kernel: SERVO normal : data=1
Dec  7 11:32:54 is002404 kernel
...
```

A KILL event on ft_task SERVO is produced to test event detection and the task switching mechanism

```
Dec  7 11:33:49 is002404 kernel: SERVO normal : ft-task 2, thread -850526208 cancelling,
no_cycle=60, normal_behavior
Dec  7 11:33:49 is002404 kernel: FT_Controller : ft_task_id=2 PTHREAD_KILL
Dec  7 11:33:49 is002404 kernel: FT_Appli_Monitor : Function ft_notify_failed_thread
Dec  7 11:33:49 is002404 kernel: FT_Appli_Monitor : before switch ft_current_appli_mode= {2 ,
NOMINAL}
Dec  7 11:33:49 is002404 kernel:
```

The THREAD_KILL event is detected by the ft_controller

A notification is issued to ft_appli_monitor

```
Dec  7 11:33:49 is002404 kernel: FT_Appli_Monitor : ft_new_appli_mode= {3 , WITHDRAWAL}
Dec  7 11:33:49 is002404 kernel:
Dec  7 11:33:49 is002404 kernel: SERVO degraded : ft-task 2, thread -852590592 switching to
running, degraded_behavior
Dec  7 11:33:49 is002404 kernel:
```

ft_task SERVO has been switched to degraded behavior as a result of local reaction to abnormal event.

```
Dec  7 11:33:49 is002404 kernel: SERVO degraded : data=57
Dec  7 11:33:49 is002404 kernel:
Dec  7 11:33:49 is002404 kernel: PROTO_IN degraded : ft-task 1, thread -242483200
switching to running, degraded_behavior
Dec  7 11:33:49 is002404 kernel:
```

ft_task PROTO_IN has been switched to degraded behavior as a result of application mode change. ft_task SERVO has already commutted to degraded mode so nothing more is done for application mode change completion. The new application mode is WITHDRAWAL

```
Dec  7 11:33:49 is002404 kernel: PROTO_IN degraded : VPr=57
Dec  7 11:33:50 is002404 kernel:
Dec  7 11:33:50 is002404 kernel: SERVO degraded : data=57
Dec  7 11:33:50 is002404 kernel:
...
Dec  7 11:34:47 is002404 kernel: PROTO_IN degraded : VPr=114
Dec  7 11:34:48 is002404 kernel:
Dec  7 11:34:48 is002404 kernel: SERVO degraded : data=114
Dec  7 11:34:48 is002404 kernel:
```

A second KILL is emitted towards ft_task SERVO.

```
Dec  7 11:34:48 is002404 kernel: SERVO degraded : ft-task 2, thread -852590592 cancelling,
no_cycle=60, degraded_behavior
Dec  7 11:34:48 is002404 kernel:
Dec  7 11:34:48 is002404 kernel: PROTO_IN degraded : VPr=115
Dec  7 11:34:48 is002404 kernel:
Dec  7 11:34:48 is002404 kernel: FT_Controller : ft_task_id=2 PTHREAD_KILL
Dec  7 11:34:48 is002404 kernel: FT_Controller : Cancel the degraded thread !!!
Dec  7 11:34:48 is002404 kernel: FT_Controller : ft task id              ---   2
Dec  7 11:34:48 is002404 kernel: FT_Controller : ft degraded thread kid   ---   cd2e8000
```

The event is detected by ft_controller and corresponding thread is killed. A notification is issued to ft_application monitor.

```
Dec  7 11:34:48 is002404 kernel: FT_Appli_Monitor : Function ft_notify_failed_thread
Dec  7 11:34:48 is002404 kernel: FT_Appli_Monitor : before switch ft_current_appli_mode= {3 ,
WITHDRAWAL}
Dec  7 11:34:48 is002404 kernel:
Dec  7 11:34:48 is002404 kernel: FT_Appli_Monitor : ft_new_appli_mode= {4 , STOP}
Dec  7 11:34:48 is002404 kernel:
```

New application mode must be STOP.

```
Dec  7 11:34:48 is002404 kernel: FT_Controller : Cancel the degraded thread !!!
Dec  7 11:34:48 is002404 kernel: FT_Controller : ft task id              ---   1
Dec  7 11:34:48 is002404 kernel: FT_Controller : ft degraded thread kid   ---   f18c0000
Dec  7 11:36:23 is002404 kernel:
```

The thread corresponding to ft_task PROTO_IN is killed, then the application is terminated and unloaded.

```
Dec  7 11:36:23 is002404 kernel: Application : CLEANUP application !!!
Dec  7 11:36:23 is002404 kernel: unloading mbuff
Dec  7 11:36:23 is002404 kernel: mbuff device deregistered
```

7.1.5) Degraded Mode Management: architecture overview

The current implementation architecture is based on the OCERA hard real-time platform which consists of RTLinux hard RT level extended with OCERA components (mainly Posix extensions and various high level schedulers implementing algorithms such as cbs, edf).

The fault-tolerance components consists of two complementary components: **ftappmon** and **ftcontroller**; that provide a framework for implementing degraded mode management support. Located at the application level, they provide both global monitoring of application and local control of execution. The current version handles only Hard real-time RTLinux level. The two components must be used together.

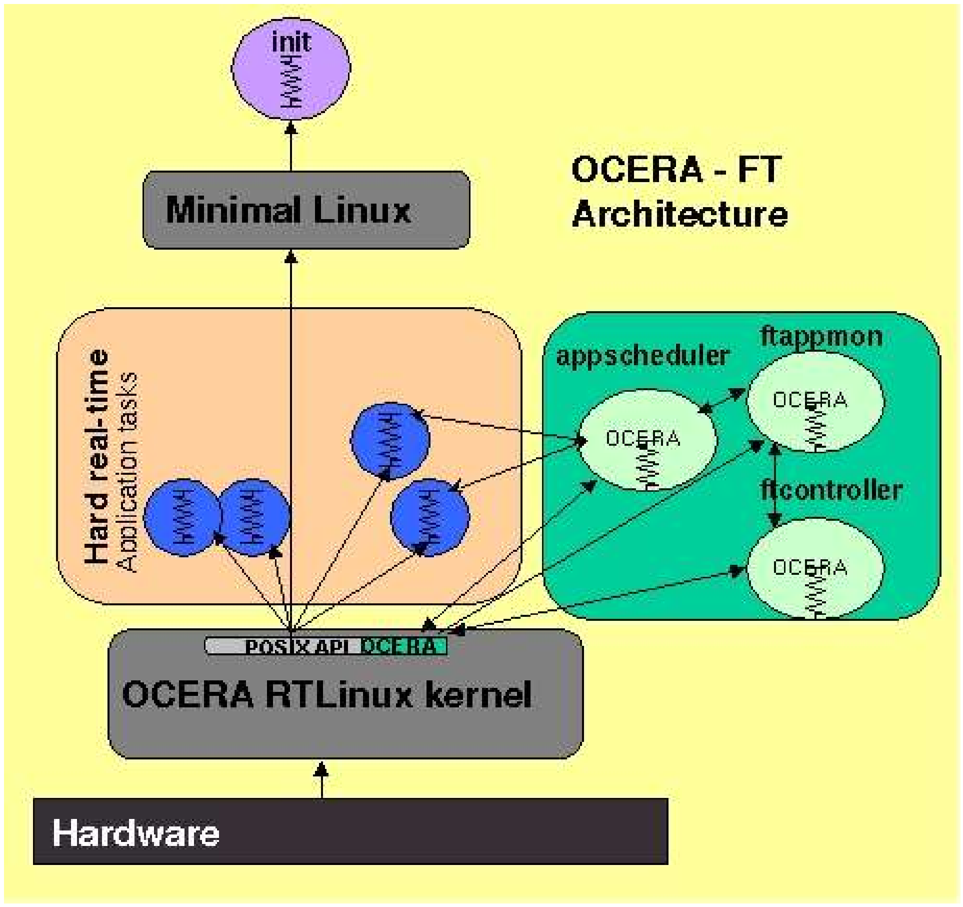


Figure 7.1.6 FT Degraded Mode Management Architecture Overview

The **ftappmon** (application fault-tolerance monitor) component is in charge of applying application mode change on notification (from **ftcontroller**) of an abnormal situation related to a particular ft_task. A ft_task is a user task for which fault-tolerance is required. It involves two alternate threads, one implementing a nominal behavior and the second one implementing a degraded behavior. These two threads are created during the application init phase but only the nominal behavior is made active. The **ftappmon** defines the impact of the event on the current running tasks and decides of a new configuration (stops tasks, switch tasks modes, activates new tasks).

The **ftcontroller**component is a low-level RTLinux application component in charge of controlling execution of ft_tasks. On detection of an abnormal situation on a thread related to a ft-task (deadline miss or abort), the ftcontroller activates if possible the alternate thread (degraded behavior thread) and propagates the event to the **ftappmonitor.**

The implementation of the FT components in kernel space provides thus :

- an application programming interface **FT-API**,
- a component **ftappmon** dedicated to the monitoring of the application,
- a component **ftcontroller** dedicated to the control of the ft-tasks behaviors,
- a cooperation between ftappmon and ftcontroller to manage switching behavioral operations,
- an interaction with the **scheduler** in order to detect abnormal kernel events.

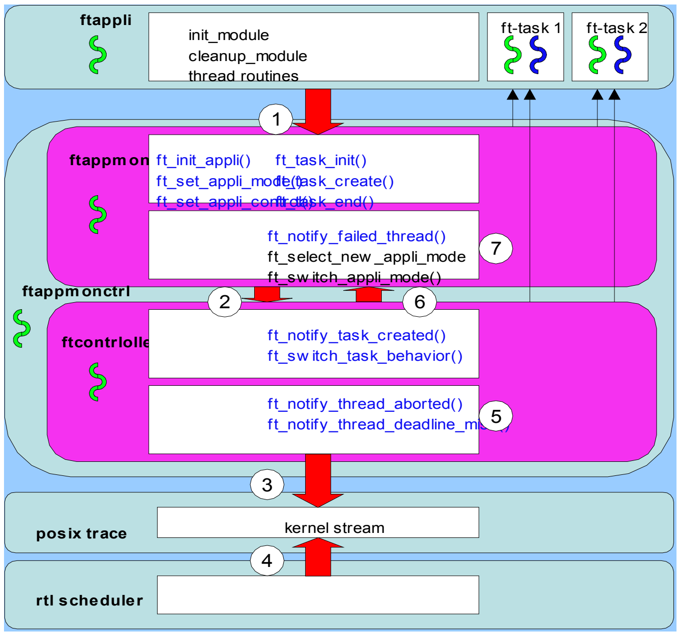


Figure 7.1.7 Overview of FT architecture and API

The FT core architecture is the **ftappmonctrl** upper-component which integrates both ftappmon and ftcontroller components. The ftappmonctrl uper-component constitutes a single RTLinux module.

The **ftappmon** component is designed for the monitoring of the FT application. It offer a **FT-API** to the application developer, to create the ft-tasks and to manage the application model for application mode transitions.

The **ftcontroller** component is designed for the control of the threads of the ft-tasks and for the detection and reaction to abnormal kernel events provided by the scheduler. The **ftcontroller** component uses the posix tracing facility offered by the OCERA **ptrace** component to survey the kernel events and react adequately. It is implemented as a periodic task at kernel level which activates the posix trace and reads kernel events related to threads in a kernel stream created by the **ptrace** component on activation of tracing facility.

The **scheduler** (rtl_sched) is an OCERA patched version of the RTLinux scheduler. It uses either standard priority (PRIO) or earliest-deadline-first (EDF+SRP) scheduling policy.

The **abnormal kernel events** actually managed by FT components are posix thread kill (PTHREAD_KILL) and deadline-miss (DEADLINE_MISS). In FT context, an error is the arrival of an abnormal kernel event on a normal or degraded thread of a ft_task. The processing of deadline-miss kernel event needs a particular OCERA patch on RTLinux kernel and the use of EDF scheduling policy.

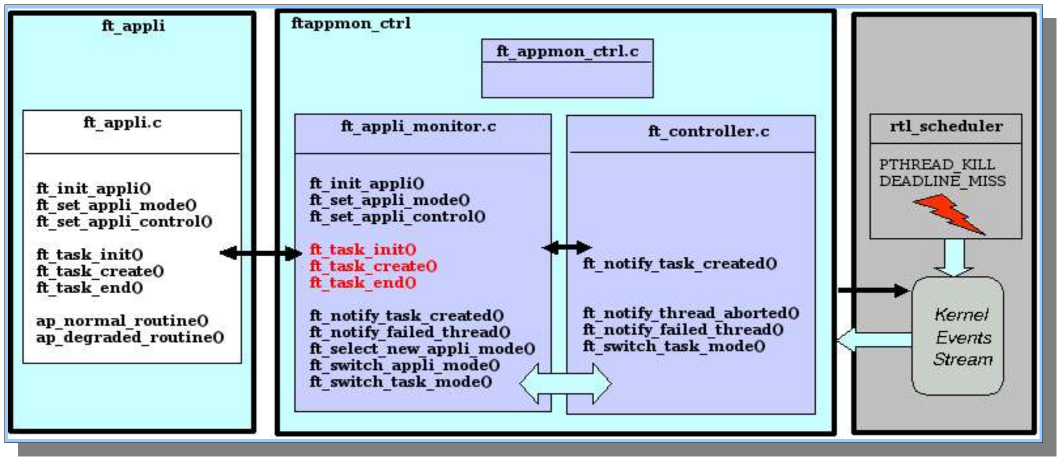
The use of FT components requires the selection of the following facilities when configuring the OCERA kernel: Hard+SoftRTLinux, the FT Application Monitor **ftappmon**, the FT Controller **ftcontroller**, the FT Builder **ftbuilder,** the posix trace **ptrace** and the **EDF** scheduling.

a) Functioning principle of the FT components

The **ftappli** application (see fig. 5), developed by application developer, calls external **FT-API** functions of **ftappmon** component at initialization (1), this induces the instantiation of the application model, the creation of the ft-tasks (so relative normal and degraded threads) and the notification of ft-task created to the **ftcontroller** (2). The scheduler produces some kernel events related to the threads (4) and the ftcontroller reads these events in the kernel stream (3). If the ftcontroller detects an abnormal kernel event relative to a thread of a ft-task, a notification of thread aborted is issued and a local behavior switch is performed on this ft-task only (5). A notification of failed thread is then issued from the **ftcontroller** towards the **ftappmon** (6). Then, if a new application mode is selected by ftappmon, an application mode switch can be activated (7). In this case, the **ftappmon** commands the **ftcontroller** to switch all the ft-tasks behaviors related to the new application mode (2).

7.1.6) FT Degraded Mode Management API overview

The API defined for the management of degraded modes is quite reduced. It is divided into external API defining primitives usable by the programmer and internal API used to communicate between ft_components. This is illustrated by the following figure.



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()**. In addition, it offers 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**.

7.2) Redundancy Management.

7.2.1) Introduction

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.

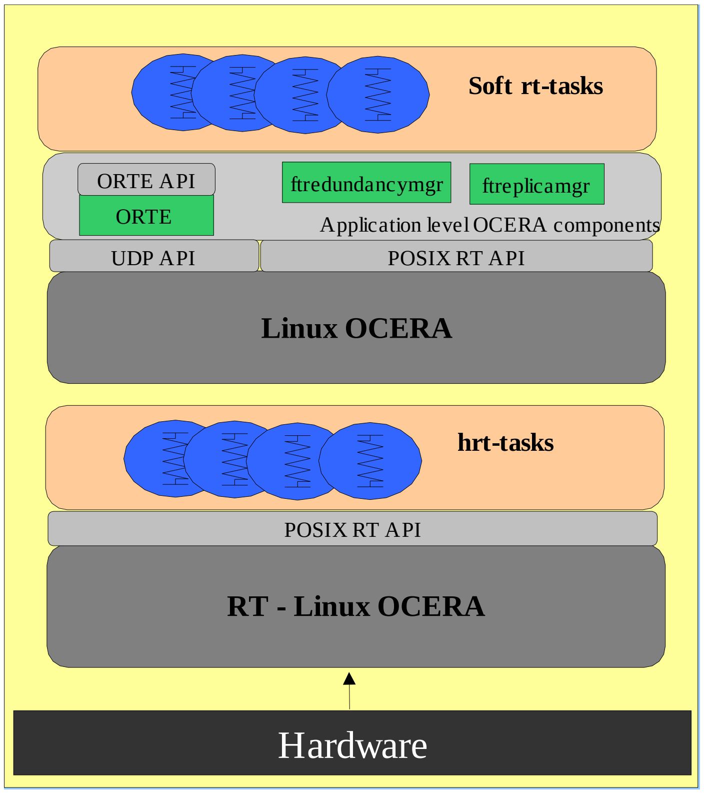


Figure 7.2.1 FT Redundancy management components location within OCERA architecture

Before entering in the details of the components architecture we describe briefly, the application and tasks model used, then we introduce the main principles of functioning of the overall architecture.

7.2.2) Application and tasks model

An application consists of a set of ftr_tasks (fault_tolerant redundant tasks).

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 the following task model.

a) ftr_tasks

A ftr_task is a real-time periodic task (same parameters as pthread scheduling parameters).

All ftr_tasks are periodic, there is no other temporal synchronization than the periodicity of the task. The basic cycle of a ftr_task instance is the following.

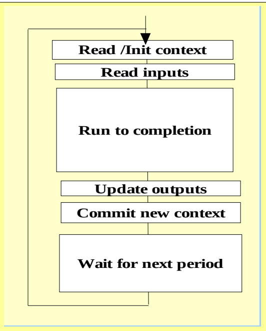


Figure 7.2.2 ftr_task execution model

A context object is defined for each ftr_task, this context contains static variables which are changed during a period and whose change is significant for the task behavior of next periods. This context is saved at each end of execution of an iteration. It is then broadcasted to the task group of replicas, so that one of them can become the new master task and start with a valid context in case of node crash.

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.

Communication with other tasks is limited to reading and writing data in predefined shared objects. Reading is done at beginning of the period, writing is done at the end of the period.

These objects have only one writer, the visibility of data is enabled to other tasks after the completion of the code of the task, at the beginning of the new period. (Which means that tasks are working on data obtained during the previous period of the writer's task). In the figure below, he green part can be accessed by any reader during current period.The red part is accessed only by owner and will be committed at the end of the period to become green. In case of error during the period a default value is issued as green value for next period.

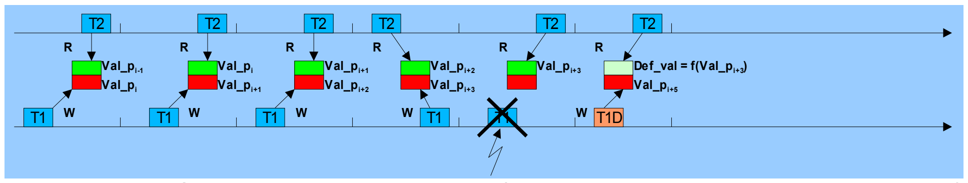


Figure 7.2.3 : Synchronized shared data. (observability is commited to the end of the period of writer).

When defining ftr_tasks, it is required to specify :

- the structure of its ftr_context object;
- the ftr_shared_data objects that will be used as inputs;
- the ftr_shared_data objects that will be written by the ftr_task (only one writer per ftr-shared_data object).

b) ftr_tasks_group

The redundancy management model adopted is a passive replicas management model.

Redundancy parameters have been introduced in the ftr_task data structure. These parameters include :

- the redundancy level required for the ftr_task (minimum and maximum number of replicas);
- the location of each replica;

From these information, and for each ftr_task, an ftr_tasks_group is defined which gathers data on :

- ftr_task_master location and status
- for each ftr_task_replica of the ftr_task_master its location and status

- current valid context of ftr_master_task (from last period)
- temporal information of ftr_task_master (date of cycle start, deadline, date of cycle end).

c) ftr_tasks group management main principles

Tasks group management is insured by the **ftreplicamgr** which periodically broadcasts the new context emitted by the ftr_master_tasks if execution cycle completed successfully.

Moreover, if the ftr_master_task is writer of a ftr_shared_data, the ftr_shared_data new value is also broadcasted to other nodes at each end of cycle.

Temporal behavior of ftr_master_task execution is controlled and notification of error is done to **ftredundancymgr** in case of deadline miss. If necessary a replica is elected as new master and the previous master is deactivated. The selection of the new master is deterministic, it is simply the ftr_task_replica located on the next available node (in an ordered list of nodes).

d) Simple example of redundancy management over two nodes

In the following simplified example, the application is composed of two ftr_tasks implemented on two nodes. The two master replicas for tasks T1 and T2 are located on Node1 and two slave replicas are located on Node 2. T1 and T2 periodically (at each end of cycle of each task) transmit their contexts (CT1 and CT2) to **ftreplicamgr** which broadcasts them to members of ftr_tasks_groups of T1 and T2 (in this simple case only to T1:s1 and to T2:s2). Moreover T2 is producer of ftr_shared_data SD1, so SD1 is also propagated to Node2.

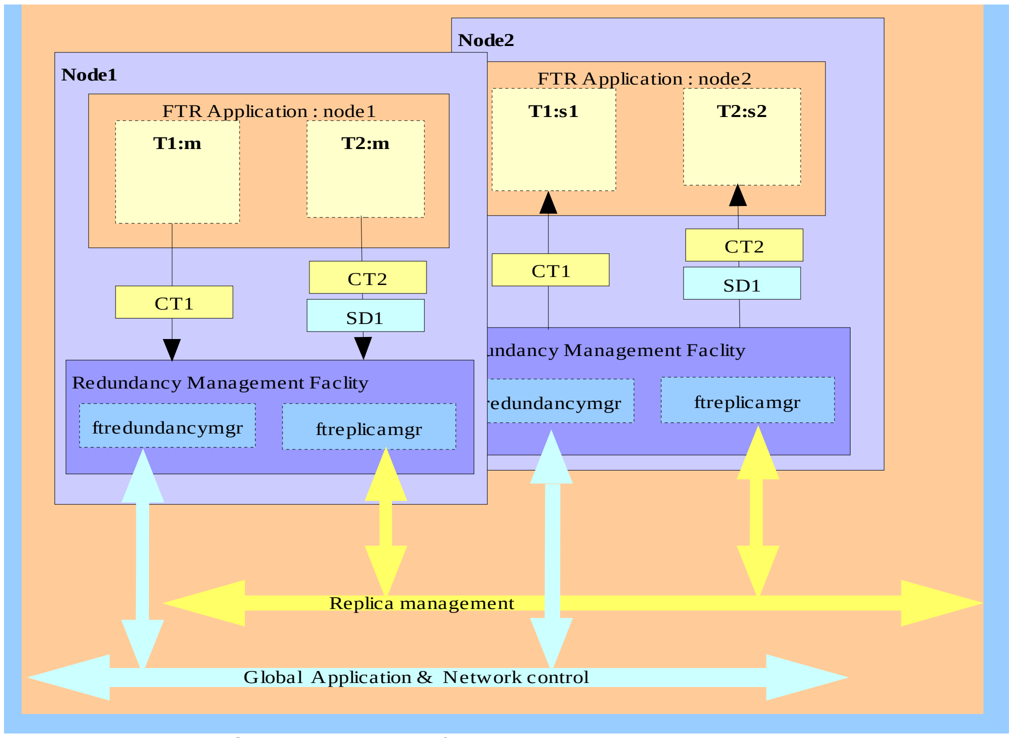


Figure 7.2.4 Simple example of redundancy management over two nodes

The **ftredundancymgr** controls global network, detects possible node crash and decide of dynamic reconfiguration when such an event happens. Information on application status is thus also replicated within each node. Such transmission of information is totally transparent to the user.

7.2.3) Faults management

a) Faults management at task level

The **ftreplicamgr** located on each node controls the execution of each master replica, namely: start-time, end-time, and timeliness of transmission of context.

If a deadline miss occurs on a ftr_task (the master did not transmit its context on time), a new master is elected amongst the corresponding ftr_tasks_group and the faulty one is terminated.

The **ftreplicamgr** notifies the **ftredundancymgr** of the fault, this latter then updates its new tasks configuration and broadcasts it to each node.

The **ftreplicamgr** located on the node where the new master task will from now on be located, switches the ftr_task_replica on and makes it run in master mode instead of slave mode. It will start at the next period (P+1) of the ftr_task with the last valid context (context P-1) . Several strategies can be envisaged to provide smoother behavior to the application, but for the moment only this rather drastic solution is implemented (one period is lost).

b) Faults management at node level

The **ftredundancymgr** of each node periodically sends a liveliness message to all other nodes, a node_failure_detection mechanism checks arrival of these messages.

A silent node is considered as faulty and retrieved from the set of available nodes. All active tasks on that node are switched off and a new replacement master task is elected for each one. The process of election is deterministic (using the ordered list of valid nodes). If it is not possible to find a new master task then the current default action is to end the overall application.

As said previously, this first implementation provides a global framework build on top of OCERA Soft RT level, all the application tasks are periodic tasks.

Though the implementation of these components was initially intended to be developed at both Hard and Soft realtime levels, the current version has been implemented using results of OCERA available at end of phase1 before full integration be ready. The implementation at hard RT level should however be easily ported to Hard real-time level when ORTE components at this level are available.

7.2.4) FT redundancy management architecture overview

As viewed in the previous section, the implementation of redundancy management requires two OCERA RTLinux components located at the Linux application level on each machine of the network.

- a Redundancy manager (**ftredundancymgr**) in charge of the global application monitoring and redundancy policy. This component is in charge of application initialization and control of overall distributed architecture. It also performs node crash detection through liveliness control using heartbeats. On detection of such failure, dynamic reconfiguration of application is activated. New master tasks are elected in order to replace tasks which were located on the faulty node. Low level control of execution of tasks is delegated to a replica manager which is in charge on insuring consistency of groups of redundant tasks (see below).
- a Replica manager (**ftreplicamgr**) in charge of the low level control of the tasks. Tasks groups are defined with a master and several slaves depending on the redundancy level required for the task. Tasks are all periodic tasks, only the master task of a group is active, at each end of cycle, checkpointing is performed. The new context of the task is then broadcasted to all replicas of the task. If a timeout is detected on a periodic task, a notification is issued to **ftredundancymgr** which will then test if corresponding node is still alive and decide about action to be undertaken. (Change current master task to an other one or use default action, or detect node crash and reconfigure all application).

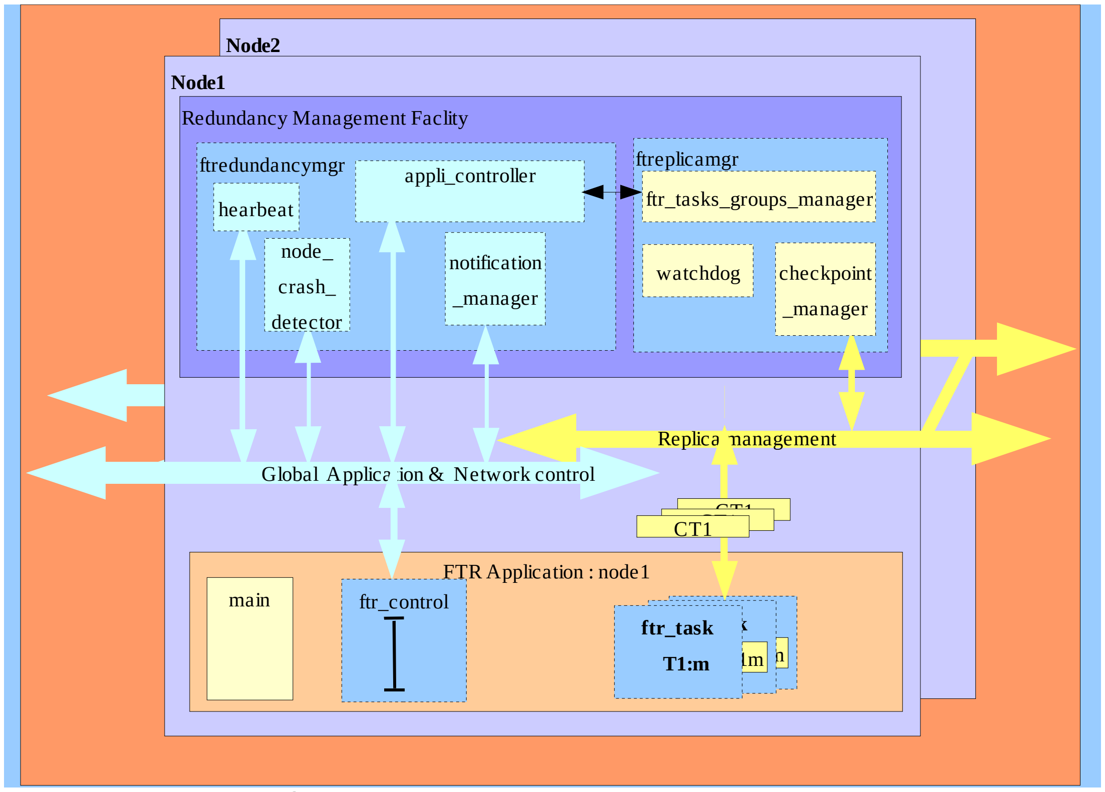


Figure 7.2.5 Global FT Redundancy Management Architecture overview

We give a rapid overview of this overall framework functioning principles in the following two sections.

a) Architecture of redundancy management facility on a node

On each node, the two components (**ftreplicamgr** and **ftredundancymgr**) are implemented as two separate threads that cooperate within a Linux process in user's space.

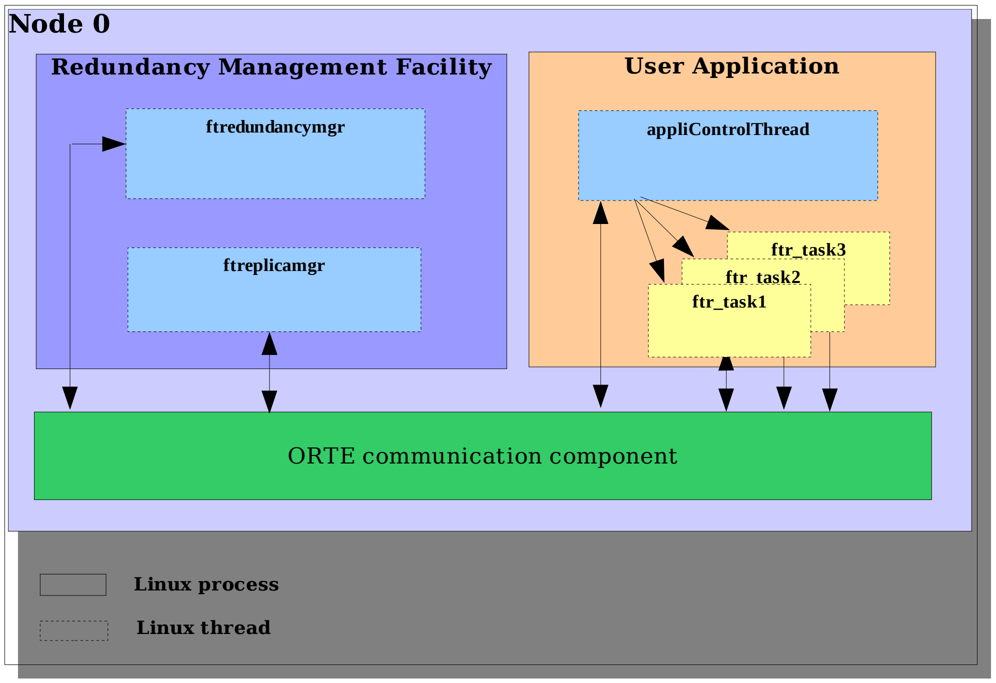


Figure 7.2.6 Overview of RedundancyManagement Architecture on a node.

The user application runs in a separate Linux process. In the current implementation, all application tasks are implemented as threads within one single application Linux process. (This choice has been made in order to be closer to the future implementation at Hard RTLinux level where all threads share the same space).

Within this process, an application control thread is created at application init, it is in charge of application tasks creation and communication with the ftredundancy management facility. This thread is transparent to the user which uses a dedicated API to create and run application tasks.

An application task is encapsulated within a ftr_task which insures periodic control of the task and checkpointing (communication with the **ftreplicamgr**).

Communication between processes and between nodes are handled by ORTE communication component.

There is one instance of each component on each node of the network. The redundancy manager on each machine has a complete knowledge of application current configuration and constraints, so that it can take decisions in an autonomous way if necessary.

The replica managers maintain a table of all ftr_tasks_groups and maintain the current status of each member of a group (master, passive_replica, unavailable_replica) and its location. Each active replica, periodically sends its new context which is then broadcasted to all other members of the group. The protocol must insure reliable atomic transfer to all members of the group of replicas. The replica manager, regularly verifies that context checkpoint has been performed on time.

b) Basic interaction between components

The main interactions between components are illustrated in following figure where the two instances of components located on a node appear as two separate threads within one Linux process, services offered are shown within oval forms. This framework is present on each node of the distributed architecture required for the application.

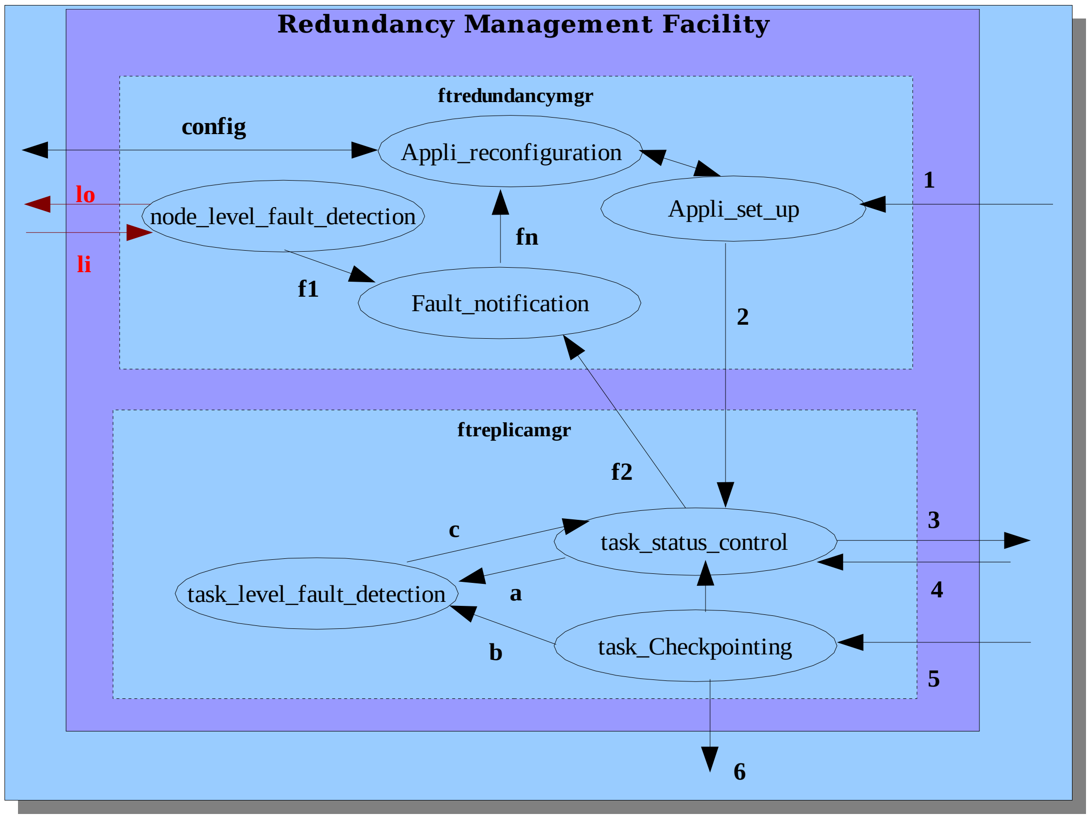


Figure 7.2.7 . Internal and external interactions of ft_redundancy components

Main functioning loop:

1 : application init

2 : tasks groups creation

3 : task creation, start or end

4 : task cycle started

5 : task cycle ended + context or task end

6 : checkpoint towards passive replicas

Node level lifeliness control :

li : lifeliness in (node_i is alive) received from each node

lo : current node is alive sent

f1 : node i not responding

Task level fault detection :

a : task started + deadline

b : task ended

c : timeout reached task not responding

f2 : task j on node n not responding

In this figure three types of protocols are shown, the first one concerns the main functioning loop, the second concerns node crash detection and related reconfiguration process, the third one concerns timeout detection at task level, the task may be faulty but not the node.

c) Application life-cycle

Previously to application start, the redundancy management must be made available. A script shell permits the installation of the two components onto a set a specified nodes. Once **ftredundancymgr** and **ftreplicamgr** are installed and ready, one can start an application.

At init, application is started in master mode on one node which is called node_0, information on application configuration is provided to the **ftredundancymgr:**

- number of tasks and tasks descriptions (including redundancy level parameter for each task),
- number of nodes and nodes_Id
- initial mapping of tasks onto nodes
- ordered list of nodes for dynamic reconfiguration on node crash (determines node replacement choice)

According to this information, the **ftredundancymgr instantiates** its internal application description table and remotely starts application on other nodes in slave mode.

It then builds tasks groups (master task + its replicas) and sends information for each group to the **ftreplicamgr** whose role is two control the functioning of each task and to maintain the consistency of all replicas for each tasks group.

The **ftreplicamgr** instantiates its own tasks groups table and task control status table, then enables creation of tasks and tasks replicas on each node (actually task creation itself is achieved within the User Application process by the internal application control thread, the task may be created as active or passive).

When all tasks are created on each relevant node, start of tasks is enabled. Each task thread becomes ready and is then started according to its realtime parameters and to scheduling policy. At each cycle, start of cycle is notified to **ftreplicamgr**; at each end of cycle, end is notified and the checkpointing of the new task context is achieved. A watchdog verifies timeliness of task completion and a fault notification is issued in case of deadline miss on a task.

On application termination all tasks are terminated, then application instances are finished on each node and application is deregistered. The redundancy management facilities stay available for a new application or may be ended by a specific command.

7.2.5) Overview of Redundancy Management API

The Fault-Tolerance components described in this document have to be used jointly since they interfere strongly. It is the reason why though each one has its own API described in a distinct section, it may be useful to get a general overview of them.

The external user API is actually restricted to very few functions :

- **ftr_application_register()**,
- **ftr_appli_desc_init(),**
- **ftr_appli_task_create()**,
- **ftr_appli_task_end(),**
- **ftr_application_terminate()**

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.

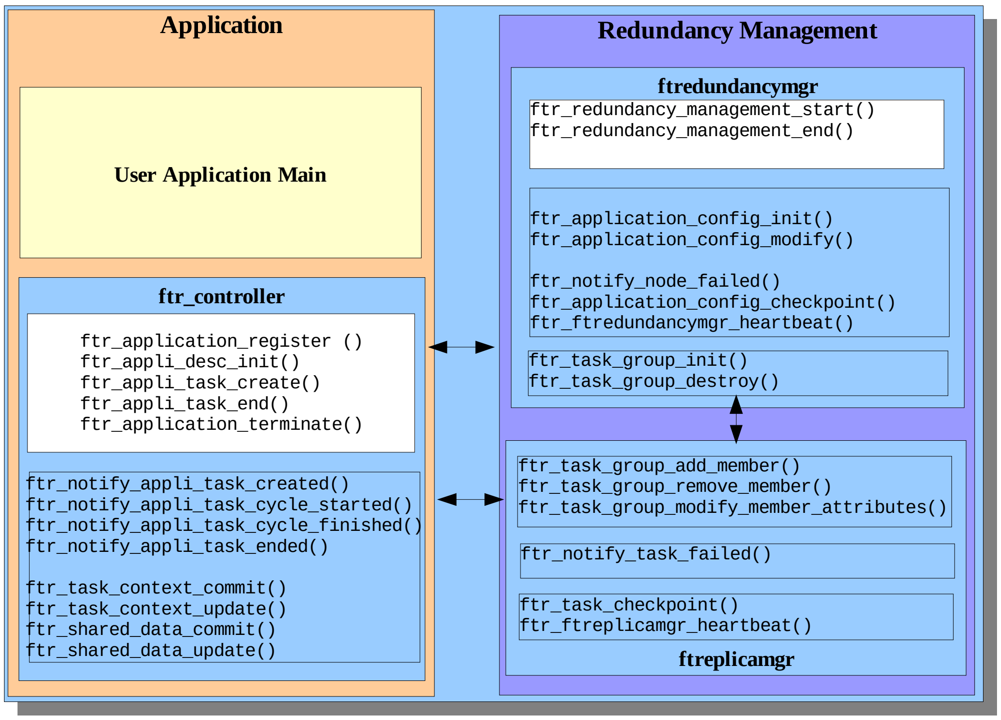

Application
User Application Main
ftr_controller
ftr_application_register ()
ftr_appli_desc_init()
ftr_appli_task_create()
ftr_appli_task_end()
ftr_application_terminate()
ftr_notify_appli_task_created()
ftr_notify_appli_task_cycle_started()
ftr_notify_appli_task_cycle_finished()
ftr_notify_appli_task_ended()
ftr_task_context_commit()
ftr_task_context_update()
ftr_shared_data_commit()
ftr_shared_data_update()
Redundancy Management
ftredundancymgr
ftr_redundancy_management_start()
ftr_redundancy_management_end()
ftr_application_config_init()
ftr_application_config_modify()
ftr_notify_node_failed()
ftr_application_config_checkpoint()
ftr_ftredundancymgr_heartbeat()
ftr_task_group_init()
ftr_task_group_destroy()
ftr_task_group_add_member()
ftr_task_group_remove_member()
ftr_task_group_modify_member_attributes()
ftr_notify_task_failed()
ftr_task_checkpoint()
ftr_ftreplicamgr_heartbeat()
ftreplicamgr

Figure 7.2.8 Global view of ft redundancy management API(s)

The user's API is described hereunder while next chapters describe in details the two FT components.

7.2.6) How to program redundant applications

a) Implementation issues and impact on user programming

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.

b) User's 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.

```
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

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.

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.

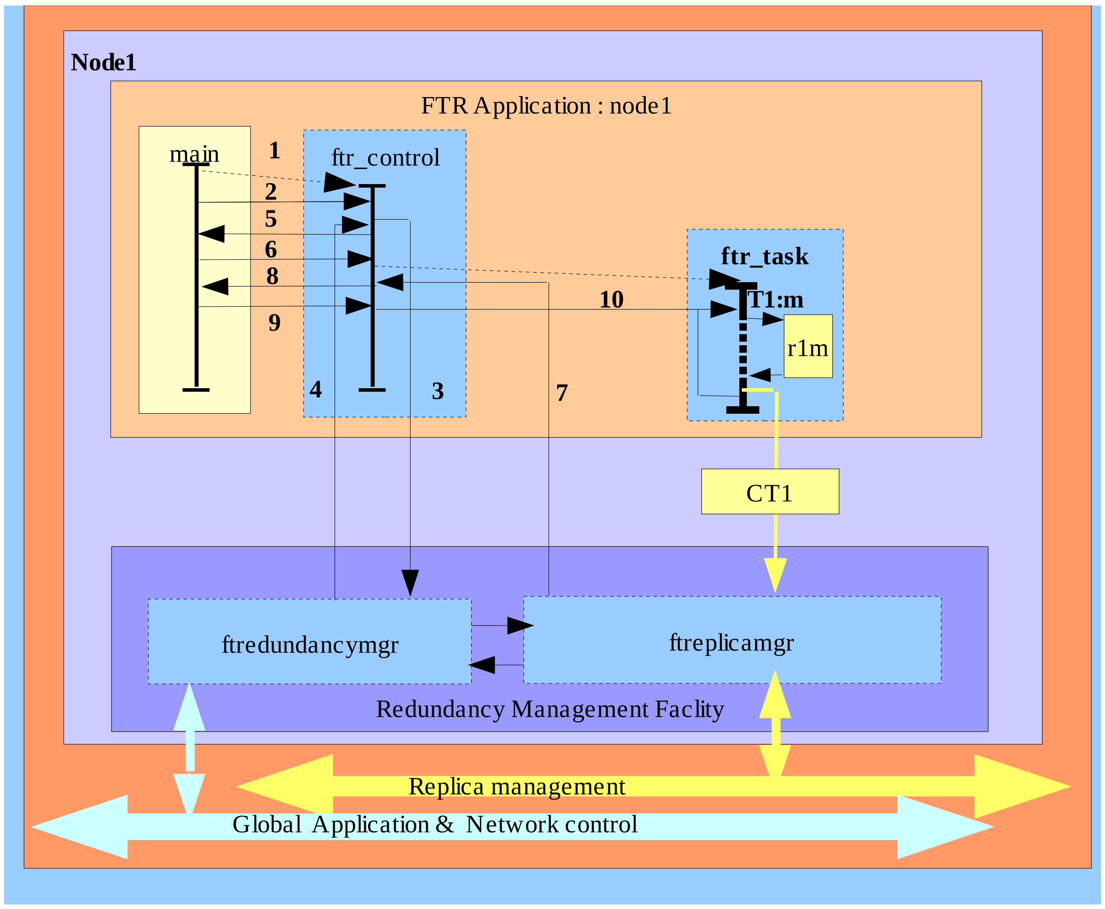


Figure 7.2.9 Interactions with redundancy Management facility components form User's Application

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.

c) 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.

7.2.7) 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.

7.2.8) Results and comments

The current implementation is still a prototype one. 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.

PART III

Building the Development Environment

8) Development Environment

By
Pierre Morel – MNIS

8.1) The tools you need

To build the application you will run on your target you will need different kind of tools, depending on the kind of application you want to build.
First of all you will need a Linux system, because OCERA and all its tools are build using Linux.
The OCERA system consist of the RTLinux-GPL real time kernel handling hard real time and the Linux kernel for soft real time and time sharing applications.
You may want to build three kind of applications:

- Hard real time applications
- Soft real time applications on Linux
- Time Sharing applications

It is obvious that you need one of the realtime applications, hard or soft if you use OCERA and you also would like to add some time sharing application for data presentation, debug, login or other non real time work.

The table here under present the tools you need depending on your target system assuming you develop on the same kind of plateform as your target.

	Hard RT	***Soft RT***	***Time Sharing***	***Embedded***
Compiler	Gcc-2.95			
Headers	RTLinux and Linux	QOS	Libc	uClibC
Libraries	RTLinux-GPL and OCERA	QOS	libc	uClibC
Basic utilities	Busybox or Linux			Busybox
Booting	Standard Linux boot utilities: lilo, grub, syslinux			u-boot, redboot, grub, etherboot
Other utilities	mke2fs			Jffs2, romfs

To this basic set, you will need to add specific libraries to build the tools, like Qt libraries to use graphical configuration for the Kernel and OCERA components, ncurses to use menu configuration for the kernel, busybox and uClibc.
All these utilities are open source software and can be found easlily on INTERNET.
We provide four ways to build an OCERA system:

- Using a CDROM distributed by one of the members of the OCERA consortium
- Using DEBIAN apt-get utility and the http://www.ocera.net/ apt server.
- Using one of the supported linux distribution and the OCERA tarball you van download from the OCERA project page on sourceforge: http://ocera.sourceforge.net/
- Using the OCERA CVS, hosted on sourceforge.

The prefer way to install OCERA is with apt-get on a Debian distribution, because you will allways get the last stable release doing so.

8.2) Installing with apt-get

8.2.1) OCERA development environment

If you allready have an installed Debian 3.0 LINUX distribution on your computer and an INTERNET access, you may also simply add a new source to your **apt sources.list** file (/etc/apt/source.list).

```
deb http://ocera.sourceforge.net/debian woody contrib
```

and issue the command:

```
# apt-get update
# apt-get install ocera-dev
```

You should the begin to install the ocera development environment DEBIAN packages and the eventual dependencies packages.
All the source files will be installed under the directory /usr/share/ocera_X.Y where X and Y are respectively the major and minor release number of ocera-dev package.

8.2.2) Installing independent components as binaries

You may also install some of the independent OCERA components as binaries or sources.
Actually there is two independent components: ORTE and LINCAN.
You can get ORTE as a development package or as a binary package you can also get ethereal enhancement for ORTE as a binary package, while LINCAN, being a Linux driver is only available as a development package.
You can download these components from the OCERA SourceForge site or install them as packages.
Assuming you already changed your apt sources.list file like we described in the previous chapter, you will issue:

```
# apt-get install orte
# apt-get install ortereal
```
or
```
# apt-get install lincan-dev
```

8.2.3) The documentation

You can install the documentation the same way:
```
apt-get install ocera-doc
```
And you will get the documentation installed in the appropriate tree:
- The documentation in /usr/share/doc
- The man pages in /usr/share/man.

8.3) Installing from a CD-ROM

8.3.1) Getting a CDROM

You can get a CD-ROM from one of the OCERA partners, you will get the list from the OCERA web site.
Another possibility is to download an ISO image from the OCERA web site.
This is certainly the best way to start since you will get a complete and tested environment for the development of your embedded system.

8.3.2) Installing the CD-ROM

To install the CD-ROM you simply need to boot on the CD-ROM and install the software as you would do for a normal DEBIAN distribution: just follow the instructions at the screen.

8.4) Supported development environment

The OCERA consortium choosed to base the developments on the Debian 3.0 Linux distribution.
The OCERA software should also compile on any Linux distribution as long as you use the following tools:

Tool	***Version***
C Compiler [2]	GCC-2.95.4
	GCC-3.34
Graphic environment	Qt-3.1.2
OCERA qconf development	libqt3-dev

2 Why two gcc compilers? Because the Qtlibraries used by Qconf will only compile with gcc-3.3 or newer, while gcc-2.95 is needed to compile the kernel as expected by Linux Torwald (see the main kernel README.txt).

Tool	***Version***
Make	Make_3.80
Autoconf	Autoconf_2.59a
Automake	Automake_1.4p4
Ada Compiler	Gnat-3.15
Libc6-dev	>=2.3.2
Busybox	>=1.0
Syslinux	>=2.11
Ncurses	>=5.4.4
Bison	Bison_1.875d
Flex	Flex_2.5.31
Libstdc++	Libstd++6_3.4.3
G++	G++_3.35

Never less we encourage you to use a standard Debian 3.0 and the appropriate packages and to download the OCERA software from the OCERA internet site or to use a ready to install OCERA development CD-ROM.

9) Cross compilation

By
Petr Cvachoucek – Unicontrol (PowerPC)
Cristina Sandoval- VisualTools (Arm)

9.1) Building a cross compiler for Linux/PowerPC platform

9.1.1) Introduction

The process of building a cross compiler targeted to Linux on various processor architectures isn't simple. Basically you're required to do these steps:

1. Compile and install binutils for your target.
2. Configure kernel to generate headers and copy arch-specific headers to prefix dir.
3. Compile gcc for target (first pass)
4. Cross-compile glibc with above tools and install.
5. Compile final gcc for target (second pass).

Doing these steps by hand is a painful process, you'll run into many troubles (many patches needed). Fortunately there is a tool widely used by developers which automates this process.
The tool is maintained by Dan Kegel and can be downloaded from http://kegel.com/crosstool/ It's highly recommended to use this tool instead of doing the things by hand.

9.1.2) Prerequisities

To successfully build the toolchain, you need a host computer with a fast connection to internet (http and ftp connectivity).
The preffered development host platform is Linux, but the toolchain can be built also on Windows (Cygwin 1.5.12-1 or newer required).

9.1.3) Downloading the crosstool package

The crosstool package can be downloaded from site http://kegel.com/crosstool.
Download, save and unpack this file:
http://kegel.com/crosstool/crosstool-0.28-rc37.tar.gz
If you look at whats inside the package, you'll find a set of scripts, configuration files and patches.

9.1.4) Choosing the toolchain versions

Another important step is to choose versions of toolchain components.
We will describe the process of building of following toolchains:

Gcc	***Glibc***	***Binutils***	***Kernel headerrs***
2.95.3	2.2.5	2.15	2.6.8
3.4.2	2.2.5	2.15	2.6.8

9.1.5) Creating the configuration file for PowerPC 603e CPU

In the package is already several configuration files for various types of PowerPC CPUs. As our development board is equipped with Motorola 8240 CPU (603e core) we need to prepare a configuration file for it.
The file will be named powerpc-603e.dat and will have this content:

```
TARGET=powerpc-603e-linux-gnu
TARGET_CFLAGS="-O -mcpu=603e"
GCC_EXTRA_CONFIG="--with-cpu=603e --enable-cxx-flags=-mcpu=603e"
```

The file must be placed at crosstool package directory.

9.1.6) Creating the script to build toolchains

Before invocation of a crosstool script we need to setup several environment variables, which controls the script. So we create a small script just for the purpose of setting these variables and then executing the crosstool script.
The file will be named build-ppc603e.sh and will have this content:

```
#!/bin/sh
set -ex
TARBALLS_DIR=./tarballs
RESULT_TOP=/opt/crosstool
export TARBALLS_DIR RESULT_TOP
GCC_LANGUAGES="c,c++"
export GCC_LANGUAGES

# Really, you should do the mkdir before running this,
# and chown /opt/crosstool to yourself so you don't need to run as root.
mkdir -p $RESULT_TOP

# Build the toolchain.  Takes a couple hours and a couple gigabytes.
eval `cat powerpc-603e.dat gcc-2.95.3-glibc-2.2.5.dat` sh all.sh --notest
eval `cat powerpc-603e.dat gcc-3.4.2-glibc-2.3.3.dat` sh all.sh --notest

echo Done.
```

The file must be placed at crosstool package directory.

9.1.7) Building the toolchains

Everything is prepared, we can start building the toolchain.
Invoke the script by command:

```
#  sh build-ppc603e.sh
```

9.1.8) Build results

When the toolchain build completes, you'll find results in the directories:

Toolchain gcc/glibc	***path***
2.95.3 / 2.2.5	/opt/crosstool/powerpc-603e-linux-gnu/gcc-2.95.3-glibc-2.2.5
3.4.2 / 2.3.3	/opt/crosstool/powerpc-603e-linux-gnu/gcc-3.4.2-glibc-2.3.3

Tarballs downloaded during the build of toolchains are stored in ./tarballs dir.

9.2) Cross Compilation for ARM/iPAQ

9.2.1) ARM Processor

Because the hardware on ARM/iPAQ is different from most desktops, a cross-compiler capable of producing code for different platform is usually needed. Toolchains contain the neccessary tools to cross-compile code for that target machine. The easiest solution is to use GCC, which is open-source and can be obtained for free.

a) Cross-Compiling for the iPAQ

You can compile applications for the iPAQ by using an x86 Linux machine and cross-compiling.
The cross toolchain works just like the standard native compiler, except that each of the tools is prefixed with "arm-linux-". For many programs, the cross-compiler can be invoked by running make as follows:

```
make CC=arm-linux-gcc all
```

If the make process calls other target-specific tools, then these also need to be specified:

```
arm-linux-ld
arm-linux-ar
arm-linux-ranlib
arm-linux-strip
arm-linux-g++
arm-linux-as
```

9.2.2) Where to find a pre-built toolchain

You can download the toolchain from handhelds site:

ftp://ftp.handhelds.org/pub/linux/arm/toolchain/
The sources corresponding to this toolchain are in
ftp://ftp.handhelds.org/pub/linux/arm/toolchain/source/

9.2.3) Porting Software to ARM Linux

Many times the software you would like to run on the iPAQ is written in C. C is not an inherently portable language. To write portable code in C generally requires some extra thought.
There are some portability issues that may need some special attention when we run into when porting applications to ARM Linux, especially from x86 Linux.

a) C Portability Issues

There are a number of areas in which the definition of a C program's behavior depend on the architecture on which the program is run. It's behavior can depend on the peculiarities of the OS, the compiler, the libraries, and the CPU.

b) Signed vs. Unsigned Characters

The C standard says that char may either be signed or unsigned by default. On x86 Linux, char is signed by default. On ARM Linux, char is unsigned by default. Comparing a char to a negative number will always return 0, because the char is unsigned and therefore positive.

c) Pointer Alignment Issues

On many CPU architectures, the memory system requires that loads of values larger than one byte must be properly aligned. Usually, this means that a 2-byte quantity must be aligned on an even address boundary, a 4-byte quantity must be aliged on a multiple of 4 boundary and sometimes 8-byte quantities must be aligned to addresses that are a multiple of 8. Depending on the CPU and the operating system, misaligned loads and stores may cause a signal, may be handled in the OS, or may be silently rounded to the appropriate boundary.
The x86 boundary imposes no such alignment restriction, so some programs written for the x86 do not use the proper alignment for other architectures.
ARM Linux defaults to silently round the address to the appropriate alignment boundary.

d) Using Memory Overlays to Convert Types

This is very non-portable. The code has to be written so that alignment, size, and endianness are all correctly handled across the supported architectures.

e) Endianness Issues

There are two basic memory layouts used by most computers, designated *big endian* and *little endian*. On big endian machines, the most significant byte of an object in memory is stored at the least signicant (closest to zero) address (assuming pointers are unsigned). Conversely, on little endian machines. the least significant byte is stored at the address closest to zero. Let's look at an example:

```
int x = 0xaabbccdd;
unsigned char b = *(unsigned char *)&x;
```

On a big endian machine, b would receive the most significant byte of x, 0xaa. On little endian machines, b would receive the least signficant byte of x: 0xdd. The x86 architecture is little endian. Many ARM processors support either mode, but usually are used in little endian mode.
Endian problems arise under two conditions:

- When sharing binary data between machines of different endianness.
- When casting pointers between types of different sizes

In the first case, the data appears in the correct location, but will be interpreted differently by the different machines. If a little endian machine stored 0xaabbccdd into a location, a big endian machine would read it as 0xddccbbaa.
In the second case, on a little endian machine there is no problem: a char, short, or int stored in an int sized variable each have the same address. On a big endian machine, if you want to be able to store a short and then read it as an int you have to increment the pointer so that the MSB lands in the right place.

10) Configuration

By
Pierre Morel - MNIS
Agnes Lanusse - CEA (Fault Tolerance)
Patrick Vanuxem - CEA (Fault Tolerance)

In this chapter you will see how to build an embedded system and how to build a training system.
A training system is a system where you install the OCERA component on the same computer as you development system. This is useful for training.
For both installation you will need to define the components you need, adjust the kernel parameters and compile the kernel, the libraries and the tools.
Then you will need to add your custom application.
The last step will be to launch the application and kernel, this is the only step that is different between the two installation so we will design this chapter as:

6. Choosing the components
   At this stage we will not go deep in the description of each component but we will have a good overview of them.
   For a deeper description of each component, you will need to go to the component dedicated section later in this guide.
7. Configuration of components, kernel and libraries
   There ou will see how to compile the kernel, we focus on the configuration tool.
8. Building the tools
   The tools are independent of the kernel and are compiled separately, like debugger, tracing tools, analysers.
9. Compiling a custom application
   We will see some little applications, exemples and the compilation's directives.
10. Installing a training systems
    This is certainly the first way yo will install OCERA.

11. Installing an embedded systems
    This part is really dedicated for embedded systems, we will see how to make a complete embedded Linux with real-time enabled.

10.1) Choosing the components

At this stage we will not go deep in the description of each component but we will have an overview of the components.
For a deeper description of each component, you will need to go to the component dedicated section later in this guide.

10.1.1) POSIX components and scheduling

What we call the POSIX components are the RTLinux-GPL enhancement or the new real-time libraries OCERA added to RTLinux-GPL.
This is the basic brick to build a real-time system and are needed by most of the other components.
They provide:

- POSIX threads
- POSIX io
- POSIX signals
- POSIX messages

The scheduling components are enhancement of the basic RTLinux scheduler.

10.1.2) Core features

These features are enhancement of the core Linux and are needed by some of the OCERA components. They are found as patches over the INTERNET, we had to modify some of them to integrate them together and together with RTLinux and OCERA components.
The core features are:

- bigphys area patch
  provides the ability to use a big physical area for the heap of the Linux Kernel. This is mandatory for the OCERA memory allocator.
- Low latency patch
  makes modifications in the kernel to provide interrupt points where the latency is otherwise to high.
- Preempt patch
  makes modification of the Linux scheduler to allow task preemption.

10.1.3) Quality of services

You will need the Quality Of Service component if you intend to develop an application under LINUX needing to have a Constant CPU Bandwidth allocated.
The best exemple may be an video application for witch you do not want the system to steal time.

10.1.4) Fault Tolerance

Using the fault tolerance component implies that your developments follows some rules. It is quite sure that you will need a good knowledge of what the fault tolerance can bring to you before enabling this option. So please refer to the appropriate chapter later in this guide.

10.1.5) Onetd

The OCERA Network Daemon is needed if your realtime application wants to access the network. You do not need it if the network access is made by a Linux thread.

10.1.6) ORTE

The OCERA Real Time Ethernet may be used as is an independent component under any Linux or even other systems or may be used inside the real time kernel, in which case it will use most of the POSIX components.

10.1.7) Can devices

Can devices may be used as is an independent component under any Linux or even other systems or may be used inside the real time kernel, in which case it will use most of the POSIX components.

10.2) Configuration of components, kernel and libraries

We begin in this chapter with the real work. You will have to effectively choose the components for your application.
The first thing to do is to go into the mai directory of the OCERA tree, it should be **/usr/share/ocera_1.1** if you have installed your system with an OCERA ISO image or in the sub-directory **ocera_1.1** if you downloaded the tarball of the version 1.1 of OCERA.
Then, assuming you have a graphic environment, generate the configuration file with:

```
make xconfig
```

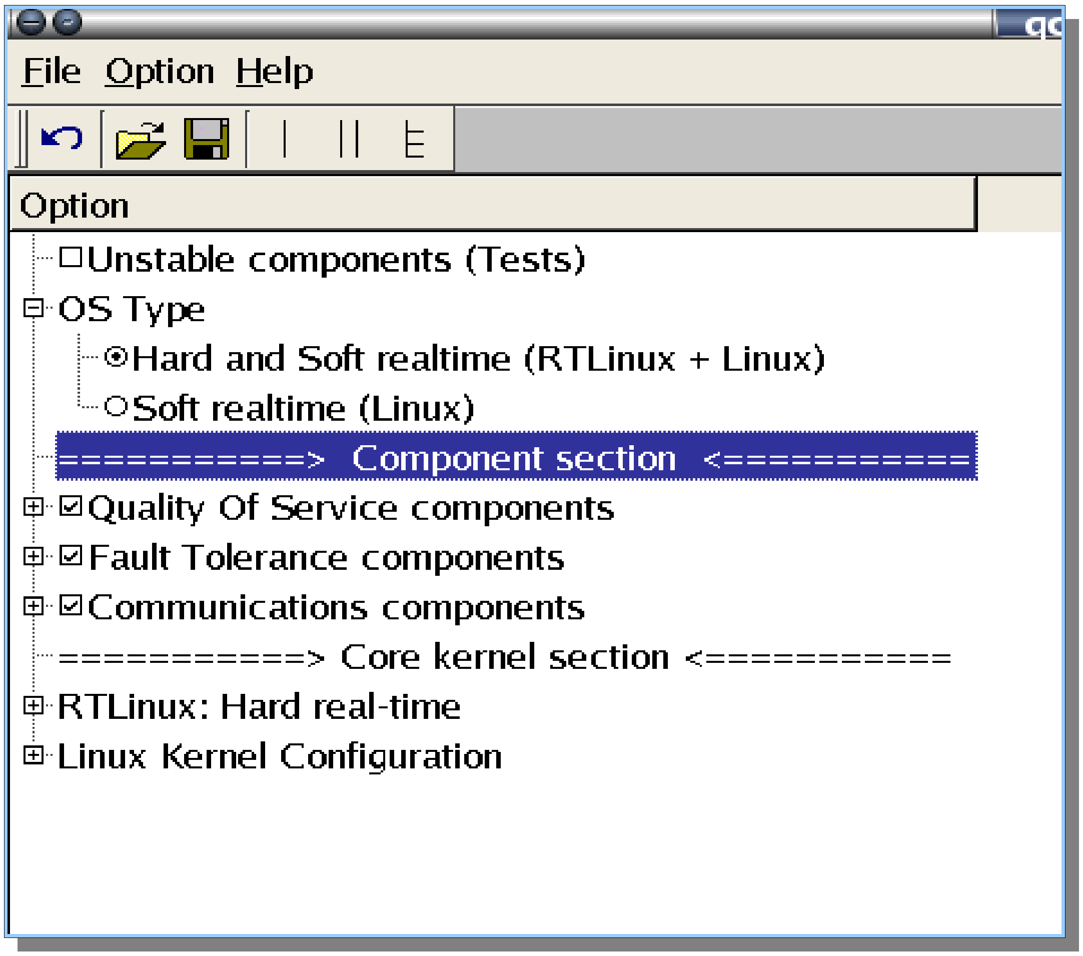
If you do not have a graphic environment, you can use the semi-graphic configuration menu with:

```
make menuconfig
```

or even the bare text configuration menu with

```
make config
```

We will assume that you have a graphic interface.
The system should compile the Qt tools needed for the configuration environment and you must get the following window on your desktop.



☐ Unstable components (Tests)
OS Type
- ◉ Hard and Soft realtime (RTLinux + Linux)
- ○ Soft realtime (Linux)

===========> Component section <===========

☑ Quality Of Service components
☑ Fault Tolerance components
☑ Communications components

===========> Core kernel section <===========

RTLinux: Hard real-time
Linux Kernel Configuration

Illustration 2General Configuration

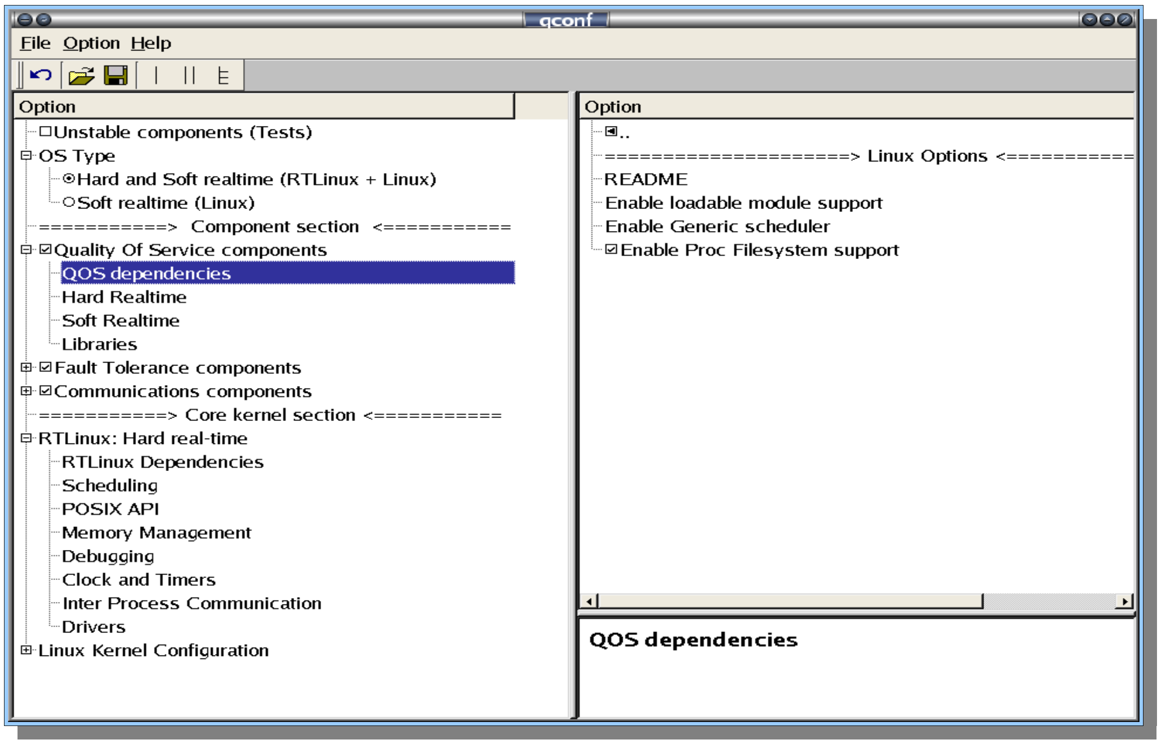
The first thing to see is that you have the following choices:

- Unstable components, enabling this will allow you to choose the components considered for now as unstable. In a major release you will not have any unstable components and in minor release you must remember that this really enable unstable components for debugging purpose only, do not expect the system to work correctly with the unstable switch activated since they indeed are unstable.
- OS Type, where you will choose if you want soft real time with Linux only kernel and a typical minimum latency around 10ms or Hard real-time with a typical latency around 10µs
- A component section with the three major components types; Quality Of Service, Fault Tolerance and Communication
- RTLinux with the RTLinux-GPL specific options and the OCERA real-time and POSIX extensions.
- and a Linux section, where you will find all the choice you are familiar with if you already compiled Linux 2.4.18 kernel.

Now let see the different possibilities you have if you open the Components menu:

10.2.1) Quality Of Service

a) Dependencies



```
☐Unstable components (Tests)
⊟ OS Type
   ⦿Hard and Soft realtime (RTLinux + Linux)
   ○Soft realtime (Linux)
===========>  Component section  <===========
⊟ ☑Quality Of Service components
   QOS dependencies
   Hard Realtime
   Soft Realtime
   Libraries
⊞ ☑Fault Tolerance components
⊞ ☑Communications components
===========> Core kernel section <===========
⊟ RTLinux: Hard real-time
   RTLinux Dependencies
   Scheduling
   POSIX API
   Memory Management
   Debugging
   Clock and Timers
   Inter Process Communication
   Drivers
⊞ Linux Kernel Configuration
```

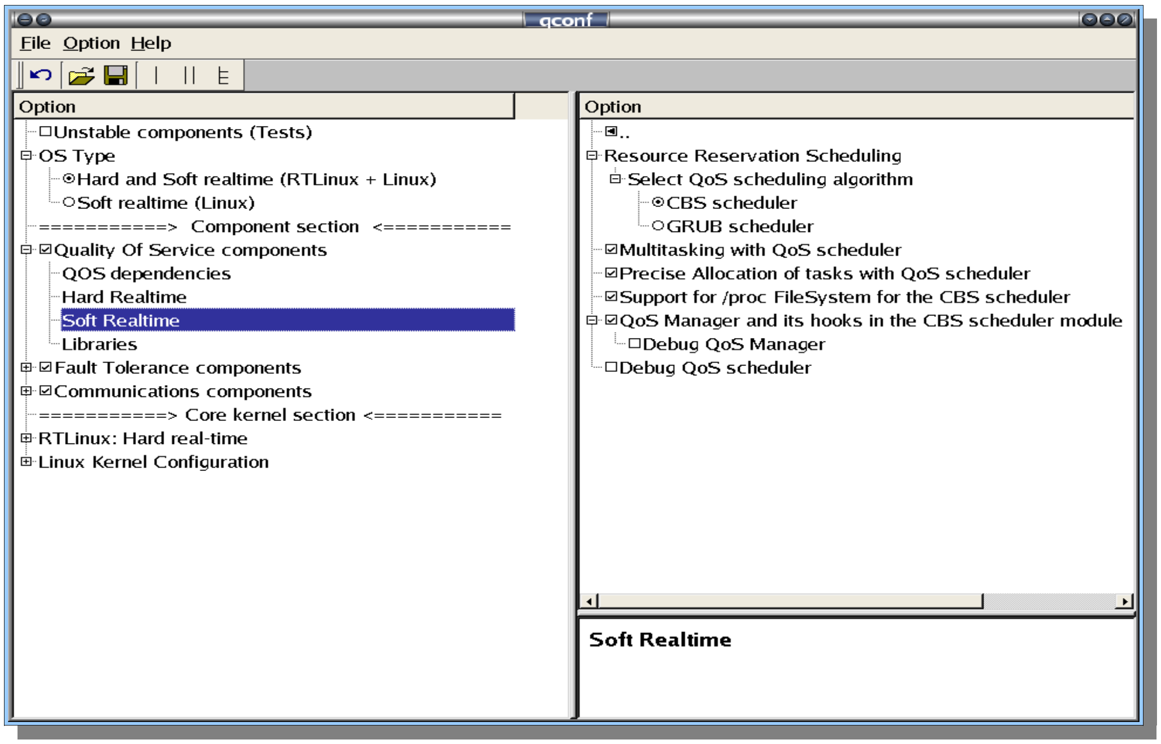
```
..
=====================> Linux Options <===========
README
Enable loadable module support
Enable Generic scheduler
☑Enable Proc Filesystem support
```

QOS dependencies

The first sub-menu in Quality Of Service is the dependencies menu, you will find such a menu at the first place in all components menu, as the name let think, this point out the dependencies that must be resolved to be able to compile the Quality Of Service components.
The Quality Of Service component needs "*module support*", "*Generic Scheduler"* and if you want sysctl support the "*Proc Filesystem support"*.
The Quality Of Service component does not provide Quality Of Service to Hard-Real time applications

b) Soft real-time

In this sub-menu, you will configure the functionalities you will use to enable Quality Of Service in the Soft-Real-Time, i.e. Linux, environment.



```
Option
  ☐Unstable components (Tests)
⊟ OS Type
    ◉Hard and Soft realtime (RTLinux + Linux)
    ○Soft realtime (Linux)
  ===========>  Component section  <===========
⊟ ☑Quality Of Service components
    QOS dependencies
    Hard Realtime
    Soft Realtime
    Libraries
⊞ ☑Fault Tolerance components
⊞ ☑Communications components
  ===========> Core kernel section <===========
⊞ RTLinux: Hard real-time
⊞ Linux Kernel Configuration
```

```
Option
  ..
⊟ Resource Reservation Scheduling
  ⊟ Select QoS scheduling algorithm
      ◉CBS scheduler
      ○GRUB scheduler
  ☑Multitasking with QoS scheduler
  ☑Precise Allocation of tasks with QoS scheduler
  ☑Support for /proc FileSystem for the CBS scheduler
⊟ ☑QoS Manager and its hooks in the CBS scheduler module
    ☐Debug QoS Manager
  ☐Debug QoS scheduler
```

Soft Realtime

Ressource Reservation Scheduling

QoS aware kernel module schedulers. By now this module supports X86, PPC and some ARM processor architectures.
You can choose between:

- CBS algorithm kernel module scheduler
- GRUB algorithm kernel module scheduler

Multitasking with QoS Scheduler

Gives a bandwidth of 10% to Linux tasks. In this way these tasks can execute during the execution of the tasks scheduled by QoS scheduler. In addition this option lets assign an amount of bandwidth to a set of tasks

Precise allocation of task with QoS scheduler

Any task scheduled by QoS scheduler executes exactly as specified by its bandwidth (even if there is only one QoS task on the system).
This let out-range some known problems associated with the QoS specific algorithm.

Support for the /proc filesystem

Enable this to allow monitoring and managing through /proc FileSystems entries of the CBS Scheduler

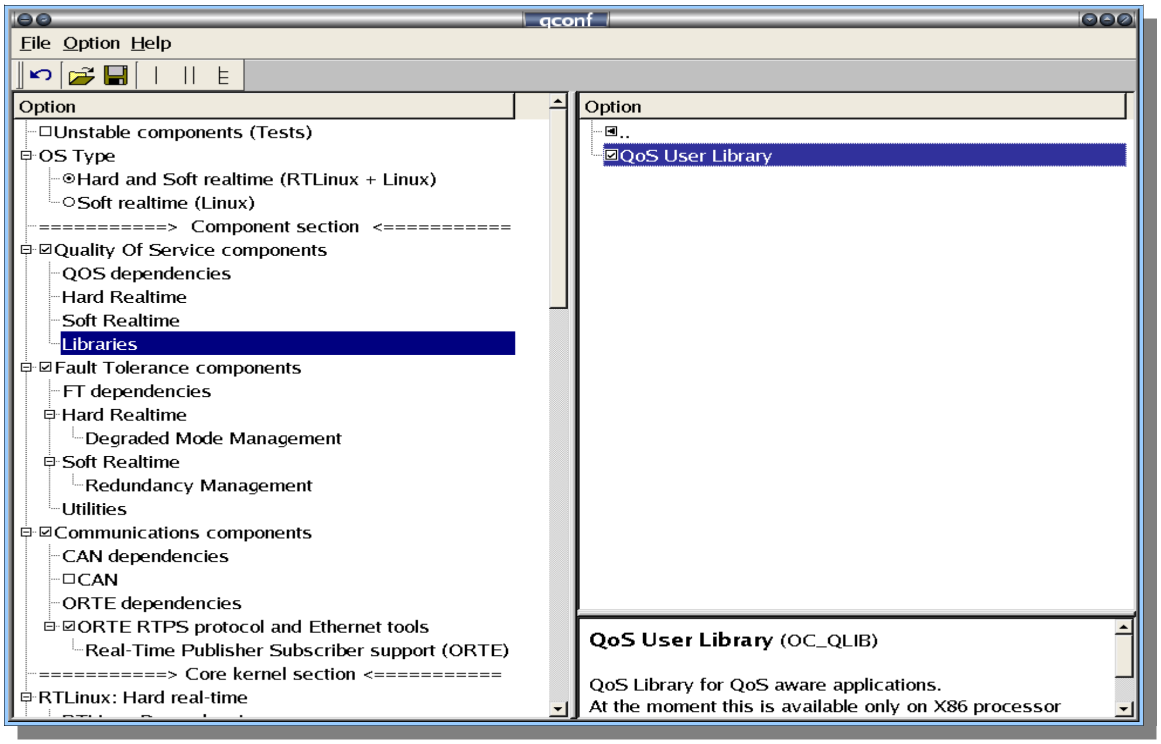
QoS Manager and its hook in the CBS scheduler module

Kernel module that allow to manage the CBS scheduler

Debug QoS scheduler

Enable printk kernel debug for QoS Manager module

c) Libraries



```
☐Unstable components (Tests)
⊟ OS Type
    ⦿Hard and Soft realtime (RTLinux + Linux)
    ○Soft realtime (Linux)
===========>  Component section  <===========
⊟ ☑Quality Of Service components
    QOS dependencies
    Hard Realtime
    Soft Realtime
    Libraries
⊟ ☑Fault Tolerance components
    FT dependencies
  ⊟ Hard Realtime
      Degraded Mode Management
  ⊟ Soft Realtime
      Redundancy Management
    Utilities
⊟ ☑Communications components
    CAN dependencies
    ☐CAN
    ORTE dependencies
  ⊟ ☑ORTE RTPS protocol and Ethernet tools
      Real-Time Publisher Subscriber support (ORTE)
===========> Core kernel section <===========
⊟ RTLinux: Hard real-time
```

```
..
☑QoS User Library
```

QoS User Library (OC_QLIB)

QoS Library for QoS aware applications.
At the moment this is available only on X86 processor

QoS Library for QoS aware applications. At the moment this is available only on X86 processor architectures.

10.2.2) Fault Tolerance

Fault-Tolerance components offer two types of facilities: **Degraded Mode Management** and **Redundancy Management.**

This results in two separate frameworks plus a common utility, the **Ftbuilder** which is a design tool that helps the developer specify the application behavior and constraints and generate specific code to support dynamic exploitation of such knowledge.

The Fault-Tolerance Building Tool, called **ftbuilder** is a TCL/TK 8.3 configuration tool that helps user specify Application FT and RT features and generates code for FT-application Management. It presently provides support for Degraded Mode Management. The ftbuilder is not a loadable module, it is provided as a directory that can be copied to user home environment to build ft_applications (see documentation for details). It is located under OCERA_DIR/components/ft.

The **Degraded Mode Management framework** is intended to be used at the hard RT level of OCERA, it requires thus a specific kernel configuration using RTLinux Hard real-time standard features, plus a few specific choices that are described below.

The **Redundancy Management framework** is intended to be used at the Soft real-time level of OCERA (Linux), it uses the ORTE communication component.

I Dependencies

Dependencies are different for each FT framework. They are detailed in the corresponding sections. In short, for **Degraded Mode Management framework,** at hard real-time level, main dependencies are related to core RTLinux features; while for **Redundancy Management framework,** at soft real-time level, dependencies concern ORTE communication components.

II Hard Real-time

The FT facilities available at hard real-time level are Degraded Mode Management.

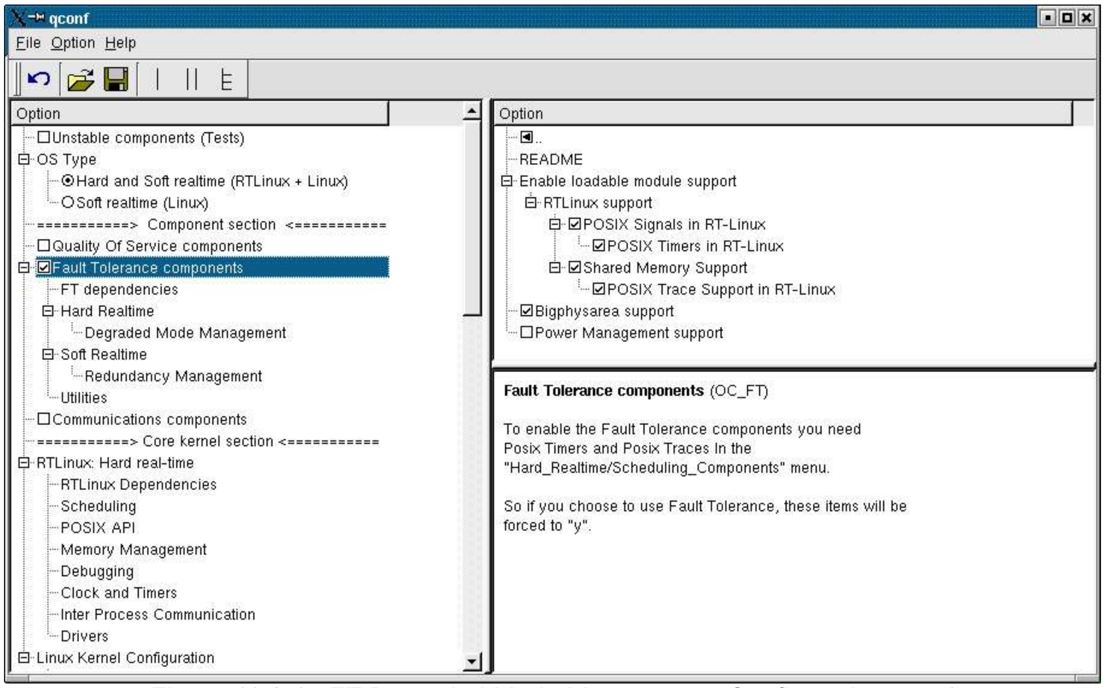
Degraded Mode Management configuration process takes three steps:

1. **OS Type Selection** : Soft and Hard real-time must be chosen and some sub-options must be checked.
2. **Components Selection** : FT components/Hard Realtime/degraded 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.



```
qconf
File  Option  Help

Option
 ☐ Unstable components (Tests)
 ⊟ OS Type
    ◉ Hard and Soft realtime (RTLinux + Linux)
    ○ Soft realtime (Linux)
 ===========> Component section <===========
 ☐ Quality Of Service components
 ⊟ ☑ Fault Tolerance components
    FT dependencies
    ⊟ Hard Realtime
       Degraded Mode Management
    ⊟ Soft Realtime
       Redundancy Management
    Utilities
 ☐ Communications components
 ===========> Core kernel section <===========
 ⊟ RTLinux: Hard real-time
    RTLinux Dependencies
    Scheduling
    POSIX API
    Memory Management
    Debugging
    Clock and Timers
    Inter Process Communication
    Drivers
 ⊟ Linux Kernel Configuration

Option
 ..
 README
 ⊟ Enable loadable module support
    ⊟ RTLinux support
       ⊟ ☑ POSIX Signals in RT-Linux
          ☑ POSIX Timers in RT-Linux
       ⊟ ☑ Shared Memory Support
          ☑ POSIX Trace Support in RT-Linux
 ☑ Bigphysarea support
 ☐ Power Management support

Fault Tolerance components (OC_FT)

To enable the Fault Tolerance components you need
Posix Timers and Posix Traces In the
"Hard_Realtime/Scheduling_Components" menu.

So if you choose to use Fault Tolerance, these items will be
forced to "y".
```

Figure 10.2.1 : 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.

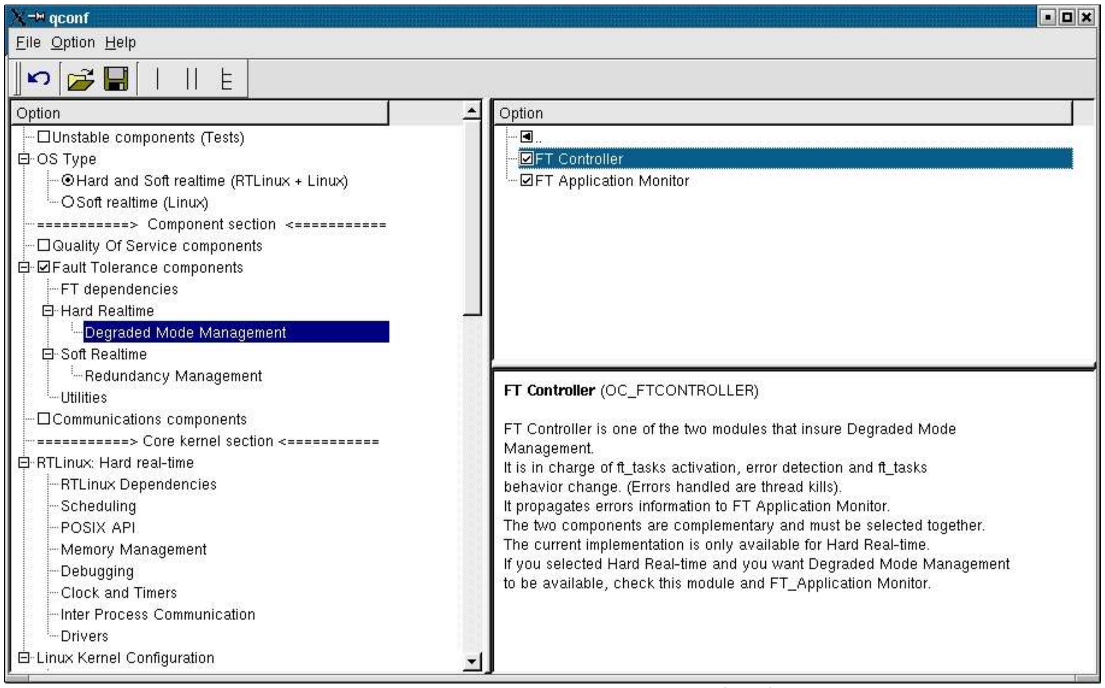
Note that POSIX Trace Support is mandatory for FT Degraded Mode Management components.

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.**



```
☐Unstable components (Tests)
⊟ OS Type
    ⦿Hard and Soft realtime (RTLinux + Linux)
    ○Soft realtime (Linux)
===========>  Component section  <===========
☐Quality Of Service components
⊟ ☑Fault Tolerance components
    FT dependencies
  ⊟ Hard Realtime
      Degraded Mode Management
  ⊟ Soft Realtime
      Redundancy Management
    Utilities
☐Communications components
===========> Core kernel section <===========
⊟ RTLinux: Hard real-time
    RTLinux Dependencies
    Scheduling
    POSIX API
    Memory Management
    Debugging
    Clock and Timers
    Inter Process Communication
    Drivers
⊟ Linux Kernel Configuration
```

```
..
☑FT Controller
☑FT Application Monitor
```

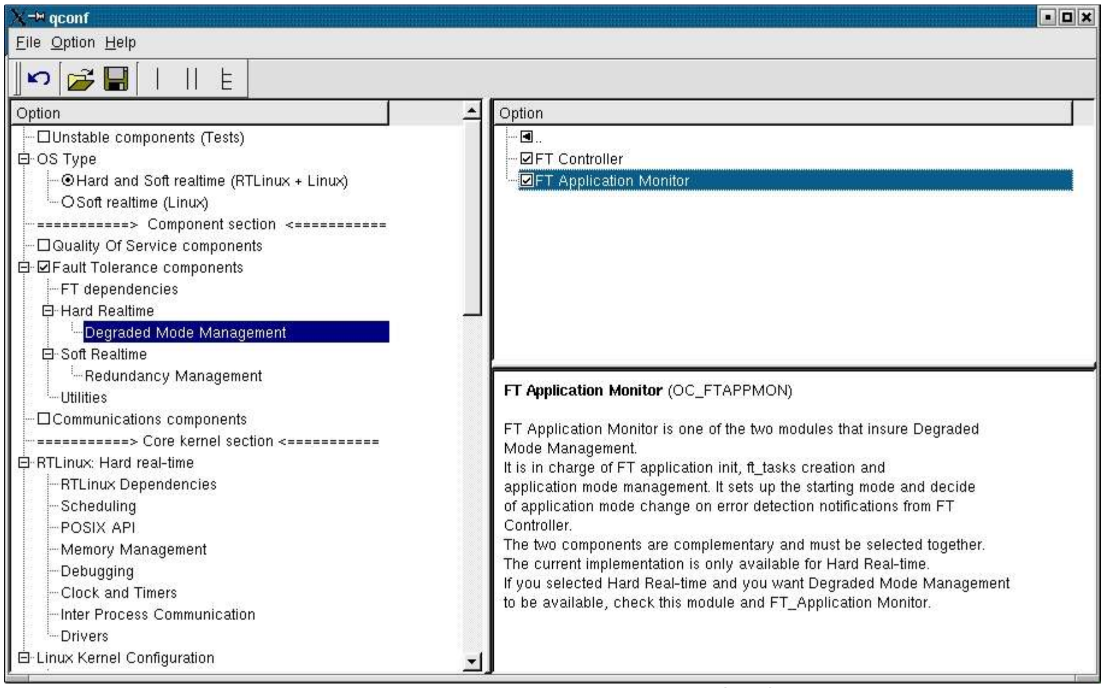
FT Controller (OC_FTCONTROLLER)

FT Controller is one of the two modules that insure Degraded Mode Management.
It is in charge of ft_tasks activation, error detection and ft_tasks behavior change. (Errors handled are thread kills).
It propagates errors information to FT Application Monitor.
The two components are complementary and must be selected together.
The current implementation is only available for Hard Real-time.
If you selected Hard Real-time and you want Degraded Mode Management to be available, check this module and FT_Application Monitor.

Figure 10.2.2 : FT Degraded Mode Management Configuration step2-1

The FTController is selected

The FT Application Monitor is selected.



[✓] FT Controller
[✓] FT Application Monitor

FT Application Monitor (OC_FTAPPMON)

FT Application Monitor is one of the two modules that insure Degraded Mode Management.
It is in charge of FT application init, ft_tasks creation and application mode management. It sets up the starting mode and decide of application mode change on error detection notifications from FT Controller.
The two components are complementary and must be selected together.
The current implementation is only available for Hard Real-time.
If you selected Hard Real-time and you want Degraded Mode Management to be available, check this module and FT_Application Monitor.

Figure 10.2.3 : FT Degraded Mode Management Configuration step2-2

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.

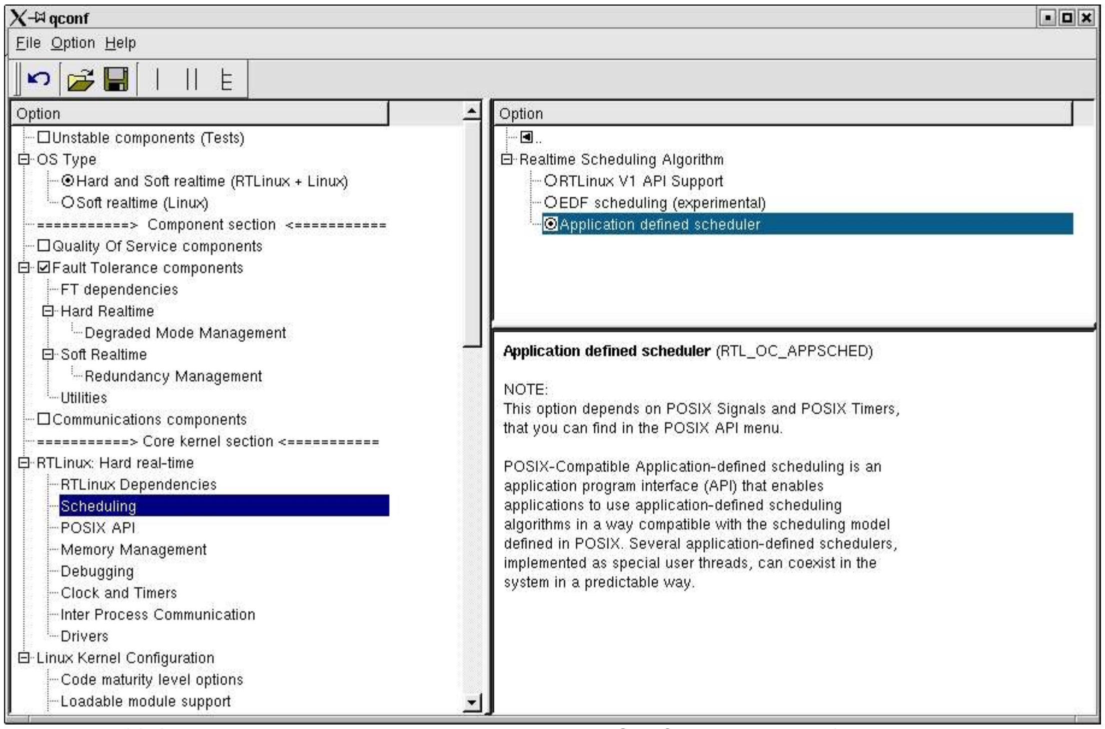
The error detection mechanism can handle two types of errors:

- 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.


qconf
File Option Help
Option
Unstable components (Tests)
OS Type
Hard and Soft realtime (RTLinux + Linux)
Soft realtime (Linux)
===========> Component section <===========
Quality Of Service components
Fault Tolerance components
FT dependencies
Hard Realtime
Degraded Mode Management
Soft Realtime
Redundancy Management
Utilities
Communications components
===========> Core kernel section <===========
RTLinux: Hard real-time
RTLinux Dependencies
Scheduling
POSIX API
Memory Management
Debugging
Clock and Timers
Inter Process Communication
Drivers
Linux Kernel Configuration
Code maturity level options
Loadable module support
Option
Realtime Scheduling Algorithm
RTLinux V1 API Support
EDF scheduling (experimental)
Application defined scheduler

Application defined scheduler (RTL_OC_APPSCHED)

NOTE:
This option depends on POSIX Signals and POSIX Timers, that you can find in the POSIX API menu.

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

Figure 10.2.4 : FT Degraded Mode Management Configuration step3 - Priority scheduler

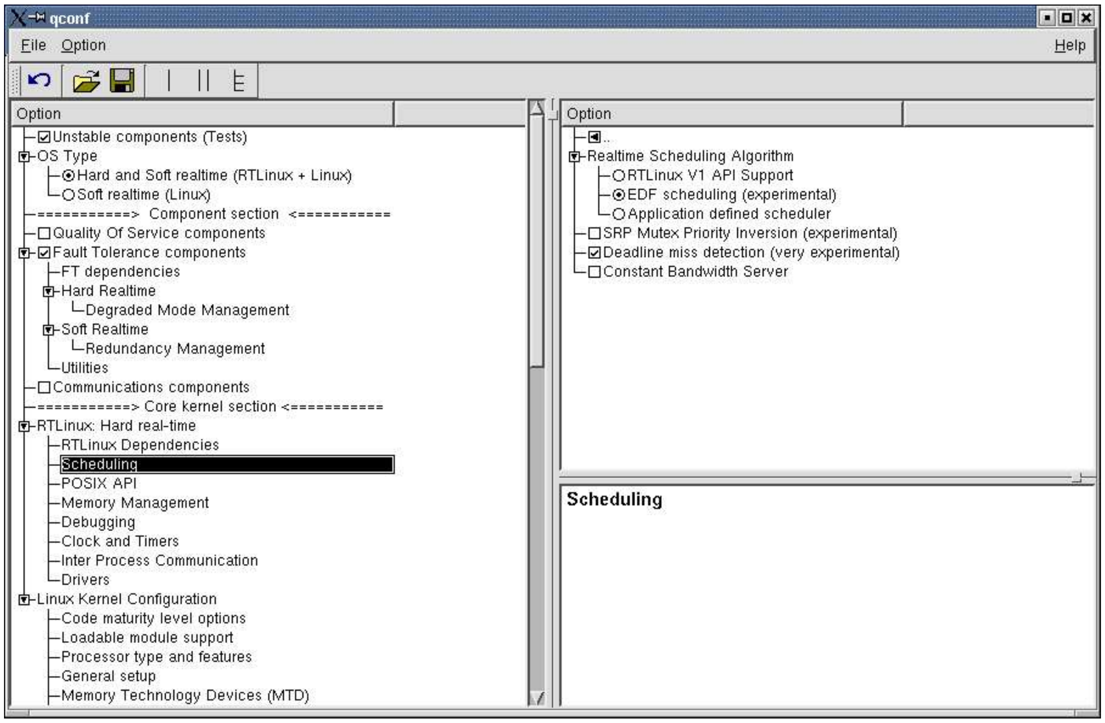
The **Application defined scheduler** component provides support for several types of scheduling policies defined above the RTLinux kernel itself. By default, the scheduling policy is based on priorities, which is the configuration that we must use in this case. For further details see section Scheduling.

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.



```
├☑Unstable components (Tests)
⊟-OS Type
│ ├◉Hard and Soft realtime (RTLinux + Linux)
│ └○Soft realtime (Linux)
├===========> Component section <===========
├☐Quality Of Service components
⊟-☑Fault Tolerance components
│ ├FT dependencies
│ ⊟-Hard Realtime
│ │ └Degraded Mode Management
│ ⊟-Soft Realtime
│ │ └Redundancy Management
│ └Utilities
├☐Communications components
├===========> Core kernel section <===========
⊟-RTLinux: Hard real-time
│ ├RTLinux Dependencies
│ ├Scheduling
│ ├POSIX API
│ ├Memory Management
│ ├Debugging
│ ├Clock and Timers
│ ├Inter Process Communication
│ └Drivers
⊟-Linux Kernel Configuration
  ├Code maturity level options
  ├Loadable module support
  ├Processor type and features
  ├General setup
  ├Memory Technology Devices (MTD)
```

```
├..
⊟-Realtime Scheduling Algorithm
│ ├○RTLinux V1 API Support
│ ├◉EDF scheduling (experimental)
│ └○Application defined scheduler
├☐SRP Mutex Priority Inversion (experimental)
├☑Deadline miss detection (very experimental)
└☐Constant Bandwidth Server
```

Figure 10.2.5 : 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.

III Soft Real-time

The 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

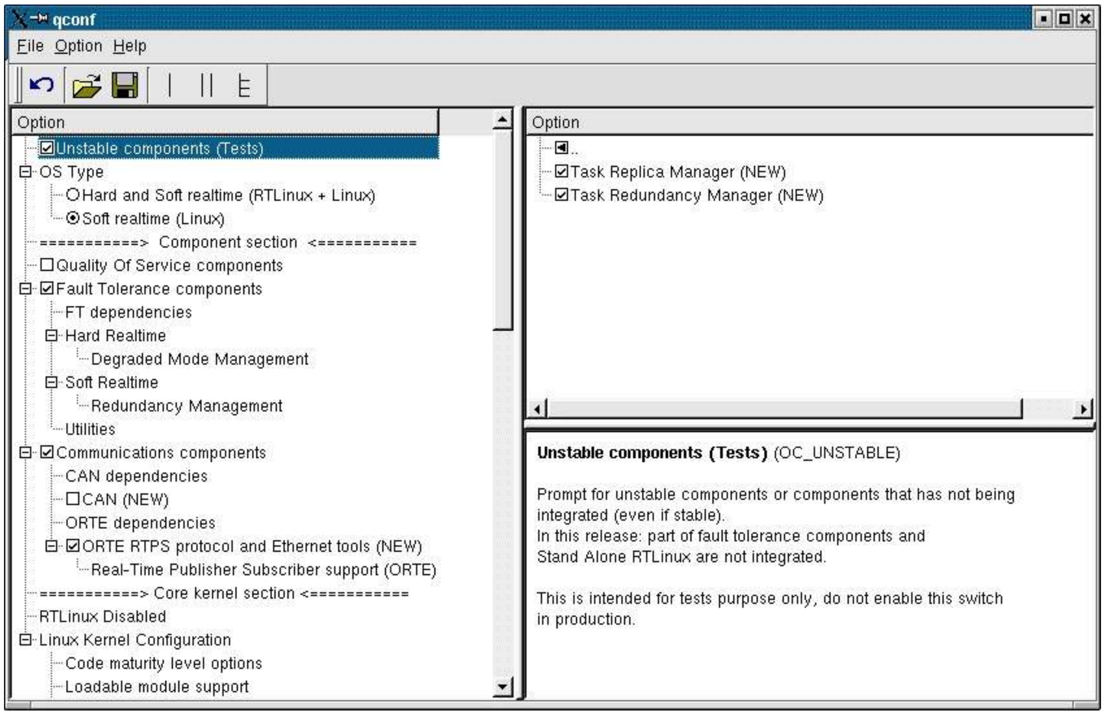
1. OS Type Selection

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

2. **Components Selection**

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.



```
Option
 ☑Unstable components (Tests)
⊟ OS Type
   ○Hard and Soft realtime (RTLinux + Linux)
   ●Soft realtime (Linux)
 ===========>  Component section  <===========
 ☐Quality Of Service components
⊟ ☑Fault Tolerance components
   FT dependencies
  ⊟ Hard Realtime
     Degraded Mode Management
  ⊟ Soft Realtime
     Redundancy Management
   Utilities
⊟ ☑Communications components
   CAN dependencies
   ☐CAN (NEW)
   ORTE dependencies
  ⊟ ☑ORTE RTPS protocol and Ethernet tools (NEW)
     Real-Time Publisher Subscriber support (ORTE)
 ===========> Core kernel section <===========
 RTLinux Disabled
⊟ Linux Kernel Configuration
   Code maturity level options
   Loadable module support
```

```
Option
 ..
 ☑Task Replica Manager (NEW)
 ☑Task Redundancy Manager (NEW)
```

Unstable components (Tests) (OC_UNSTABLE)

Prompt for unstable components or components that has not being integrated (even if stable).
In this release: part of fault tolerance components and Stand Alone RTLinux are not integrated.

This is intended for tests purpose only, do not enable this switch in production.

Figure 10.2.6 : FT Redundancy Management Configuration step1

2.2ORTE communication component selection

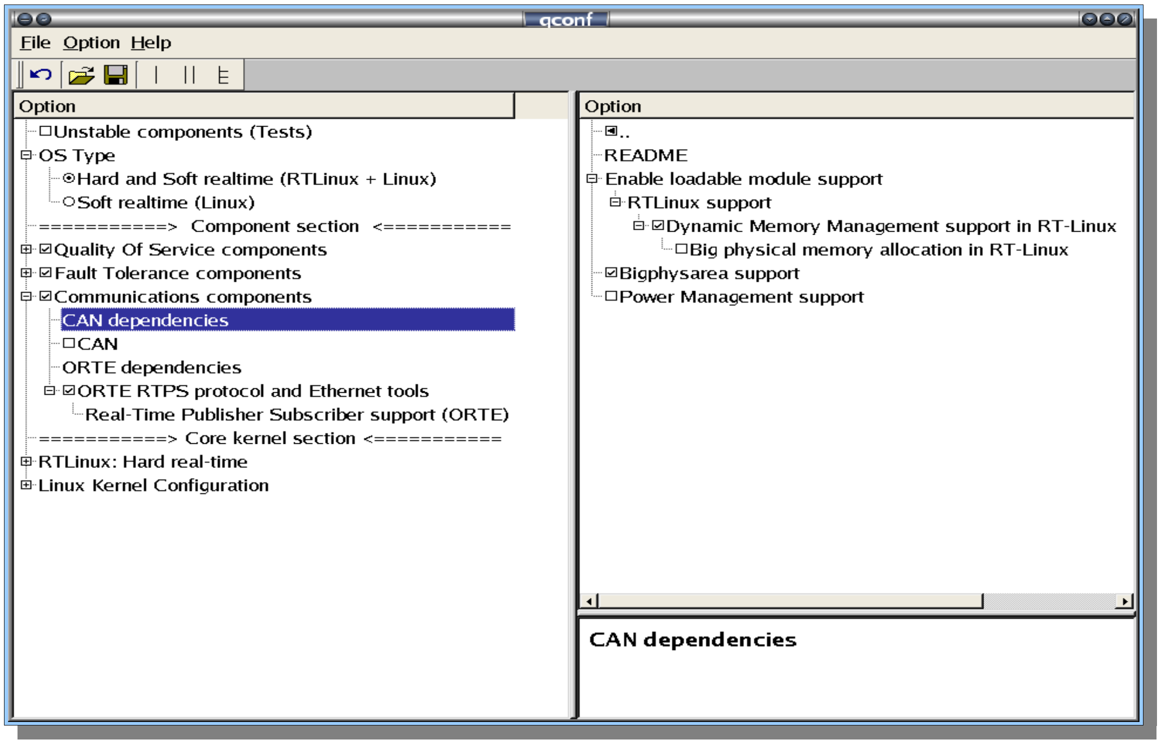
The Redundancy Management facility relies on ORTE (Realtime Ethernet) components that implement RTPS (RealTime Publisher Subscriber Protocol) communication protocol.

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

10.2.3) CAN

The CAN menu entry allow to define the way the CANBUS drivers and utility will work together with our system.

a) Dependencies



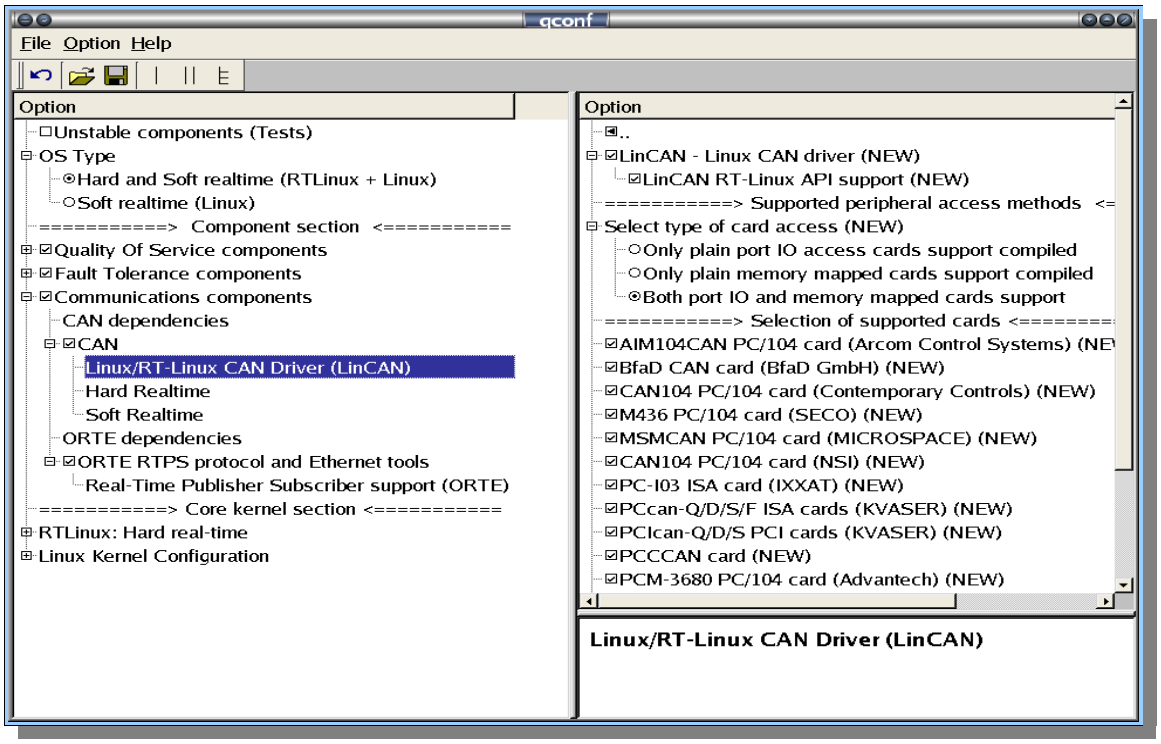
```
☐Unstable components (Tests)
⊟OS Type
   ⦿Hard and Soft realtime (RTLinux + Linux)
   ○Soft realtime (Linux)
===========>  Component section  <===========
⊞☑Quality Of Service components
⊞☑Fault Tolerance components
⊟☑Communications components
   CAN dependencies
   ☐CAN
   ORTE dependencies
   ⊟☑ORTE RTPS protocol and Ethernet tools
      Real-Time Publisher Subscriber support (ORTE)
===========> Core kernel section <===========
⊞RTLinux: Hard real-time
⊞Linux Kernel Configuration
```

```
..
README
⊟Enable loadable module support
  ⊟RTLinux support
    ⊟☑Dynamic Memory Management support in RT-Linux
       ☐Big physical memory allocation in RT-Linux
☑Bigphysarea support
☐Power Management support
```

CAN dependencies

The CANBUS drivers as other components depends on the enabling of *Loadable module support* and you must also disable the power management.
The CANBUS drivers and utilities can be used under RTLinux-GPL, the hard real-time environment or under the standard Linux kernel.
If you choose to work with hard real-time you will also need to enable the *Dynamic Memory Management support in RTLinux*, and the *Big physical memory allocation in RTLinux* switches.

b) CAN drivers



```
qconf
File  Option  Help

Option
 ☐Unstable components (Tests)
 ⊟ OS Type
    ◉Hard and Soft realtime (RTLinux + Linux)
    ○Soft realtime (Linux)
 ===========>  Component section  <===========
 ⊞ ☑Quality Of Service components
 ⊞ ☑Fault Tolerance components
 ⊟ ☑Communications components
     CAN dependencies
   ⊟ ☑CAN
       Linux/RT-Linux CAN Driver (LinCAN)
       Hard Realtime
       Soft Realtime
     ORTE dependencies
   ⊟ ☑ORTE RTPS protocol and Ethernet tools
       Real-Time Publisher Subscriber support (ORTE)
 ===========> Core kernel section <===========
 ⊞ RTLinux: Hard real-time
 ⊞ Linux Kernel Configuration

Option
 ..
 ⊟ ☑LinCAN - Linux CAN driver (NEW)
     ☑LinCAN RT-Linux API support (NEW)
 ===========> Supported peripheral access methods  <=
 ⊟ Select type of card access (NEW)
     ○Only plain port IO access cards support compiled
     ○Only plain memory mapped cards support compiled
     ◉Both port IO and memory mapped cards support
 ===========> Selection of supported cards <========
 ☑AIM104CAN PC/104 card (Arcom Control Systems) (NE
 ☑BfaD CAN card (BfaD GmbH) (NEW)
 ☑CAN104 PC/104 card (Contemporary Controls) (NEW)
 ☑M436 PC/104 card (SECO) (NEW)
 ☑MSMCAN PC/104 card (MICROSPACE) (NEW)
 ☑CAN104 PC/104 card (NSI) (NEW)
 ☑PC-I03 ISA card (IXXAT) (NEW)
 ☑PCcan-Q/D/S/F ISA cards (KVASER) (NEW)
 ☑PCIcan-Q/D/S PCI cards (KVASER) (NEW)
 ☑PCCCAN card (NEW)
 ☑PCM-3680 PC/104 card (Advantech) (NEW)

Linux/RT-Linux CAN Driver (LinCAN)
```

In this sub-menu you will choose the CAN driver for your card or you cards. You can indeed have up to 4 CANBUS cards in your system.

LinCAN - Linux CAN driver

The driver can be compiled in two modes:
- Linux only driver, prerequisite is only kernel modules support
- Linux+RT-Linux driver

The RT-Linux and RT-Linux dynamic memory support is required for the second compilation mode. Driver provides equivalent device driver API for both Linux and RT-Linux in such case.

LinCAN RT-Linux API support

RT-Linux application interface to LinCAN driver.
This modifies behavior of the LinCAN driver, the chips supporting interrupt routines are called from RT-Linux worker threads

Select type of card access

Select the type of the card to be supported by the LinCAN driver.

- The first choice is to support only cards with linear I/O port chip access style.
- The second choice is to support cards with directly memory mapped chips.
- The third choice enables runtime selection of the chip access style when card registration function is invoked. This can be used for cards with segmented and indexed mapping of the chips as well.

Using the last entry, the driver will compile with dynamic port access. This enables to support both, memory mapped and port IO style cards and cards, which use segmented and indexed access to chip ports.
This is best choice for general case, but results in slightly bigger chip access overhead.

Supported cards

The following CANBUS cards are already supported:

- AIM104CAN PC/104 card (Arcom Control Systems)
- BfaD CAN card (BfaD GmbH)
- CAN104 PC/104 card (Contemporary Controls)
- M436 PC/104 card (SECO)
- MSMCANPC/104 card (MICROSPACE)
- CAN104PC/104 card (NSI)
- PC-I03ISA card (IXXAT)
- PCcan-Q/D/S/FISA cards (KVASER)
- PCIcan-Q/D/SPCI cards (KVASER)
- PCCCANcard
- PCM-3680PC/104 card (Advantech)
- ISAmemory mapped sja1000 CAN card (PiKRON Ltd.)
- pip5computer (MPL)
- SmartCANcard
- SSVcard

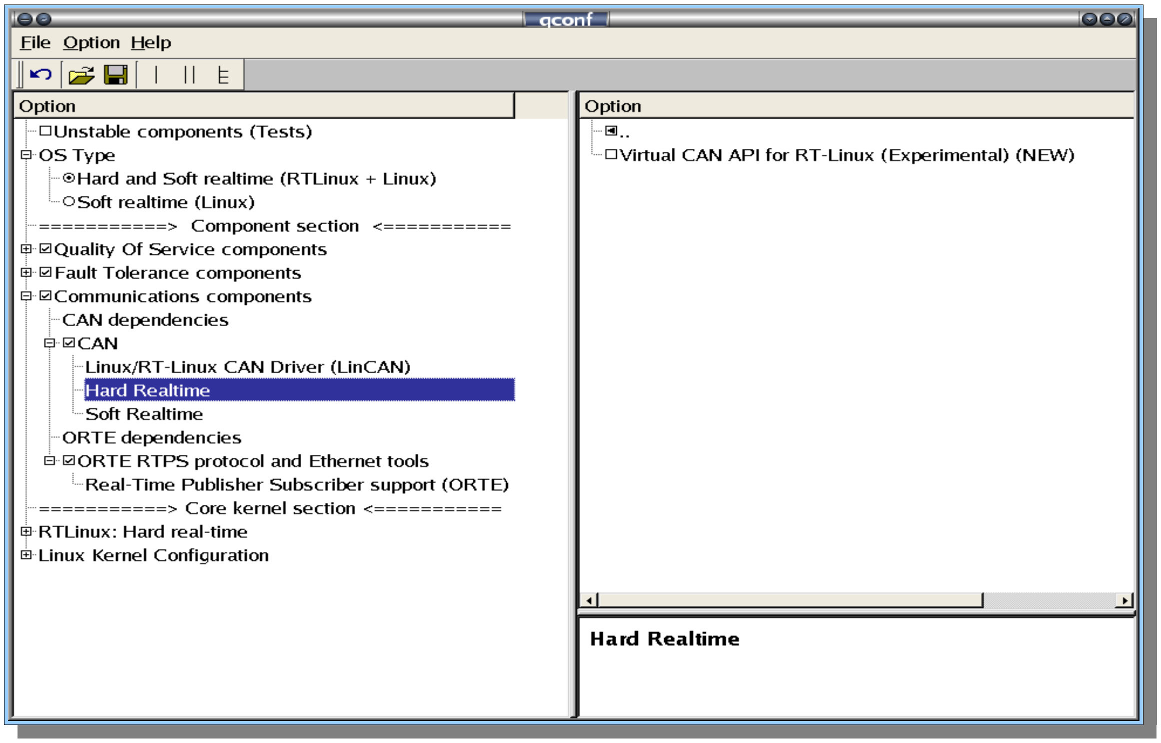
There is also a template driver *Templatedriver* for developing drivers for other cards

Virtual CAN board

The driver also support a virtual interface for testing purposes under Linux.
Virtual CAN board support. This enables to interconnect local clients/applications with exactly same interface as if real CAN bus is used. It can even emulate some transfer delays for Linux only compilation.

It can be used even for interconnection and testing of Linux and RT-Linux CAN devices.

c) Hard real-time



```
□Unstable components (Tests)
⊟ OS Type
   ⦿Hard and Soft realtime (RTLinux + Linux)
   ○Soft realtime (Linux)
===========>  Component section  <===========
⊞ ☑Quality Of Service components
⊞ ☑Fault Tolerance components
⊟ ☑Communications components
   CAN dependencies
   ⊟ ☑CAN
      Linux/RT-Linux CAN Driver (LinCAN)
      Hard Realtime
      Soft Realtime
   ORTE dependencies
   ⊟ ☑ORTE RTPS protocol and Ethernet tools
      Real-Time Publisher Subscriber support (ORTE)
===========> Core kernel section <===========
⊞ RTLinux: Hard real-time
⊞ Linux Kernel Configuration
```

```
..
□Virtual CAN API for RT-Linux (Experimental) (NEW)
```

Hard Realtime

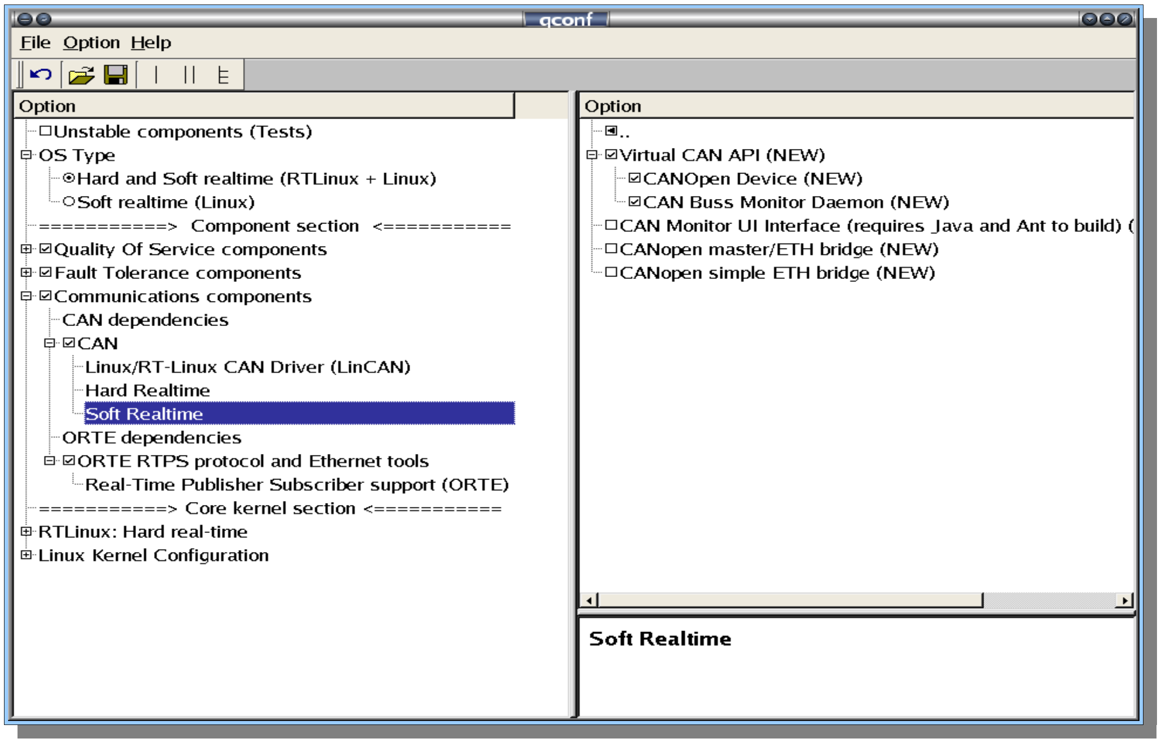
Virtual CAN API for RT-Linux (Experimental)

RT-Linux version of VCA library.

CANOpen Device for RT-Linux (Not finished yet)

RT-Linux version of CANopen slave device.

d) Soft real-time



```
☐Unstable components (Tests)
⊟ OS Type
   ⦿Hard and Soft realtime (RTLinux + Linux)
   ○Soft realtime (Linux)
===========>  Component section  <===========
⊞ ☑Quality Of Service components
⊞ ☑Fault Tolerance components
⊟ ☑Communications components
   CAN dependencies
   ⊟ ☑CAN
      Linux/RT-Linux CAN Driver (LinCAN)
      Hard Realtime
      Soft Realtime
   ORTE dependencies
   ⊟ ☑ORTE RTPS protocol and Ethernet tools
      Real-Time Publisher Subscriber support (ORTE)
===========> Core kernel section <===========
⊞ RTLinux: Hard real-time
⊞ Linux Kernel Configuration
```

```
..
⊟ ☑Virtual CAN API (NEW)
   ☑CANOpen Device (NEW)
   ☑CAN Buss Monitor Daemon (NEW)
☐CAN Monitor UI Interface (requires Java and Ant to build) (
☐CANopen master/ETH bridge (NEW)
☐CANopen simple ETH bridge (NEW)
```

Soft Realtime

Virtual CAN API

This library implements abstraction layer above low level system dependent CAN driver API and basic CANopen SDO/PDO transfer protocols.
CANopen Object Dictionary functions are provided by library too.
The VCA library can be used to build CANopen master and slave devices.

CANOpen Device

Linux version of configurable dynamic CANopen slave device which builds its Object Dictionary from EDS (CANopen Electronic Data Sheet).

CAN Buss Monitor Daemon

Daemon providing TCP/IP bridge to CAN/CANopen network.

CAN Monitor UI Interface (requires Java and Ant to build)

UI interface for CAN/CANopen buss monitoring and CANopen device Object Dictionary manipulations.

Communicates with CAN Monitor Daemon through TCP/IP sockets.

CANopen master/ETH bridge

CANopen master/ETH bridge (requires C++)

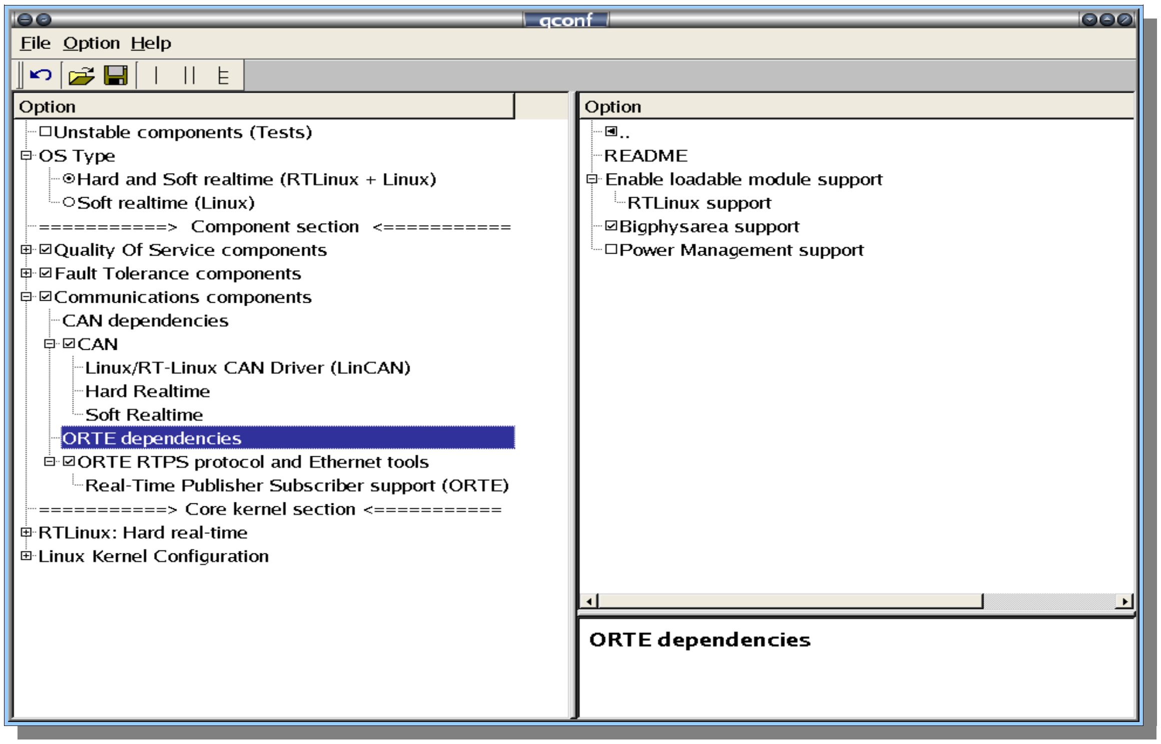
CANopen simple ETH bridge

CANopen simple ETH bridge

10.2.4) ORTE

ORTE,the Ocera Real-Time Ethernet, is based on Real-Time Publisher Subscriber support (ORTE)

a) Dependencies

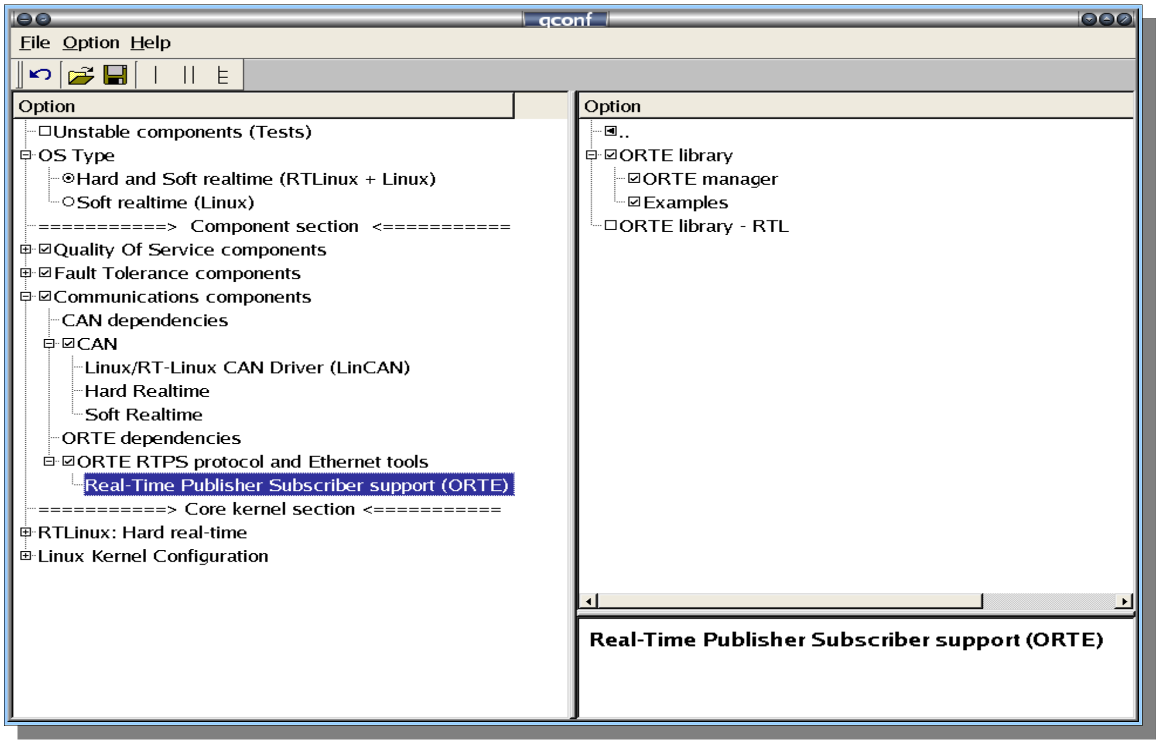


```
Option
 ☐Unstable components (Tests)
⊟ OS Type
   ⦿Hard and Soft realtime (RTLinux + Linux)
   ○Soft realtime (Linux)
 ===========>  Component section  <===========
⊞ ☑Quality Of Service components
⊞ ☑Fault Tolerance components
⊟ ☑Communications components
   CAN dependencies
  ⊟ ☑CAN
     Linux/RT-Linux CAN Driver (LinCAN)
     Hard Realtime
     Soft Realtime
   ORTE dependencies
  ⊟ ☑ORTE RTPS protocol and Ethernet tools
     Real-Time Publisher Subscriber support (ORTE)
 ===========> Core kernel section <===========
⊞ RTLinux: Hard real-time
⊞ Linux Kernel Configuration
```

```
Option
 ..
 README
⊟ Enable loadable module support
   RTLinux support
 ☑Bigphysarea support
 ☐Power Management support
```

ORTE dependencies

ORTE, as other components depends on the enabling of *Loadable module support* and you must also disable the power management.



```
Option
 ☐Unstable components (Tests)
⊟ OS Type
   ⦿Hard and Soft realtime (RTLinux + Linux)
   ○Soft realtime (Linux)
 ===========>  Component section  <===========
⊞ ☑Quality Of Service components
⊞ ☑Fault Tolerance components
⊟ ☑Communications components
   CAN dependencies
  ⊟ ☑CAN
      Linux/RT-Linux CAN Driver (LinCAN)
      Hard Realtime
      Soft Realtime
   ORTE dependencies
  ⊟ ☑ORTE RTPS protocol and Ethernet tools
      Real-Time Publisher Subscriber support (ORTE)
 ===========> Core kernel section <===========
⊞ RTLinux: Hard real-time
⊞ Linux Kernel Configuration
```

```
Option
   ..
⊟ ☑ORTE library
    ☑ORTE manager
    ☑Examples
  ☐ORTE library - RTL
```

Real-Time Publisher Subscriber support (ORTE)

b) Soft real-time

ORTE library

The Orte Library allow to develop Linux programs using the RTPS protocol to cooperate in an ORTE network.

ORTE manager

The ORTE manager is a Linux program handling Publishers and Subscribers requests. You need to have at least one manager in an RTPS network.

Examples

As you may expect this entry enable the building of exemples. If you begin with ORTE we advise you to build the exemples.

c) hard real-time

ORTE library – RTL

The Orte Real-time Library allow to develop RTLinux tasksusing the RTPS protocol to cooperate in an ORTE network

ORTE manager – RTL

The RTL-ORTE manager is a RTLinux task handling Publishers and Subscribers requests.
You need to have at least one manager in an RTPS network.

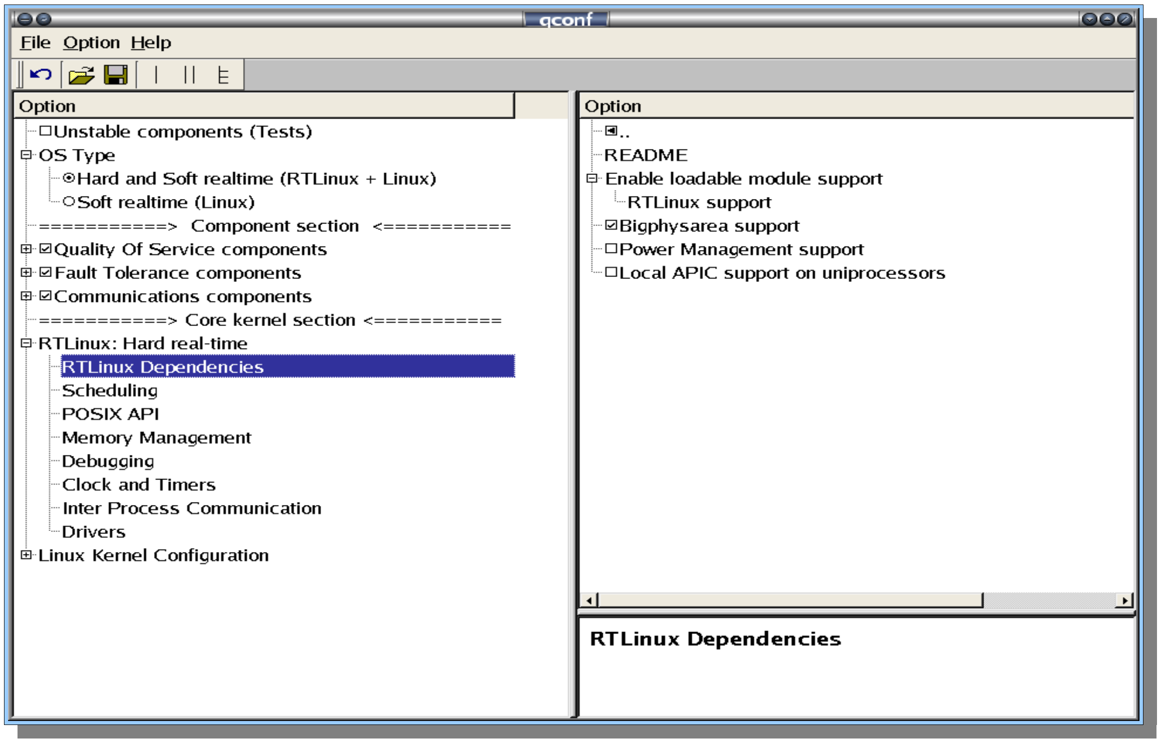
Examples – RTL

As you may expect this entry enable the building of exemples of Real-time tasks. If you begin with ORTE we advise you to build the exemples.

10.2.5) RTLinux

a) Dependencies

If you intend to use RTLinux-GPL as a real-time kernel under Linux, you will need to enable *Loadable Module Support,* and to disable *Power management support* and *Local APIC support* if you use mono-processor main-boards.



☐Unstable components (Tests)
⊟ OS Type
  ⦿Hard and Soft realtime (RTLinux + Linux)
  ○Soft realtime (Linux)
===========> Component section <===========
⊞ ☑Quality Of Service components
⊞ ☑Fault Tolerance components
⊞ ☑Communications components
===========> Core kernel section <===========
⊟ RTLinux: Hard real-time
  RTLinux Dependencies
  Scheduling
  POSIX API
  Memory Management
  Debugging
  Clock and Timers
  Inter Process Communication
  Drivers
⊞ Linux Kernel Configuration

..
README
⊟ Enable loadable module support
  RTLinux support
☑Bigphysarea support
☐Power Management support
☐Local APIC support on uniprocessors

RTLinux Dependencies

b) Scheduling

Realtime Scheduling Algorithm

You may choose between different scheduling algorithm:

- **RTLinux V1 API Support**
  Say Y here if you need the old RTLinux v1 API.
- **EDF scheduling (experimental)**
  Say Y if you want to use the Earliest Deadline First scheduling (EDF) policy. Tasks with closer deadline are scheduled first. The EDF policy is only applied among tasks of the same priority. Therefore, it is possible to jointly use fixed and dynamic priorities. This is an extension to the POSIX API. Therefore, you have to disable "RTLinux V1 API Support". If you say Y here you may also need "SRP Mutex Priority Control Inversion". It is safe to say yes

- **Application defined scheduler**
  This option depends on POSIX Signals and POSIX Timers, that you can find in the POSIX API menu. POSIX-Compatible Application-defined scheduling is an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way.

Library of Application Schedulers: Disabled

SRP Mutex Priority Inversion (experimental)

To enable this option you need to select "EDF Scheduling" in this same menu.
Say Y if you want to use the Stack Resource Protocol (SRP) to control the priority inversion of the mutex. The SRP is an improvement over the "POSIX priority ceiling protocol" to allow dynamic priorities.
This is an extension to the POSIX API. Therefore, you have to disable "RTLinux V1 API Support".
If you use EDF scheduling and mutex, then you have to say Y here.
It is safe to say yes.

Constant Bandwidth Server

To enable this option you need to select "EDF Scheduling" in this same menu.
Say Y if you want to use the Constant Bandwidth Server scheduling policy (CBS).
This is an extension to the POSIX API. Therefore, you have to disable "RTLinux V1 API Support".
It is safe to say yes.

c) POSIX API

POSIX Standard IO

To enable this option you need to select "EDF Scheduling" in this same menu.
Say Y if you want to use the Constant Bandwidth Server scheduling policy (CBS).
This is an extension to the POSIX API. Therefore, you have to disable "RTLinux V1 API Support".
It is safe to say yes

POSIX Priority Protection

Enabling this option, RTLinux mutexes will support the PTHREAD_PRIO_PROTECT protocol.
Please, remember that the default mutex protocol is PTHREAD_PRIO_NONE. Therefore, you have to explicitly request to use PTHREAD_PRIO_PROTECT protocol on every mutex you want (see the pthread_mutexattr_setprotocol and pthread_mutexattr_setprioceiling functions).

POSIX Barriers

Barriers is a synchronisation facility used to ensure that all threads have reached a particular stage in a parallel computation before allowing any to proceed to the next stage.

POSIX Signals

A POSIX signal is the software equivalent of an interrupt or exception occurrence. Say Y here if you want to make use of the POSIX interface to signals. This allows you to send signals (RTL_SIGUSR1,RTL_SIGUSR2 and from signal RTL_SIGRTMIN to RTL_SIGRTMAX) to threads (pthread_kill) blocking signals (pthread_sigmask), suspend a thread waiting for a signal to arrive (sigsuspend) and install signal handlers, among other things. Signals are in determinated cases a good interprocess communication mechanism.

Handling of processor exception through POSIX signals

If you want to handle processor exceptions through POSIX signals.
This option is incompatible with RTLinux Debugger because both use the same underlying hardware.

POSIX Timers

POSIX timers allows a mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed. Facilities supported by POSIX timers that are desirable for real-time operating systems:

- Support for additional clocks.
- Allowance for greater time resolution (modern timers are capable of nanosecond resolution; the hardware should support it)
- Ability to use POSIX Signals to indicate timer expiration.

POSIX timers depends on POSIX signals.

POSIX Messages Queues

The message passing facility described in IEEE Std 1003.1-2001 allows processes and threads to communicate through system-wide queues. These message queues are accessed through names. A message queue can be opened for use by multiple sending and/or multiple receiving processes.
Interprocess communication utilizing message passing is a key facility for the construction of deterministic, high-performance realtime applications.
Some characteristics of POSIX Message Queues are:

- Fixed size of messages
- Prioritization of messages
- Asynchronous notification

Maximum number of open message queue descriptors

Maximum number of message priorities

Maximum number of message queues

Maximum number of messages in the default message queue attributes

Maximum message size in the default message queue attributes

POSIX Trace Support

This package adds (most of) the tracing support defined in the POSIX Trace standard. The POSIX Trace standard defines a set of portable interfaces for tracing applications. This standard was recently approved (September 2000) and now can be found integrated with the rest of POSIX 1003.1 standards in a single document issued in 2001 after the approval of both the IEEE-SA Standards Board and the Open Group.
To enable this option you need "Shared Memory Driver" (Memory Management->Shared Memory Support).

d) Memory Management

Dynamic Memory Management support

This component provides the standard malloc and free functions to allocate and release dynamic memory.
The internal design implements a good-fit policy using a two level segregated fit data structure. The execution time is both bounded and fast. Also, wasted memory is very low (much better than a buddy system).
As far as the authors know, this is, jointly with the half-fist allocator (proposed by Takeshi Ogasawara), the only allocators that meet real-time requirements.
By default, the initial memory pool is allocated using the Linux vmalloc() function. If you need a different type of memory (DMA safe), please enable the next option.

Big physical memory allocation

This option depends on linux BIGPHYSAREA option that you can also find in the RTLinux dependencies menu.
Select this option of you need to allocate memory able to be used by DMA devices (memory physically contiguous).
If you do NOT select this option then the RTLinux dynamic memory allocator will allocate the initial memory pool using the vmalloc() function (which means that the memory returned by the RTLinux allocator are not guaranteed to be physically contiguous).
Remember to configure properly your Linux boot so that the Linux BigphysArea driver is initialised properly (see the BigPhysArea documentation).
You will have to say yes here if you intend to use the Ada Real-Time Environment and say "no" otherwise, unless you know need to allocate physically contiguous blocks of memory.

Shared Memory Support

Provides a non-standard mechanism to share memory between RTLinux threads and Linux processes.
Some RTLinux facilities (POSIX trace Support) relies on this driver.
NOTE: "POSIX Trace Support" depends on this option. If you deselect "Shared Memory Support" then "POSIX Trace Support" will be automatically deselected.

e) Debugging

Enable debugging

This option compiles RTLinux modules with debugging support.
Say Y if you want to debug RT-programs but take care that this option will greatly extend the size of the programms.

rtl_printf uses printk

Say Y here if you want rtl_printf output to be buffered and then passed to Linux printk facility. It is useful if you are in X-Windows, since the output can then be viewed via dmesg and/or syslog. In certain situations, you may want to disable this option, for example when Linux has no chance to print a message (a crash occurs).

RTLinux Tracer support (experimental)

The RTLinux tracer allows tracing various events in the system.
This facility has been obsoleted by the POSIX Trace implementation.

RTLinux Debugger

This option depends on POSIX exceptions in the POSIX API menu being disabled.

f) Clock and Timers

Synchronized clock support

RT-Linux High Resolution Timers: Disabled

g) Inter Process Communication

Wait queue facility that can be used from RTLinux and Linux

This package contains a basic mechanism for synchronising RTLinux tasks and Linux kernel threads. This mechanism provides basic facilities for suspending RTLinux tasks and Linux kernel threads waiting for common events.
This facility defines two functions: one for sleeping a job on a queue (shq_wait_sleep) and another function to wakeup all jobs waiting in that queu (shq_wait_wakeup).
'shq_wait_sleep' suspends the current thread or task in the corresponding queue until a 'shq_wait_wakeup' is invoked over this queue.

Inter process communication through RTLinux FIFO

This allow you to synchronize Real-Time threads between each other and also to synchronize Real-Time threads and Linux processes through /dev/rtf special files.

Max number of fifos

Define the maximum number of fifos used in the system. These are the real-time FIFOs including pre-allocated and non pre-allocated fifos used by the Real-Time tasks as well as for Linux/RTLinux synchronisation.

Preallocated fifo buffers

This permit to allocate the FIFO buffers at the system start instead of allocate the buffers during the tasks runtime.

Size (in bytes) of preallocated fifos

If you preallocated the RTL-FIFOs you must define here the size you want to allocate for them.
It is the real size here and all FIFOs have the same size. If you do not know what to do with this, let the value to the 2048 default.

Number of preallocated fifos

If you preallocated the RTL-FIFOs you must define here the number of pre-allocated fifos you want to initialize.

h) Drivers

AT the moment we do not have a lot of drivers integrated in OCERA Real-Time part.
The CAN-BUS cards drivers are defined earlier in the communication components part.

Serial Port Driver

If you want to access a serial port from a real-time task you must use this interface.
In this case, you cannot use the serial Linux driver from within Linux.

Floating Point Support

This allows the use of FP operations in real-time threads.

10.2.6) Linux

We advise you to use the standard LINUX documentation to configure LINUX.
You can enable any driver you want in the LINUX configuration tree, the one that would not work with RTLinux-GPL are disabled if you did choose RTLINUX.

10.3) Building the kernel and components

10.3.1) make

It is time now to build the kernel and the components.
Now that you have configure the system you just have to issue a make command in the main OCERA directory.

```
# make
```

The complete kernel, modules and components should compile and be placed in the **target-${ARCH}** directory.

10.3.2) The target directory structure

Directory	***Usage***
boot	Linux kernel
lib/modules	All the Linux and RTLinux modules
usr/lib/	Libraries
usr/include	Headers
usr/local/orte	ORTE subdirectories with manager binary and exemples.
usr/local/can	CAN-Open subdirectories with utilities and monitor
usr/local/ft	Fault Tolerance tools
usr/local/qos	QoS tools
usr/local/ocera	miscellaneous OCERA tools like tracer, debuggers

10.4) Building the tools

The tools are independent of the kernel and are compiled separately.
Each partner developing a tool and having time in this WP should send me a chapter.
Here tools like debugger, tracing tools, analysers.

THIS CHAPTER IS TBD (To Be Done)

11) System Integration

Pierre Morel - MNIS

11.1) Compiling a custom application

You will have all informations on this subject in the Programmer's Guide.
As a summary the way to build an application is the following, depending if you want to build a Linux or an RTLinux-GPL application.

11.1.1) Linux Application

To build a Linux application using OCERA , you only need to link your application with the components library you need.
After having built the kernel and components, you find the libraries and headers under the target-${ARCH} directory.

File	*Usage*
Usr/lib/liborte.a	ORTE Library
Usr/lib/libperiodic.a	QOS Library
Usr/include	All RTLinux, Linux and components include files
Usr/lib/libposix_trace.a	POSIX trace libraries

After you built your application you will have to put it on the right place on the embedded system or on the training system.
This is explained in the next chapter.

11.1.2) RTLinux Application

If you build a RTLinux application, you will have to build your application following some rules.
The makefile
Here is an exemple of RTLinux application, follow it to retrieve the right compilers options.
Remember, we give you only an overview here in this guide, you will learn much more on the compiler options in the "**OCERA programmer's Guide"**.

```
all: frank_module.o frank_app
# simple.o

include ../../rtl.mk
frank_app: frank_app.c
    $(CC) ${INCLUDE} ${USER_CFLAGS} -O2 -Wall frank_app.c -o frank_app

frank_module.o: frank_module.c
    $(CC) ${INCLUDE} ${CFLAGS} -c frank_module.c -o frank_module.o
```

As a summary: include the **rtl.mk** file and compile the RTLinux application as a Linux module.
You will then need to install the module on your running system and insert it into the kernel to launch your Real-Time task.

11.2) Installing a training systems

This is certainly the easiest way to install the OCERA system, just copy the **target-${ARCH}** subdirectories to the root directory of your system.
For exemple, assuming you have compile for a **i386** architecture and your system is the target, just do:

```
# cd target-i386
# tar cf - * | ( cd / ; tar xvf - )
```

to **overwrite** your OCERA libraries in /usr/lib, your kernel modules in /lib/modules/ocera-1.0.0/ and your include files in /usr/include.
If you want to keep the old versions, take care to back it up before.
With this command you will also copy the kernel, **vmlinuz-2.4.18-ocera-1.0.0**, into the /boot directory and you will need to install the kernel reference into the boot loader using the boot loader installer.
Assuming you use **lilo,** you will modify the file **/etc/lilo.conf** with something like:

```
lba32
# Change boot and root !!
boot=/dev/hda
root=/dev/hda1
#
install=/boot/boot-menu.b
map=/boot/map
delay=20
prompt
timeout=150
vga=normal

image=/boot/vmlinuz-2.4.18-ocera-0.5
   label=ocera(0.5)
   read-only
   append="bigphysarea=1024"
   optional
```

Of course you must change the boot and root entries according to your settings. Look at the lilo man pages for more informations using lilo.
After editing /etc/lilo.conf, just run the installer and reboot:

```
# lilo
# reboot
```

And you must now be rebooting with the OCERA kernel ready to launch your hard real-time and/or soft real-time applications.

11.3) Installing an embedded systems

Trough there is many ways to install an embedded system and even more to download the code into the embedded system we will only focus on the root file system creation and booting on a CDROM.
The steps to build an OCERA system are:

- retrieve the sources
- building the tools, kernel and applications
- make a working filesystem
- Install a boot system

11.3.1) Retrieve the sources

You can retrieve the sources from the SourceForge server.

Actual sources, at the moment this paper is being written is ocera-1.0.0

a) From a Debian APT-Site

This is the best way to get the complete sources, development environment and tools if you

- have an Internet connection
- use a Debian Distribution, which is strongly recommended as it is the only supported Linux distribution.

In that cas, you may add the following lines to the sources.list file of the /etc/apt directory:

```
deb http://www.ocera.net/ocera sarge main
```

and issue an update for the apt system followed by an installation like:

```
apt-get update
```

```
apt-get install ocera-dev
```

At the time we write this documentation, we have four packages that could be installed

package	***content***	***command to issue***
Development	All sources: Real Time kernel, Soft real time linux kernel, OCERA components	apt-get install ocera-dev
apt-get install ocera-tools	Tracing tools	apt-get install ocera-tools
apt-get install ocera-doc	All the documentation including white papers	apt-get install ocera-doc
apt-get install ocera-rtlgnat	The Ada compiler and Real-time runtime	apt-get install ocera-rtlgnat

b) From the tarball

This is certainly the best way to have a stable version if you use another Linux distribution than Debian.
Using another distribution is possible, although not recommended, you will be on your own to resolve possible dependencies problems.
In that case, download ocera-1.0.0 from the summary page of the OCERA SourceForge site:

http://sourceforge.net/projects/ocera/

c) From the CVS

If you want to retrieve the sources of OCERA Components, Linux and RTLinux from the OCERA CVS server, you can do the following:
Make a directory (suppose /usr/share/OCERA) and change to this directory and issue the following commands:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/ocera login
cvs checkout -P -r ocera-1_0_0-release -d ocera-1.0.0 ocera
```

This will check out the , at the time we write the document, actual stable version and release of the OCERA CVS and create the /usr/shar.e/OCERA/ocera-1.0.0 directory and the ocera structure and files inside.
If you prefer to check out the last snapshot with the last untested bug correction, you can issue:

```
cvs -d:pserver:anosous-
embranchementsymous@cvs.sourceforge.net:/cvsroot/ocera login
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/ocera co ocera
```

This will create the ocera structure in /usr/share/OCERA/ocera

11.3.2) Compile the kernel

Refer to the Chapter 2 to learn more about the configuration options.

```
cd ocera
make xconfig
```

> **take care to not use the local APIC if you use a single board system because there is still a bug in the configuration's options.**
> **You must also enable the VT, Virtual Terminal support in the character devices drivers and VGA text console.**

> **If you do not want to build the documentation and the applications, you can comment out the entries in the makefile.**

Then:

```
make
```

This should give you the following directories in the ocera-1.0.0 directory:

- target-i386
  - boot
    - vmlinuz-2.4.18-ocera-1.0.0
    - System.map-2.4.18-ocera-1.0.0
  - dev
  - etc
    - rc.d/init.d/rtlinux
  - lib
    - modules/2.4.18-ocera-1.0.0 with all drivers and RTLinux modules
  - usr
    - lib with orte and posix development libraries
    - bin with the ORTE binaries and tests programs
    - include
    - rtlinux with RTLinux demo and tests programs

At this point you may choose between:

- Installing your ocera kernel on you system, then you can act as with any standard linux kernel.
- Installing your ocera kernel as an embedded system, using emdebsys or a simple busybox.

11.3.3) Installing OCERA in a busybox environment

a) retrieve BusyBox and syslinux

If you have an OCERA CDROM, you will have all tools on the CDROM and if you installed an OCERA development environment from a CDROM you must have the busybox and syslinux tools already installed.
If not, you must download it from the network with something like:

```
wget http://busybox.net/downloads/busybox-1.00-pre3.tar.bz2
```

```
wget http://syslinux.zytor.com/download/syslinux-2.06.tar.gz
```

Then retrieve a basic template file system from mnis:

```
wget http://www.mnis.fr/download/basiclinuxfs-0.1.tgz
```

b) be sure to use the proper development tools

use ***dpkg -l*** to verify the versions:

Program	Version	definition
gcc	2.95.4-14	The GNU C compiler.
Bin86	0.16.0-2	16-bit assembler and loader
make	3.79.1-14	The GNU version of the "make" utility.
Autoconf	2.57-1jlb	automatic configure script builder
automake	1.4-p4-1.1	A tool for generating GNU Standards-compliant

c) build the tools

```
tar jxvf busybox-1.00-pre3.tar.bz2
tar zxvf syslinux-2.06.tar.gz

cd syslinux-2.06
make all

cd busybox-1.00-pre3
make menuconfig
make dep
make
make install
```

d) make the target file system:

```
mkdir TARGET
cd TARGET
tar zxvf basiclinuxfs-0.1.tgz
( cd ../busybox-1.00-pre3/_install; tar cf - ) | tar xvf -
cp ../target-i386/boot/System.map-2.4.18-ocera-1.0.0 boot
cp ../target-i386/boot/vmlinuz-2.4.18-ocera-1.0.0 boot
cp -r ../target-i386/lib/modules lib
```

```
Change the configuration files in TARGET/etc to fit your needs
Make the root file system from the TARGET directory:
mke2fs /dev/ram0
mount /dev/ram0 /mnt
(cd TARGET ; tar cf - *) | (cd /mnt ; tar xvf -)
umount /mnt
dd if=/dev/ram0 of=root
gzip root
```

e) make the boot system: exemple: a CDROM

```
mkdir ISO
cp /usr/src/linux/arch/i386/boot/bzImage  ISO/ocera
rdev /dev/ram0 ISO/ocera
cp root ISO
cp isolinux-2.06/isolinux.bin ISO
cp isolinux-2.06/sample/syslogo.lss ISO
```

put something in ISO/boot.msg like:

```
^L
^Xsplash.lss


           ^O07OCERA STANDALONE CD^O07
```

to change the start image, the splash, use a png file in 639x320x4 format.
edit ISO/isolinux.cfg
default ocera

```
prompt 1
timeout 600
display boot.msg
label ocera
  kernel ocera
  append initrd=root.gz
```

Build the image with:

```
mkisofs -R -b isolinux.bin  -no-emul-boot -boot-load-size 4 -boot-info-table -o ocera.iso ISO
cdrecord dev=0,0,0 ocera.iso
```

Then booting on the CD will install the root file system in memory (/dev/ram0) and you can go on by testing your application.

11.4) Running a hard real-time application

To run a hard real-time application you need to insert the RTLinux-GPL modules into the kernel.
You will see more details on choosing the right modules in the next chapters describing in details the components.
What you need to know now is

- where are the modules
- what they do in short
- how to run a test application

The modules real-time modules are in the **/lib/modules/ocera-1.1/misc** directory.

Module name	***Component***	***Description***
cbs_sched.o	Quality Of Service	The RTLinux CBS scheduler is needed if you want to use this scheduling algorithm for your applications.
ftappli.o	Fault Tolerance	
ftappmonctrl.o	Fault Tolerance	
mbuff.o	memory	
qmgr_sched.o	scheduling	
rtl.o	RTLINUX CORE	
rtl_fifo.o	RTLINUX CORE	
rtl_ktrace.o	RTLINUX CORE	
rtl_malloc.o	RTLINUX CORE	
rtl_mqueue.o	RTLINUX CORE	
rtl_posixio.o	RTLINUX CORE	

Module name	***Component***	***Description***
rtl_sched.o	RTLINUX CORE	
rtl_time.o	RTLINUX CORE	

APENDICES

12) Hardware and software issues

Pierre Morel - MNIS

The are some hardware and software issue to consider when realizing a real-time system with RTLinux.
In particular, using a standard low cost PC may introduce some constraints due to optimization that have been made with other goal than real-time.
Some Hardware change dynamically the way the processor or the system BUS operates and some software change hardware configuration.

12.1) New modules and Video

You must take care when using a module that you did not compile with the patched Linux tree you got from the OCERA CVS or from an OCERA ISO image.

The problem comes from un-patched CLI/STI instructions that may affect the real-time performances of the system.
The same problem occurs with the XF86 servers because some of them will use CLI/STI to optimize the code.
To now if a module or a X11 server uses these instructions, you can use the following command:

```
objdump -d module.o | grep cli
```

12.2) NTP

The Network Time Protocol is used to synchronize the system clock with an external system supposed to be accurate over the network.
To achieve its work, some implementations of NTP may slidly change the frequency of the board to counter the derive of the clock.
While this is perfect for a time sharing system with 10ms slices, it has main drawbacks on a real-time system with hundreds of nanoseconds accuracy.
You can still use NTP if you makes sure it does not change the frequency but makes time jumps. These jumps are made on the GLOBAL CLOCK used by the Linux system, while the RTLinux part will have access to other unchanged clocks like the REALTIME CLOCK or the MONOTONIC CLOCK

12.3) Floppy and ISA

The ISA bus may introduce some problems in the system in case you use it. The ISA BUS bridge slow the PCI BUS speed to allow transfers to the ISA BUS.
The floppy on an IBM PC computer may reduce the speed of the BUS when accessed.
The best ist not to use these hardware.

12.4) Power saving

APM systems usually change the CPU clock to reduce the power usage of the system. This is of course incompatible with real-time accuracy.
It is generally better not to use the power saving mechanism and if needed to reduce the clock speed at boot time.

12.5) System Management Mode

Some board use the Intel's System Management Mode, on which upon receive of a System Management Interrupt, the processor will enter a slow mode or even stop. Of course, this feature must be disabled.

13) Qualifying an OCERA system

By
Stanislav Benes - Unicontrol

13.1) Software Criticality Level Definitions

We can generally divide software for control systems according to the Software Criticality Level, which is an assessment how dangerous can be an anomalous behavior of the software for health or lives of people.

- Level A: Software error can cause death of many people.
- Level B: Software error can cause death of a small number of people.
- Level C: Software error can cause discomfort, possibly including injuries.
- Level D: Sofware error can cause discomfort.
- Level E: Sofware error has no effect.

13.2) How to keep the criticality level when developing an application

13.2.1) Software verification effort

Software Criticality Level affects effort which is necessary for verification of the software.
In case of critical application (levels A, B, C) a certification authority usually gives an approval to using of a software in an application. The certification authority can state conditions or standards, which the software should comply with. The applicant for an approval to using of a software in a critical application should expect following types of guidelines both for OCERA components and for application components:

- Level A: Every software requirement (i.e. every feature stated in Programmer's Guide) and software design requirement (i.e. every feature stated in Component Design Description document) should be verified, every conditional statement in source code should be verified in relationship with other conditional statements in the same module. The results of the verification should be documented.
- Level B: Every software requirement (i.e. every feature stated in Programmer's Guide) and software design requirement (i.e. every feature stated in Component Design Description document) should be verified, every conditional statement in source code should be verified. The results of the verification should be documented.
- Level C: Every software requirement (i.e. every feature stated in Programmer's Guide) should be verified. The results of the verification should be documented.
- Level D: Every verified software requirement (i.e. feature stated in Programmer's Guide) should be documented.

- Level E: No guidelines for software verification are stated.

13.2.2) Rules for software documentation

Software Criticality Level affects detailing and an approval procedure which is necessary for the documentation of the software. In case of critical application the certification authority can state conditions or standards, which the software documentation should comply with.
The applicant for an approval to using of a software in a critical application should expect following types of guidelines both for documentation of OCERA components and for application components:

- Level A: Every software or documentation change should be approved by a worker responsible for software approval.
- Level B: Every software or documentation change should be approved by a worker responsible for software approval.
- Level C: Every change in software requirements, source code, executable code, software configuration index, software life cycle environment configuration index (i.e. compilers, linkers etc.) and SW accomplishment summary should be approved by a worker responsible for software approval.
- Level D: Every change in software requirements, source code, executable code, software configuration index and SW accomplishment summary should be approved by a worker responsible for software approval.
- Level E: No guidelines for software documentation are stated.

13.2.3) Assessment of the level of criticality of OCERA components

Ocera components are generally designed for applications in level D. Concrete assessment of every Ocera component will be stated in document D12.4 "Final Assesment and Evaluation Report", including usability from criticality level point of view. Users can turn to Ocera consortium if they need to use an Ocera component in an application with a higher level of criticality.

14) Performance

By
Nobody - Nowhere

14.1) Hard Realtime performances

Using OCERA, you can expect the following performances:

Table 5-1. Hard Realtime performances

type of measure done	result	commentary
Interrupt latency		
Task switch latency		
Event latency		
Barrier latency		
Number of threads		

CBS reservation

14.2) Soft Realtime performances

Using OCERA, you can expect the following performances:

Table 5-2. Soft Realtime performances

type of measure done	result	commentary
Interrupt latency		
Task switch latency		
Semaphore latency		
Number of tasks		
CBS reservation		

14.3) General performance

Table 5-3. General performances

type of measure done	result	commentary
Filesystem size		
File size		
Realtime Ethernet bandwidth		
CAN BUS bandwidth		

14.4) Footprint

Table 5-4. footprint

type of measure done	result	commentary
Stand alone RTLinux	100k byte	This is achieved by using the Stand Alone RTLinux with all components
standard Embedded system	4M byte	
complete system footprint	???very large???	If you put every services of Linux here like SQL databases, web servers, security and auditing tools, you can have a quite big system.

15) Applications

By
Cristina Sandoval – Visual Tools (VTWV application)

15.1) VTWV. iPAQ Application

15.1.1) Application Overview

VTWV stands for Visual Tools Wireless Viewer. It is a video player for handhelds that receives wirelessly and presents video streams transmitted by one (or several) Visual Tools DVRTs (Digital Video Recorders and Transmitters). It is able to find out which DVRTs are present in the LAN and request them to stream video from any of its cameras.
VTWV will also connect to Visual Tools PeCo servers. The PeCo servers are people counters that use state-of-the-art computer vision algorithms to count people in a video sequence taken with a zenital camera.
It is developed in the Ewe programming environment, a Java VM based on Waba that is small, fast, works in Windows, Linux for x86, Pocket PC and Linux for Arm.

15.1.2) Application Requirements

This is the basic list of requirements the VTWV must fulfill:

- Hardware Platforms

This application will be developed for Compaq/hp iPAQs that come with an Intel Arm CPU.
This will be the basic target platform. However we will also develop the application so it works on x86 platforms. The iPAQ must have built-in WiFi or will be equipped with a Compact Flash 802.11b Wireless LAN Card. "

- Integration with Visual Tools DVRT products
  As stated before, this video viewer will be integrated with Visual Tools DVRTs. It will be able to find DVRTs in the LAN, request and receive video from them. The basic topology will be a managed (also called BSS) wireless LAN were several DVRTs are contected (wired) to an Access Point and the iPAQ wirelessly connected to the Access Point

15.1.3) How to compile the application and download it to the iPAQ.

First of all, you will need to download and install the Ewe Virtual Machine and Development Environment version for your platforms that you can get from http://www.ewesoft.com/Downloads/Downloads.html and follow installation steps.

In the OCERA CVS multimedia/vtwv directory you will find the Java sources as well as the Ewe extension to decode MPEG streams using the ffmpeg library.
Some of the other files you will find here are:

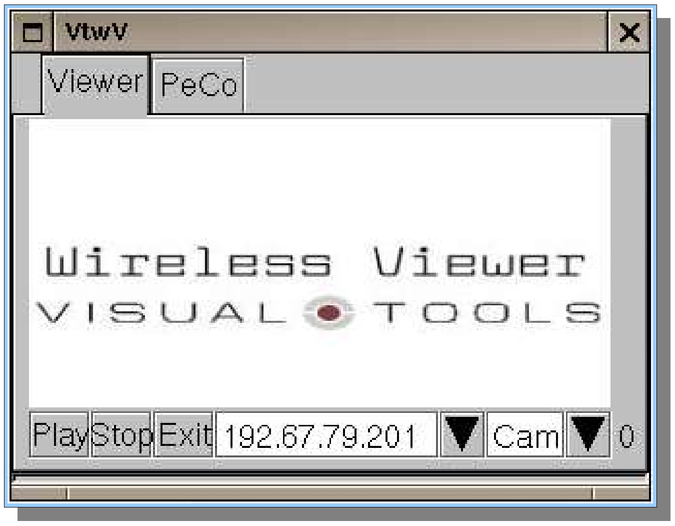
vtwv.desktop : File to add an entry in the Opie desktop of the Linux iPAQ
vtwv.jnf : Jewel file needed to build the .ewe file (kind of .jar file)
vtwv.ewe : Kind of .jar file that contains all Java classes, images, libraries,... needed by VTWV

To compile the VTWV application for your PDA you should run the Jewel.ewe application:

```
ewe $JEWEL_PATH/Jewel.ewe vtwv.jnf
```

This will create in the ./classes directory a vtwv.ewe portable file that you will be able to run in your PDA typing a command line like the following:
ewe vtwv.ewe -a DVRT1 -A DVRT2 -c MPEG -W 300 -H 150

where:
DVRT1: IP address of one of the DVRT that is transmitting a video stream
DVRT2: IP address of the DVRT that is giving people counting information (not needed).
This will open the vtwv application where the video stream from the selected server and camera will be displayed.


VtwV
Viewer
PeCo
Wireless Viewer
VISUAL TOOLS
Play
Stop
Exit
192.67.79.201
Cam
0