



WP10 - User Guide

Deliverable D10.3 - User Guide

by Pierre Morel, Miguel Masmano-Tello, Agnes Lanusse, Ismael Ripoll, and Stanislav Benes

Published June 2003

Copyright © 2003 by Ocera

Table of Contents

Chapter : 1	Overview.....	6
1.1	Short description of OCERA.....	6
1.2	Intended Audience.....	6
1.3	What will you find in the User's Guide.....	7
1.4	History of the OCERA project.....	8
1.5	And what is in OCERA?.....	9
1.5.1	historical choices.....	9
1.5.2	Quick overview of the architecture.....	9
1.5.3	The components.....	10
1.6	Comparison with other Real-time Operating Systems.....	11
1.7	What's next.....	12
Chapter : 2	Getting started.....	13
2.1	Supported environment.....	13
2.2	Getting all from Internet.....	13
2.3	Installing from a CD-ROM.....	14
2.3.1	Getting a CDRom.....	14
2.3.2	Installing the CD-ROM.....	14
2.3.3	The documentation on the CD-ROM.....	14
2.4	Installing from the network.....	14
2.4.1	OCERA development environment.....	14
2.4.2	Installing independent components as binaries.....	14
2.4.3	The documentation.....	15
2.5	What's next?.....	15
Chapter : 3	Building an OCERA system.....	16
3.1	Choosing the components.....	16
3.1.1	POSIX components and scheduling.....	17
3.1.2	Core features.....	17
3.1.3	Quality of services.....	17
3.1.4	Fault Tolerance.....	17
3.1.5	ORTE.....	17
3.1.6	Can devices.....	17

3.1.7 Ada.....	18
3.1.8 RtxFS.....	18
3.1.9 SA-RTLinux.....	18
3.2 Configuration of components, kernel and libraries.....	18
3.2.1 Quality Of Service.....	20
3.2.2 Fault Tolerance.....	22
3.2.3 CAN.....	25
3.2.4 ORTE.....	30
3.2.5 RTLinux.....	32
3.2.6 Linux.....	37
3.3 Building the kernel and components.....	38
3.3.1 make.....	38
3.3.2 The target directory structure.....	38
3.4 Building the tools.....	38
3.5 Compiling a custom application	39
3.5.1 Linux Application.....	39
3.5.2 RTLinux Application.....	39
3.6 Installing a training systems.....	40
3.7 Installing an embedded systems.....	40
3.7.1 Retrieve the sources.....	41
3.7.2 Compile the kernel.....	41
3.7.3 Installing OCERA using emdebsys.....	42
3.7.4 Installing OCERA in a busybox environment.....	42
3.8 Running a hard real-time application.....	44
Chapter : 4 POSIX components.....	46
Chapter : 5 Quality of service.....	47
Chapter : 6 ORTE.....	48
Chapter : 7 CAN Devices.....	49
Chapter : 8 Fault-Tolerance.....	50
8.1 Degraded Mode Management.....	50
8.1.1 Design choices.....	50

8.1.2	How to use it.....	51
8.1.3	Degraded Mode management: architecture overview.....	59
Chapter : 9	Ada.....	61
9.1	Overview.....	61
9.2	ADA RT implementation in OCERA: RTLGnat.....	61
9.2.1	Presentation of RTLGnat.....	61
9.2.2	Features.....	61
9.2.3	Requirements.....	61
9.2.4	Real-Time performance.....	62
9.2.5	Known problems and limitations.....	62
9.2.6	Compilation notes.....	62
9.2.7	Version history.....	62
9.2.8	Main sites.....	63
9.2.9	ADA Library.....	63
9.2.10	Compiling.....	63
9.2.11	Sample program.....	63
9.3	. What's next.....	64
Chapter : 10	Chapter 11. Debugging and tracing.....	65
10.1	Debugger.....	65
10.2	Analyzers.....	65
10.2.1	Supervision tools.....	65
10.2.2	Setting POSIX traces.....	66
10.3	What's next.....	66
Chapter : 11	RtxFS.....	67
Chapter : 12	SA-RTLinux.....	68
Chapter : 13	Cross compilation.....	69
Chapter : 14	Qualifying an OCERA system	70
14.1	Software Criticality Level Definitions.....	70
14.2	How to keep the criticality level when developing an application.....	70

14.2.1	Software verification effort.....	70
14.2.2	Rules for software documentation.....	71
14.2.3	Assessment of the level of criticality of OCERA components.....	71
Chapter : 15	Performance.....	72
15.1	Hard Realtime performances.....	72
15.2	Soft Realtime performances.....	72
15.3	General performance.....	73
15.4	Footprint.....	74
15.5	What's next.....	74
Chapter : 16	Appendix A. control command with CAN.....	75
Chapter : 17	Appendix B. Real Time Ethernet.....	76
Chapter : 18	Appendix C. Robotic application.....	77
Chapter : 19	Appendix D. Streaming video.....	78
Chapter : 20	Glossary.....	79
Chapter : 21	Index.....	80
Chapter : 22	Bibliography.....	81

Chapter : 1 Overview

This is the OCERA User Guide. We will try through this document to help you understanding what is OCERA and how to use it.

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 Short description of OCERA

OCERA stands for **Open Components for Embedded Real time 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 with or without the RTLinux-GPL patch.

The component are Open Source and most of the components licenses are GPL¹.

1.2 Intended Audience

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

This guide is also intended for **teachers** who want to rapidly build a real-time plate-form for training **students**. In this case the teachers will also have interest in the OCERA document named *TRAINING DOCUMENTATION AND CASE STUDIES*, where they will find ready to use training examples.

To understand how OCERA kernel and components 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 more skills. First in a development language, either C, C++ or Ada depending on the developments you intend to do and which components you intend to use.

To use the OCERA components you may not need experience but you will need to study their interfaces and the interactions between them:

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.

Fault Tolerance Systems, is greatly related with scheduling components and Quality Of Service.

Quality Of Service component is quite intuitive at first sight but you will see that the usage is more complex in association with Fault Tolerance, or depending on the way you will handle exceptions you will need a good knowledge of scheduling in both real-time and time sharing spaces.

CAN devices is a stand alone component, if you use it, it is assumed you already have experience with CAN buses and CAN devices.

ORTE is also a stand alone component, you can use it apart if you want. But what is the interest of a Real Time Ethernet without a real time application above. So you will certainly use it in conjunction with at least the Real Time POSIX components.

Scheduling components is the name we defined at the origin for all components in the real time kernel. I prefer to separate between **Real Time POSIX components** like POSIX threads, POSIX signals, POSIX barriers etc; on one hand and **Scheduling components**, like EDF and CBS schedulers, application scheduler RTL-GNAT on the other hand.

The port of GNAT, Gnu **Ada**, to the real time kernel, RTL-GNAT is quite independent in that you do not need any knowledge of the underlying components to use RTL-GNAT. You need Ada knowledge and quite no Linux knowledge is required to start an Ada application.

This guide will give to you a good overview of different part of OCERA like system interfaces, components and development tools but if you need details on the programming interface for hard and soft real-time applications you will have to refer to the *OCERA PROGRAMMER'S GUIDE*.

1.3 What will you find in the User'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 this chapter presenting OCERA, you will find the following chapters in this guide:

- **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 (UC)
- real time Ethernet provided by (CTU)
- two level application provided by (CEA)
- streaming video provided by (VT)

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

1.4 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 OCERA uses the LINUX/RTLinux combination, which already exists and enhanced it by:

- rewriting some of the RTLinux primitives to correct it or to make them POSIX compliant.
- adding new POSIX components to RTLinux, like new schedulers (CBS), barriers, timers and more
- enhancing the Linux scheduler to add CBS scheduling
- adding tracing tools 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 RTLinux configuration and generation with Linux configuration and generation in a single CML2 configuration using emdebsys generation tools.

1.5 And what is in OCERA?

OCERA stands for

- **O**pen source
- **C**omponents for
- **E**MBEDDED
- **R**Real time system
- **A**pplications

1.5.1 historical choices

At the beginning of the project we studied real-time operating systems to see what would be of interest to have in a optimal real-time operating system. Since we did not want to reinvent the wheels and our purpose is to enhance OpenSource Real time Operating system, we next, studied two open source real-time system that could be appropriate, RTLinux and the derivative, RTAI. At this time, we did not use RTAI because we found it too far from our goals and our choice went on RTLinux-GPL. You can read the details in the document D1.1, and RTLinux versus RTAI that you must be able to download from OCERA web site.

1.5.2 Quick overview of the architecture

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. It is a Hard real-time level with interrupt latency and thread switch latency in the order of 10 micro seconds.
- A **time sharing operating system**, the Linux level, having the full functionalities of the original Linux operating system.

Both operating systems co-operate in two 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. RTLinux-GPL, calls the Linux handlers with a so called *soft IRQ* as soon as nothing more is to be done at real-time level.
- **Real-time fifo**: if a real-time thread and a Linux task want exchange data, they must do it through *real-time fifo*, this is actually the only 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.

We found of great interest to have the possibility to add a soft real-time level to Linux, using and enhancing the Andrew Morton LOW LATENCY patch, this is the first brick to provide *Soft real-time* and *Quality 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.

1.5.3 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 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:

I Communication

Communication components may be an exception in that they can be implemented directly under Linux, without any other components. But if they are integrated in RTLinux, they must use the new POSIX Interface.

a 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.

b 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 transfers time over Ethernet to be predictable.

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

II Fault Tolerance

Fault tolerance may use every other components . It uses at least the new POSIX interface and must work 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: Fault Tolerance

III 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 preemptive. To do this we have to use the preemptive patch of XXXX for Linux.
- 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 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 example for this is a real time video streaming application.

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

IV RTLinux components

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

By the way OCERA integrated new components: a UDP stack used by the RTLinux-GPL implementation of ORTE, a Real time IDE disk interface used by the tracing tools, Ada runtime and a Stand Alone RTLinux-GPL

You will have a brief introduction to these components here in this chapter but we will come back to these components in dedicated chapters at the end of the User's Guide.

a Scheduling components

b POSIX Interface

c UDP stack

d IDE disk interface

e Ada runtime

f Stand Alone RTLinux

1.6 Comparison with other Real-time Operating Systems

The OCERA deliverable D1.1 presents a study of majors real-time operating systems. You will find there a deep study of the benefit of one or the other operating systems.

As a sum up we can say:

OCERA is a good alternative to standard real-time systems like VRTX, OSEK or QNX and it is OpenSource and GPL.

OCERA is a better alternative than RTLinux or RTAI alone because it offers a POSIX API , a great documentation and integrate development tools and new components like:

- an integration environment.
- development tools to produce UML definitions.
- really POSIX API
- a lot of useful documentation, User Guide, Programmer's guide, white papers, manual pages, and different articles.
- Quality of service
- Fault tolerance
- Real time Ethernet
- Real time tracing tools

OCERA offers a different approach than JALUNA, through the integration with the Linux kernel. You can with OCERA directly make use of the Linux drivers from the RTLinux context by using the LinRTL synchronization interface developed in OCERA.

This allows Real time tasks to use the network, the Hard-drives and other drivers like USB, without to have to rewrite a driver. And this feature allow you to use all existing Linux drivers.

Another advantage of OCERA versus JALUNA is that OCERA integrate Quality Of Service, with the use of the CBS scheduler and Bandwidth reservation at the Hard Real time level (RTLinux Real time task) and at the Soft Real time level (Linux Real time tasks)

The advantage of JALUNA on OCERA is the possibility to make a warm restart of the Linux actor and this is important in the case of HA systems. OCERA as JALUNA can do warm restart of the Hard Real time tasks and allows reconfiguration of the tasks and replicas.

1.7 What's next

Now that you have an Idea on what OCERA can be used for, you will learn how to get started with the OCERA system by downloading the documentation and the software.

Chapter : 2 Getting started

2.1 Supported environment

The OCERA consortium chose 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:

C Compiler	GCC-2.95.4
Graphic environment	Qt-3.1.2
Make	
Autoconf	
Automake	
Ada Compiler	Gnat-3.14

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 ti install OCERA development CD-ROM.

2.2 Getting all from Internet

Of course to begin to work with OCERA you will need to get the software and the documentation.

You can get everything from the OCERA web site at <http://www.ocera.org/> sources tarball, specific snapshots, patches and you can also get helped through the OCERA SourceForge site at <http://ocera.sourceforge.net/> where you can find mailing lists, bug trackers and where you can browse the development CVS or download the last development version from the CVS.

You can download the Debian distribution from the DEBIAN web site at <http://www.debian.org/> .

You can download the right Ada compiler from the GNAT FTP site at <ftp://cs.nyu.edu/pub/gnat/3.14p/> .

It is out of our scope to explain to you how to install a Linux distribution and you will need to follow the documentation on each site to install the different software needed.

We will provide you the instructions for installing the development environment from an OCERA CDROM in the next chapter.

2.3 Installing from a CD-ROM

2.3.1 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 or simply download an ISO image.

This is certainly the best way to get started since you will get a complete and tested environment for the development of your embedded system.

2.3.2 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.

2.3.3 The documentation on the CD-ROM

Another advantage of the OCERA CD-ROM is that you get all the documentation ready to use on the CD-ROM.

2.4 Installing from the network

2.4.1 OCERA development environment

If you already 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.

These ocera-dev packages dependencies are:

```
lib-qt3
qt3-dev-tools
qt3-tools
rtlnat
orte
lincan
```

2.4.2 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
```

2.4.3 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

2.5 What's next?

Now that you have got the software and the documentation you are ready to make your first steps with OCERA.

Chapter : 3 Building an OCERA system.

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:

- 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.
- Configuration of components, kernel and libraries
There you will see how to compile the kernel, we focus on the configuration tool.
- Building the tools
The tools are independent of the kernel and are compiled separately, like debugger, tracing tools, analysers.
- Compiling a custom application
We will see some little applications, examples and the compilation's directives.
- Installing a training systems
This is certainly the first way you will install OCERA.
- 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.

3.1 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.

3.1.1 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.

3.1.2 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.

3.1.3 Quality of services

You will need the Quality Of Service component if you intend to develop an application needing to have a Constant CPU Bandwidth allocated.

The best example may be an video application for witch you do not want the system to steal time.

3.1.4 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.

3.1.5 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.

3.1.6 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.



3.1.7 Ada

The is a real time Ada runtime library and the real time Ada compiler.

3.1.8 RtxFS

The Real Time File System and IDE interface

3.1.9 SA-RTLlinux

The Stand Alone Real Time Linux.

3.2 *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 main directory of the OCERA tree, it should be `/usr/src/ocera` if you have installed your system with an OCERA ISO image or in the sub-directory `ocera-1.0.0` if you downloaded the tarball of the version 1.0.0 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.

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

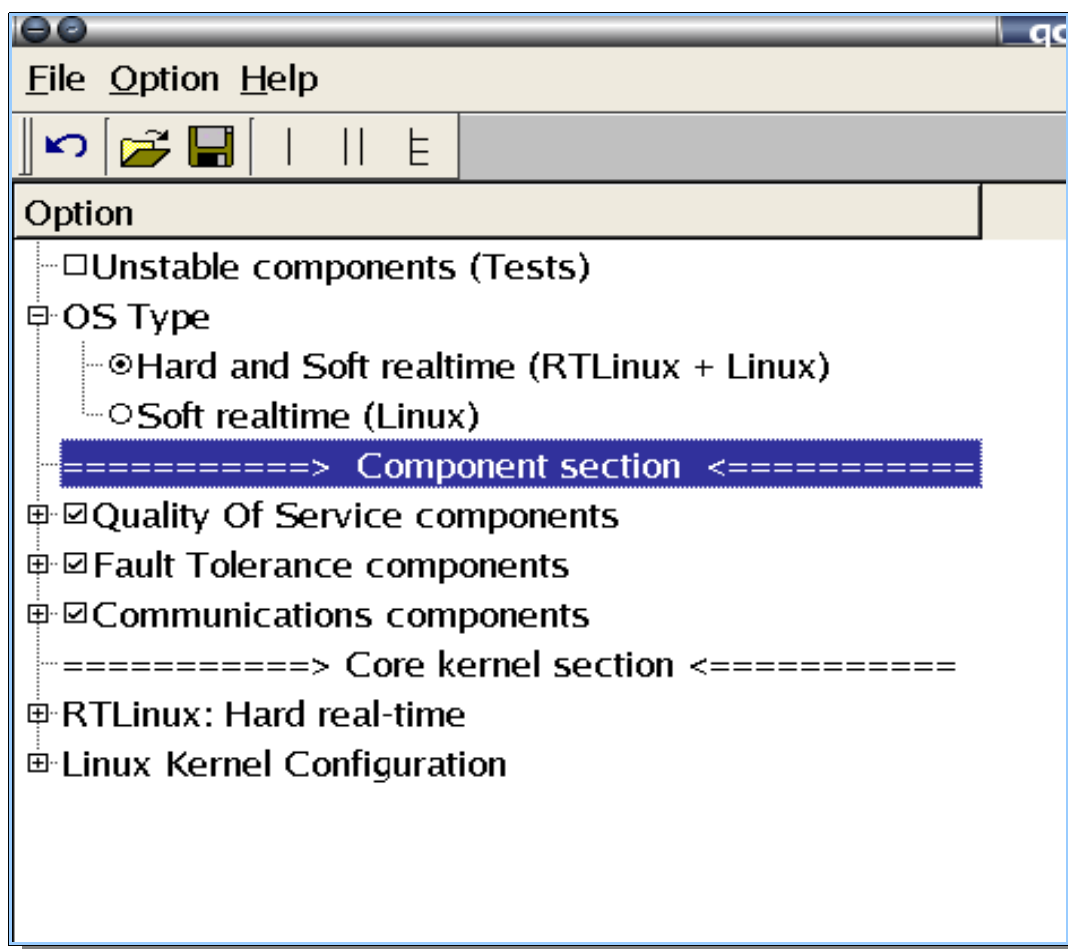


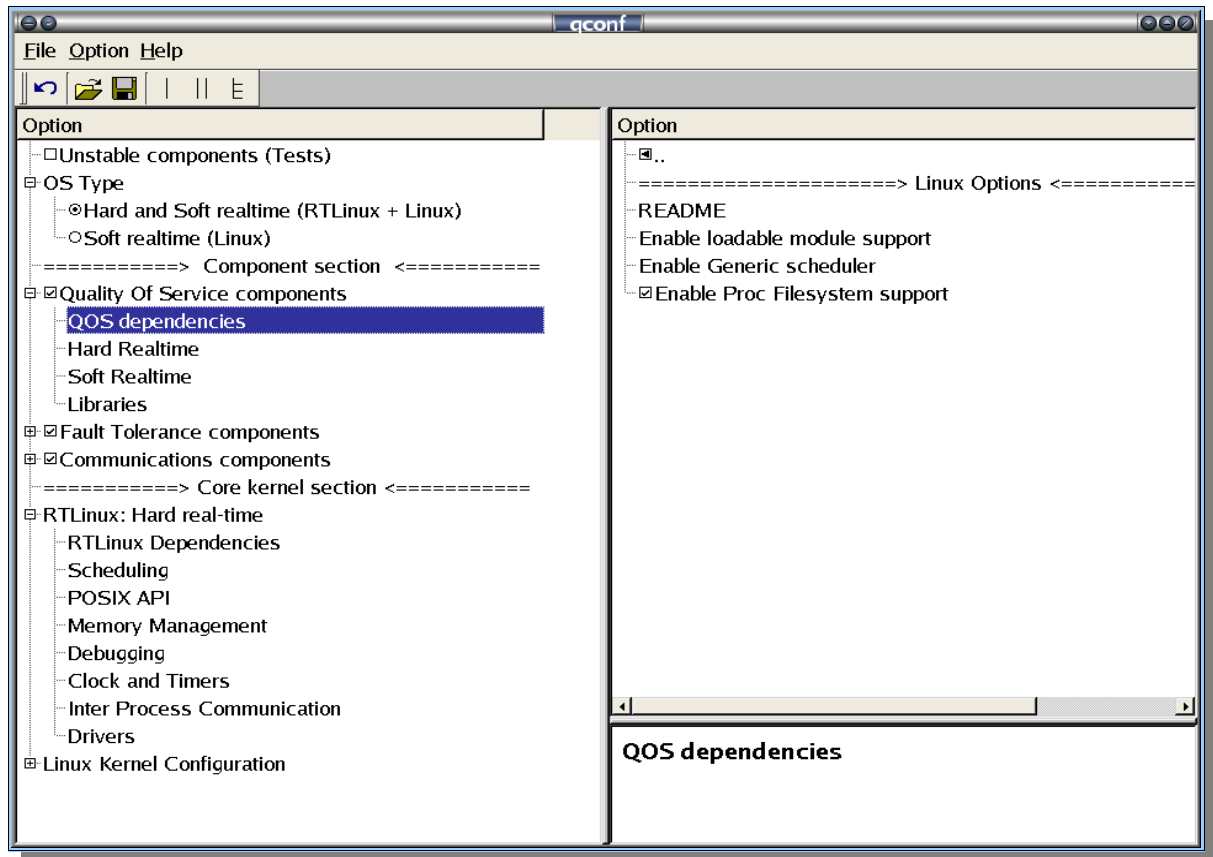
Illustration 1 General Configuration

- 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:

3.2.1 Quality Of Service

I Dependencies



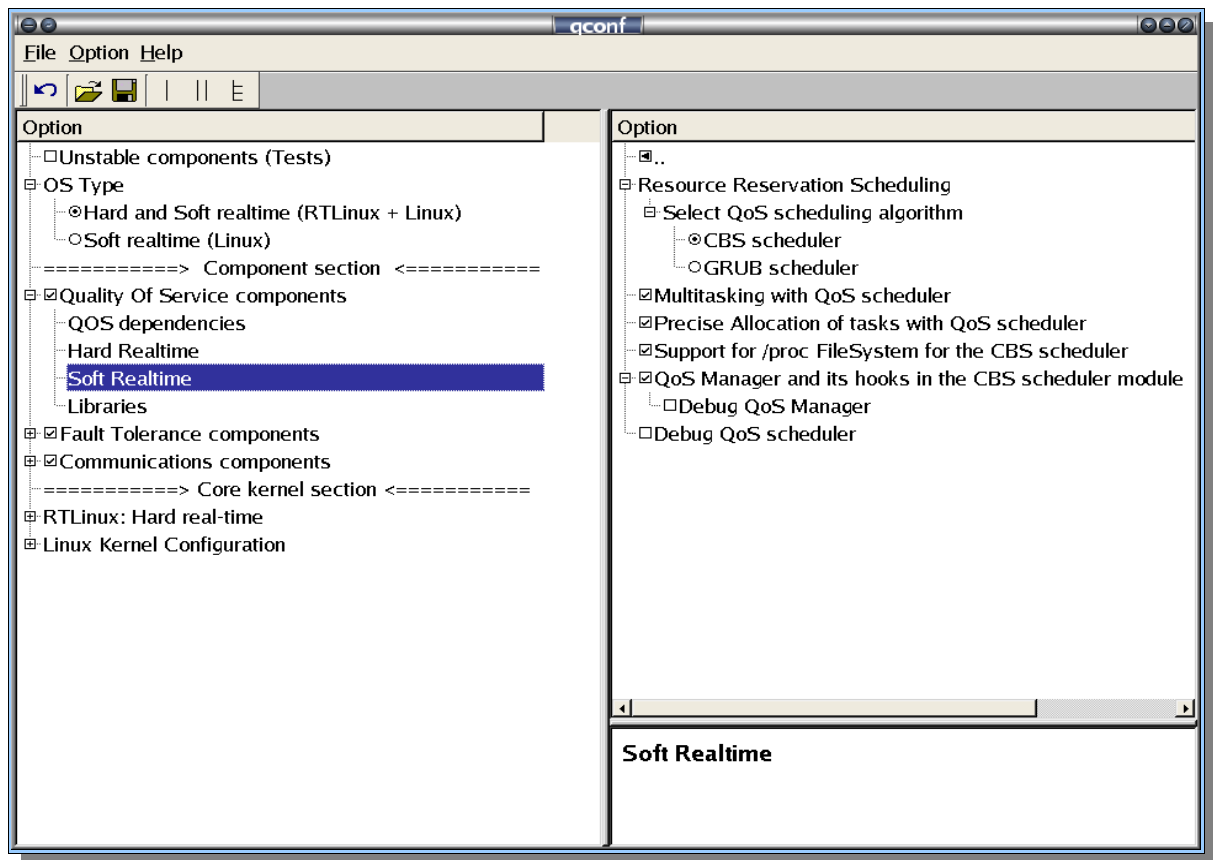
The first sub-menu in Quality Of Service is the dependencies menu, you will find such a menu at the first place in all components menu, as the name let think, this point out the dependencies that must be resolved to be able to compile the Quality Of Service components.

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

II 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.



a Resource 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

b 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

c 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.

d Support for the /proc filesystem

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

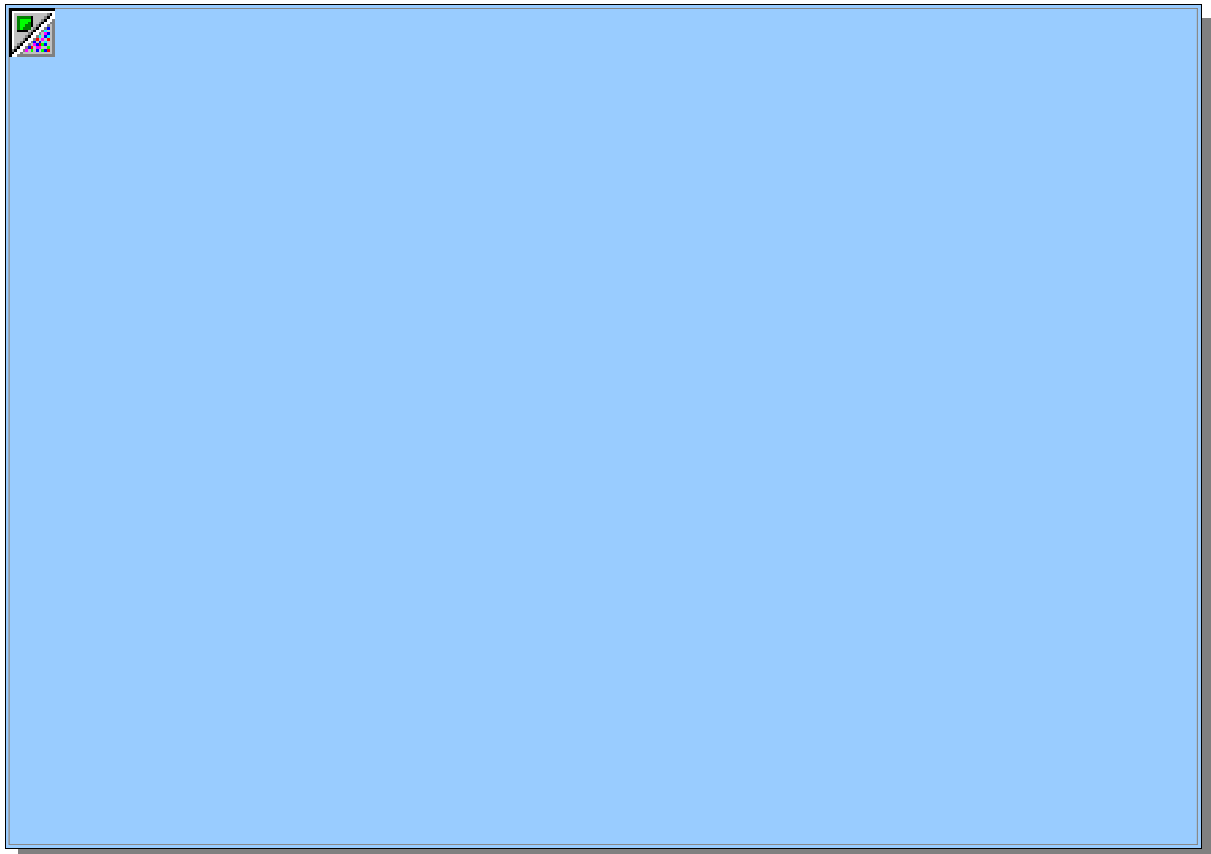
e QoS Manager and its hook in the CBS scheduler module

Kernel module that allow to manage the CBS scheduler

f Debug QoS scheduler

Enable printk kernel debug for QoS Manager module

III Libraries



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

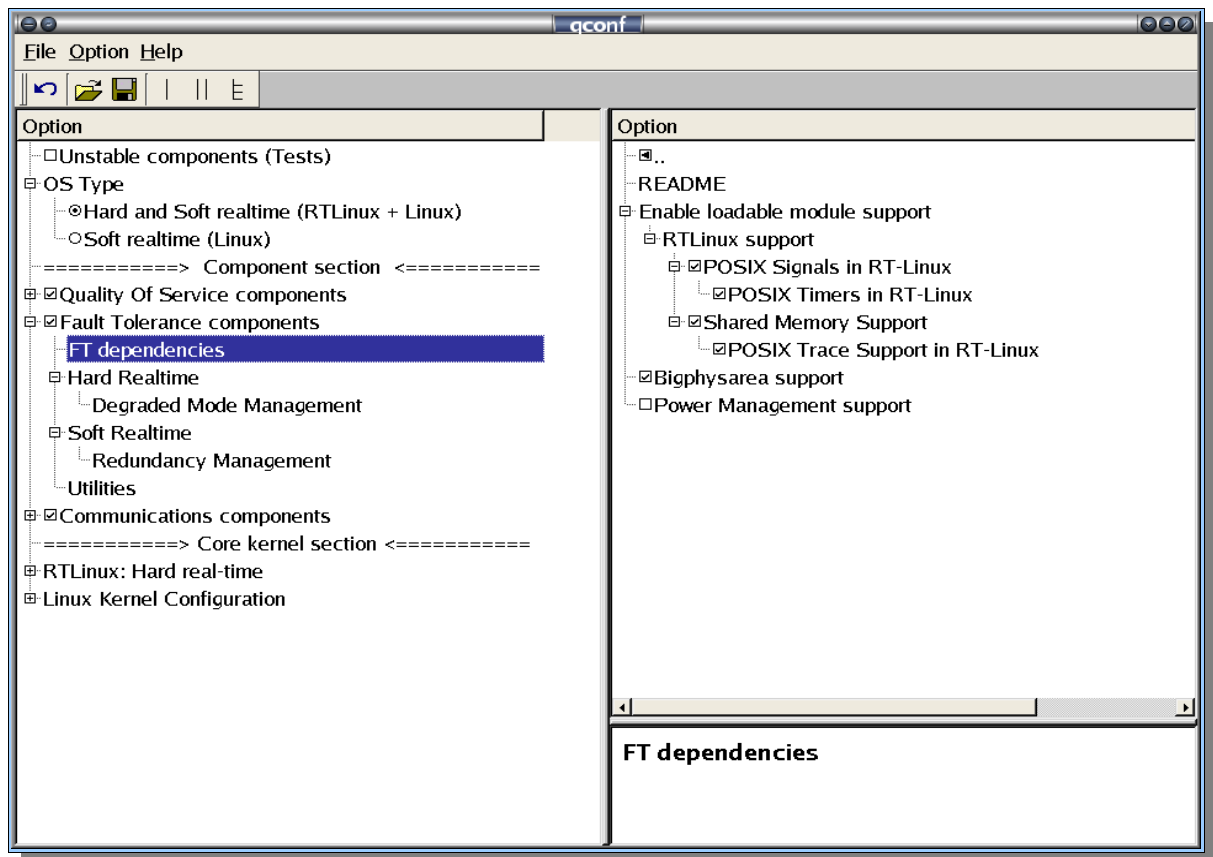
3.2.2 Fault Tolerance

I Dependencies

The Fault Tolerance component has dependencies with

- POSIX timers
- POSIX signals
- POSIX traces
- Shared memory allocator that in turn depends on the BIGPHYSAREA.

of the Hard real-time kernel. These entries must be enable to use Fault Tolerance.



As for other components, through the use of RTLinux-GPL, the modules support must be enabled and as we are working with hard real-time, Power Management can only be a problem to us and must be disabled.

II Hard Real-Time

The hard real-time functionalities of the Fault Tolerance component handles the *Degraded Mode Management*.

a FT Controller

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.

b FT Application Monitor

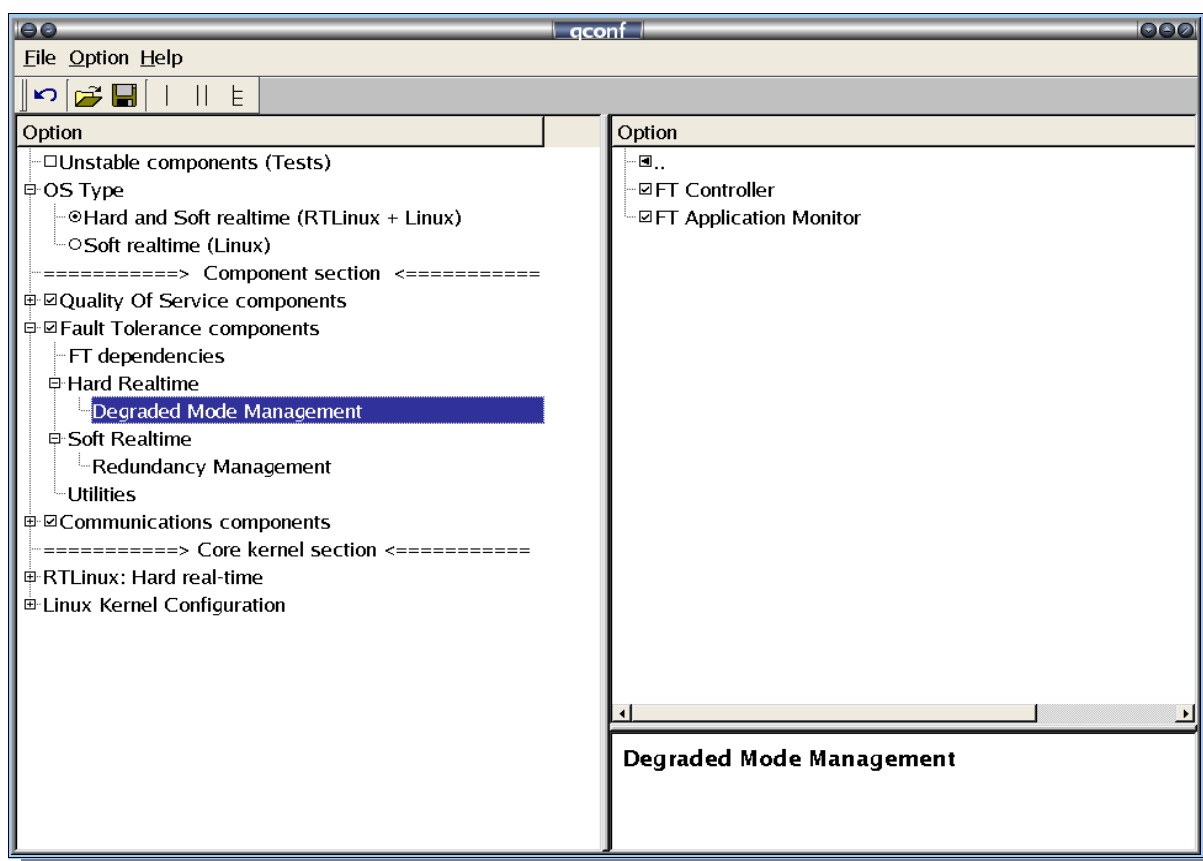
FT Application Monitor is one of the two modules that insure Degraded Mode Management.

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.



III Soft Real-Time

a Redundancy Management

Redundancy Management is not yet available. Say no here

IV Utilities

The Fault-Tolerance Building Tool, called ftbuilder is a TCL/TK 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.

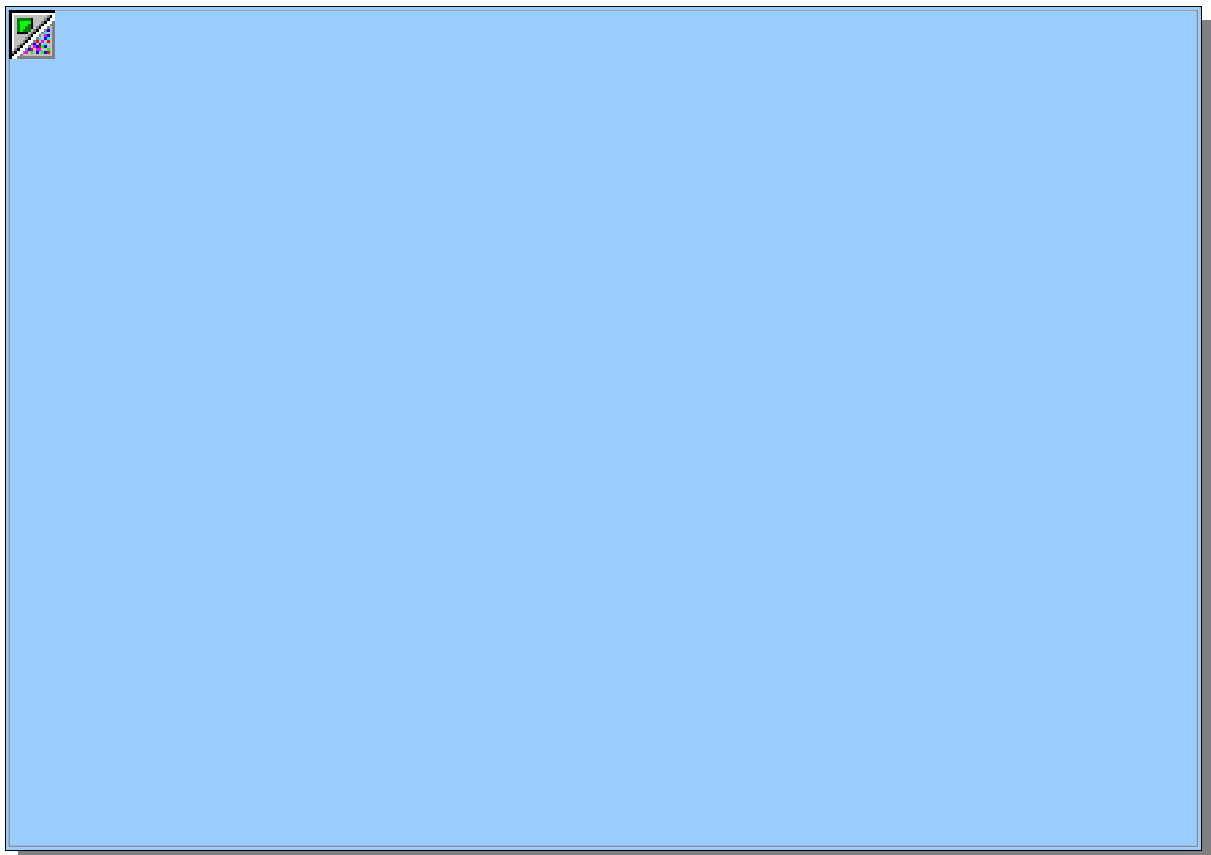
The ftbuilder is not a loadable module, it is provided as a directory that can be copied to user home environment to build ft_applications (see documentation for details).

It is located under OCERA_DIR/components/ft.

3.2.3 CAN

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

I Dependencies



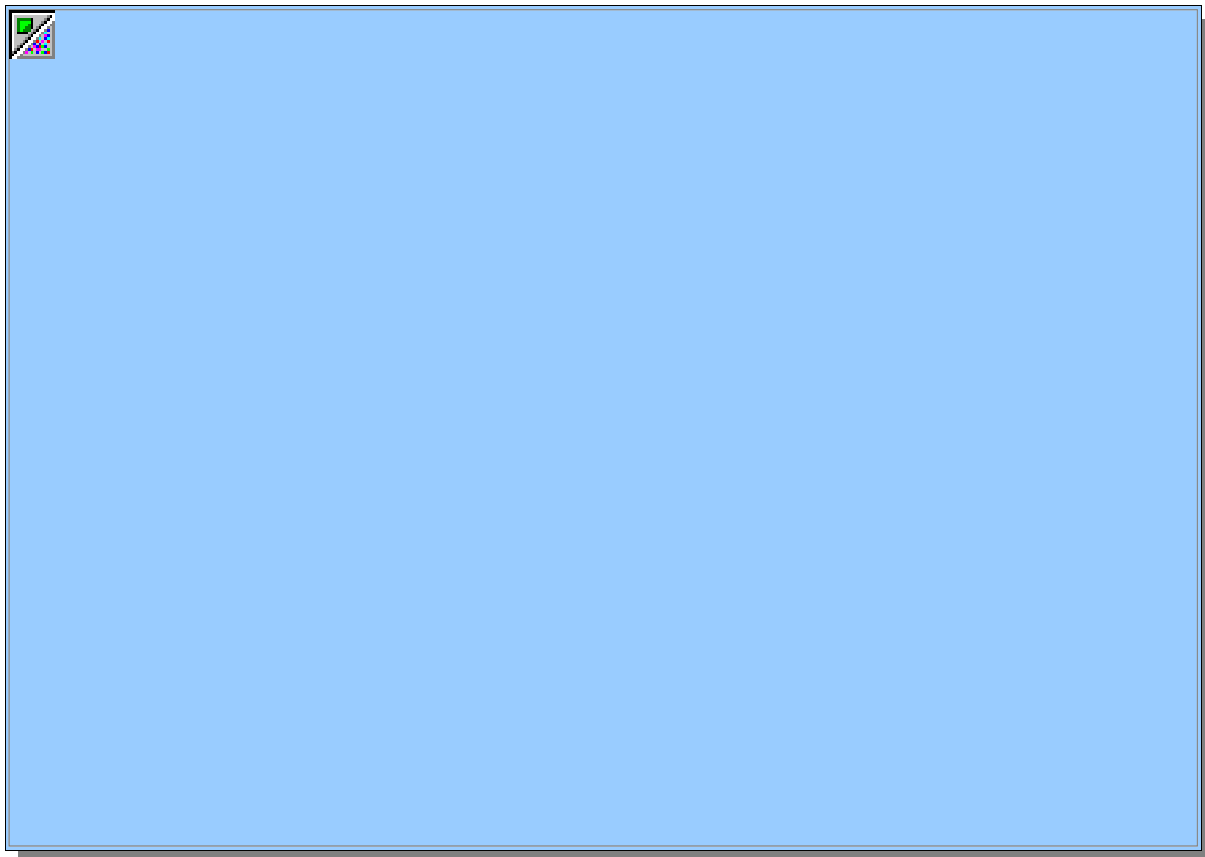
The CANBUS drivers as other components depends on the enabling of *Loadable module support* and you must also disable the power management.

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.

II CAN drivers

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.



a 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.

b 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

c 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.

d 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)
- PClcan-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

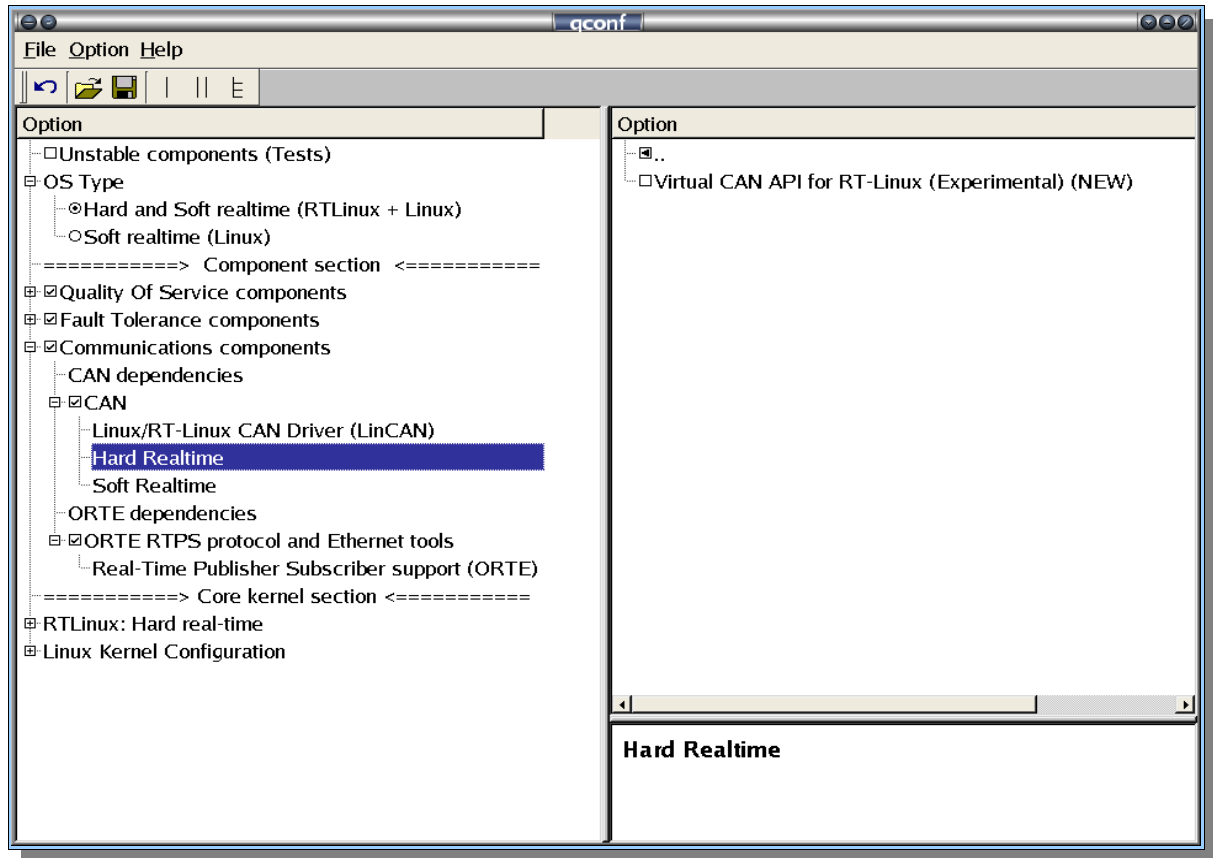
e Virtual CAN board

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.

III Hard real-time



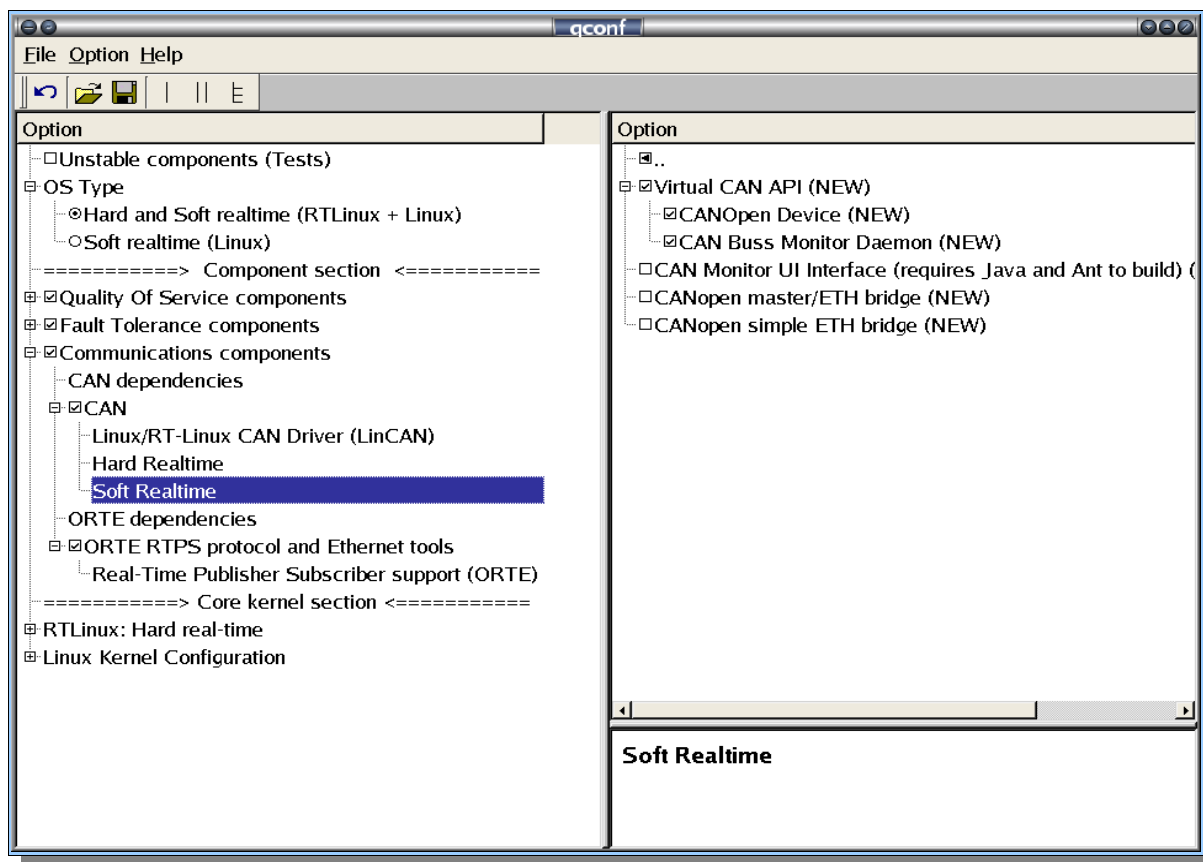
a Virtual CAN API for RT-Linux (Experimental)

RT-Linux version of VCA library.

b CANOpen Device for RT-Linux (Not finished yet)

RT-Linux version of CANOpen slave device.

IV Soft real-time



a 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.

b CANOpen Device

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

c CAN Buss Monitor Daemon

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

d 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.

e CANopen master/ETH bridge

CANopen master/ETH bridge (requires C++)

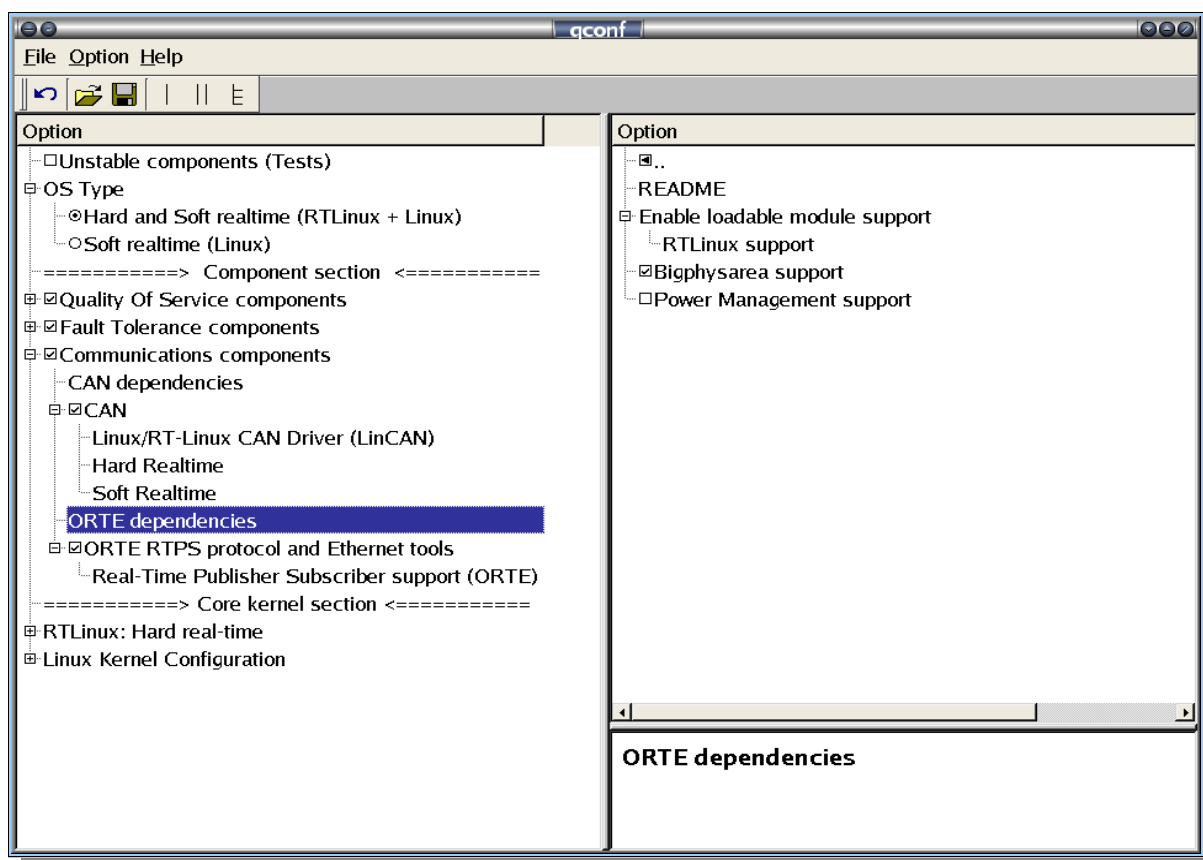
f CANopen simple ETH bridge

CANopen simple ETH bridge

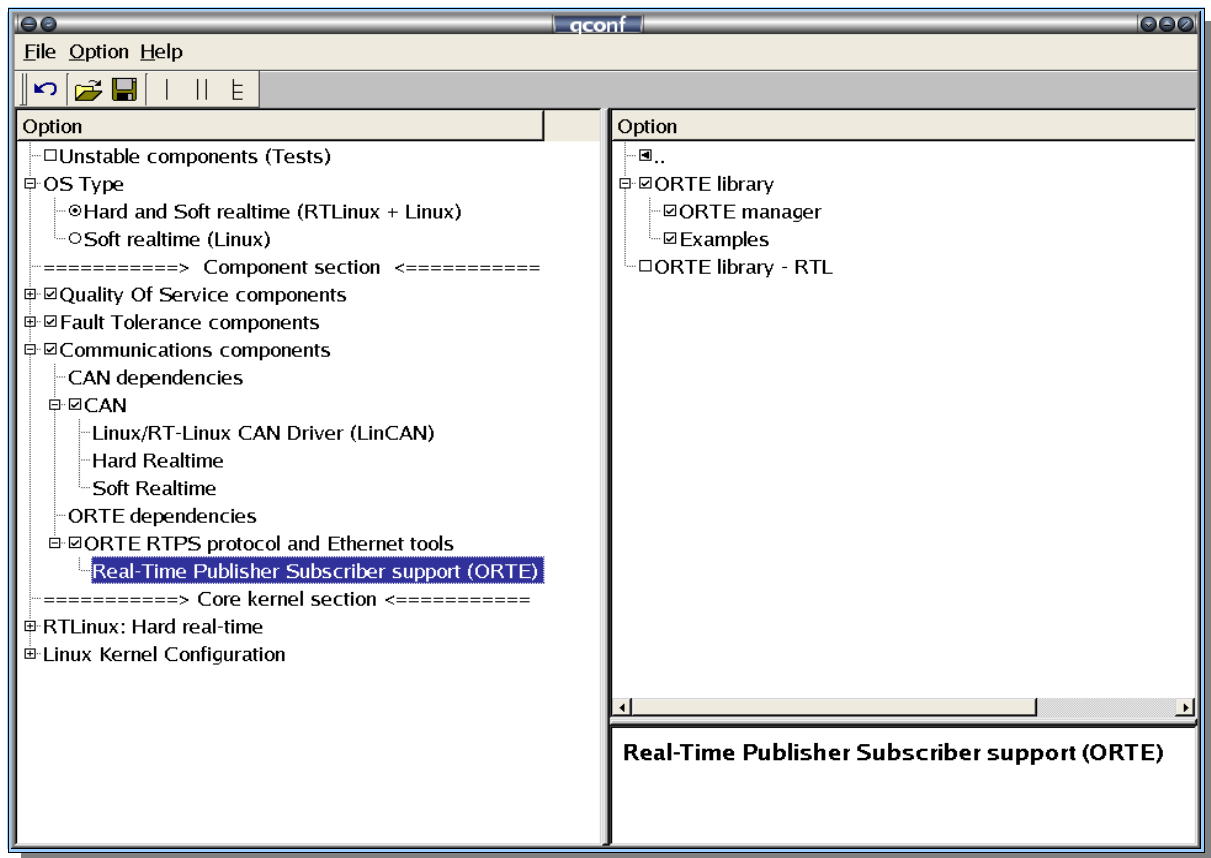
3.2.4 ORTE

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

I Dependencies



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



II Soft real-time

a ORTE library

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

b 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.

c Examples

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

III hard real-time

a ORTE library – RTL

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

b 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.

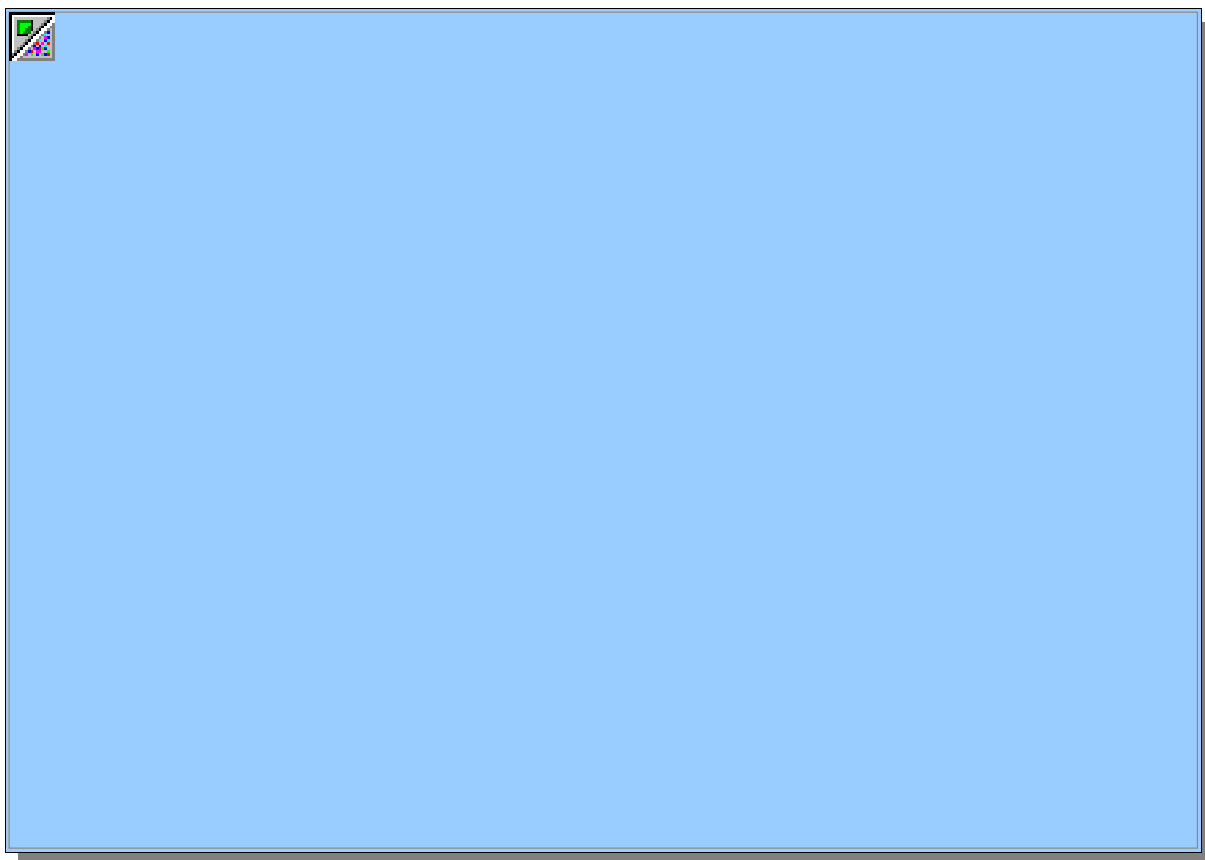
c Examples – RTL

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

3.2.5 RTLinux

I 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.



II Scheduling

a Real time 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.

b Library of Application Schedulers: Disabled

c 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.

d 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.

III POSIX API

a 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

b 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).

c 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.

d 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 many cases a good interprocess communication mechanism.

e 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.

f 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.

g 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 real time applications.

Some characteristics of POSIX Message Queues are:

- Fixed size of messages
- Message Priority of messages
- Asynchronous notification

h Maximum number of open message queue descriptors

i Maximum number of message priorities

j Maximum number of message queues

k Maximum number of messages in the default message queue attributes

l Maximum message size in the default message queue attributes

m 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).

IV Memory Management

a 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-fit 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.

b Big physical memory allocation

This option depends on Linux BIGPHYSAREA option that you can also find in the RTLinux dependencies menu.

Select this option if 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.

c 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.

V Debugging

a Enable debugging

This option compiles RTLinux modules with debugging support.

Say Y if you want to debug RT-programs.

b rtl_printf uses printf

Say Y here if you want rtl_printf output to be buffered and then passed to Linux printf 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).

c RTLinux Tracer support (experimental)

The RTLinux tracer allows tracing various events in the system.

This facility has been obsoleted by the POSIX Trace implementation.

d RTLinux Debugger

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

VI Clock and Timers

a Synchronized clock support

b RT-Linux High Resolution Timers: Disabled

VII Inter Process Communication

a 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 queue (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.

b 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.

c 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.



d Preallocated fifo buffers

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

e 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.

f Number of preallocated fifos

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

VIII 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.

a 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.

b Floating Point Support

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

3.2.6 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.

3.3 Building the kernel and components

3.3.1 make

It is time now to build the kernel and the components.

Now that you have configured 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.

3.3.2 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 examples.
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

3.4 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)

3.5 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.

3.5.1 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.

<i>File</i>	<i>Usage</i>
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.

3.5.2 RTLinux Application

If you build a RTLinux application, you will have to build your application following some rules.

The makefile

Here is an example 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.

3.6 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 example, 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.

3.7 Installing an embedded systems

Trough there is many ways to implement 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 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

3.7.1 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

I From the tarball

This is certainly the best way to have a stable version.

Just download ocera-1.0.0 from the summary page of the OCERA SourceForge site:

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

II 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 /CVS)

```
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 /CVS/ocera

3.7.2 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/rtnix
- 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
 - rtnix 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.

3.7.3 Installing OCERA using emdebsys

If you want to use emdebsys, just do:

```
make xrootfs
```

and follow the instructions.

THIS CHAPTER IS TBD (To Be Done)

3.7.4 Installing OCERA in a busybox environment

I 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
```

II be sure to use the proper development tools

use **dpkg -I** 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

III 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
```

IV 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
```

V make the boot system: example: 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.

3.8 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-1.0.0/misc` directory.

<i>Module name</i>	<i>Component</i>	<i>Description</i>
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	
rtl_sched.o	RTLINUX CORE	
rtl_time.o	RTLINUX CORE	

Chapter : 4 POSIX components

THIS CHAPTER IS TBD (To Be Done)

Chapter : 5 Qua lity of service

THIS CHAPTER IS TBD (To Be Done)

Chapter : 6 ORTE

THIS CHAPTER IS TBD (To Be Done)

Chapter : 7 CAN Devices

THIS CHAPTER IS TBD (To Be Done)

Chapter 8 Fault-Tolerance

8.1 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. All together these components constitute a basic framework to handle degraded mode management that will be progressively enriched.

8.1.1 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.

1 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 runtime 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 modeling of the application relies on the task model described in the next section.

8.1.2 How to use it

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.

In the following image we show the main development process for FT applications.

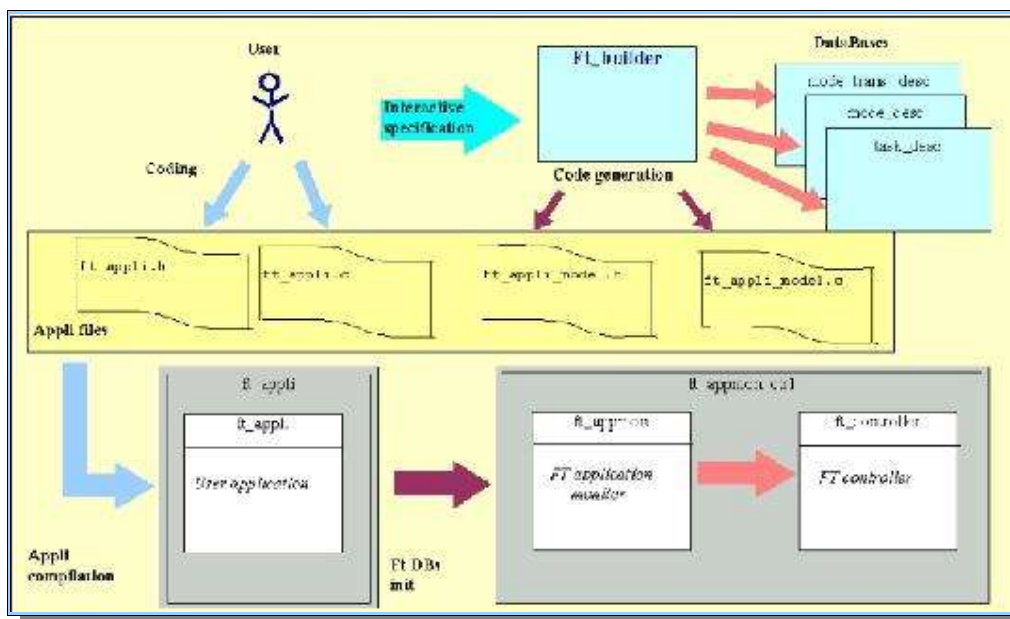


Figure 10-1. FT Coding Process

1 Application design using FT builder

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

a 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 behaviour 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.

b Verification

Verification consists in :

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

c 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 tasks 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 database, the AppliModesTable and the AppliControlDataBase.

d FT_builder 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. In the next sections we review dedicated acquisition windows for task, modes, and transition specification.

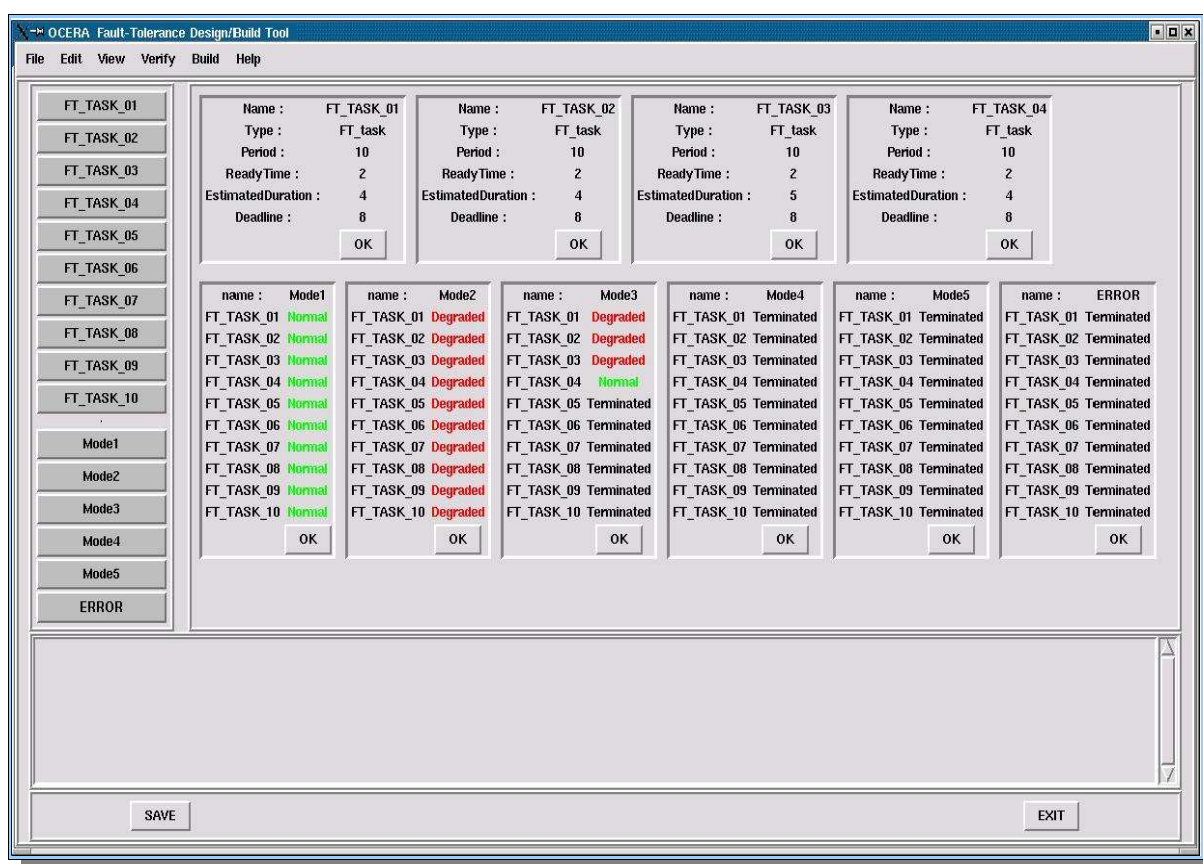
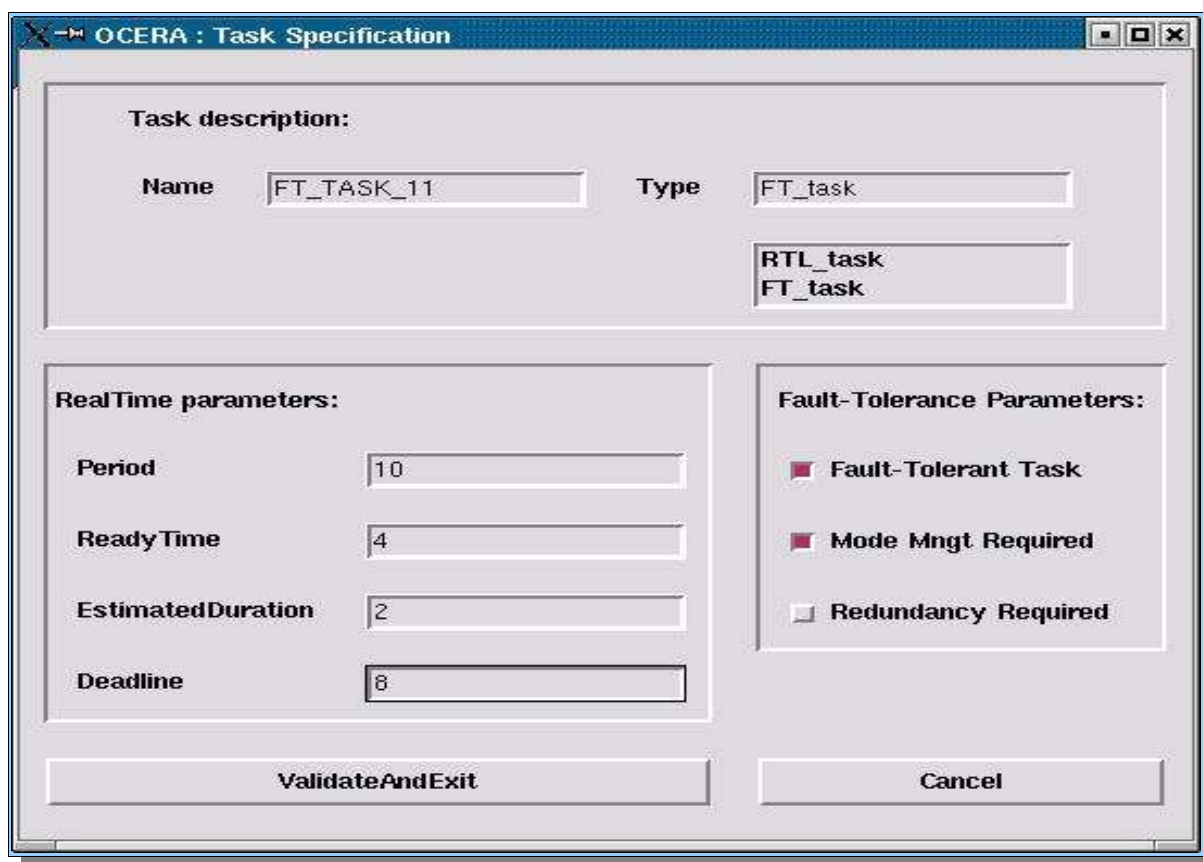


Figure 10-2. FT builder general view

II Task specification

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



The image shows a Windows-style dialog box titled "OCERA : Task Specification". It is divided into several sections:

- Task description:**
 - Name:** A text box containing "FT_TASK_11".
 - Type:** A dropdown menu currently showing "FT_task". Below it, a list shows "RTL_task" and "FT_task" as options.
- RealTime parameters:**
 - Period:** A text box containing "10".
 - ReadyTime:** A text box containing "4".
 - EstimatedDuration:** A text box containing "2".
 - Deadline:** A text box containing "8".
- Fault-Tolerance Parameters:**
 - ☒ **Fault-Tolerant Task**
 - ☒ **Mode Mngt Required**
 - ☐ **Redundancy Required**

At the bottom of the dialog, there are two buttons: "ValidateAndExit" and "Cancel".

Figure 10-3. FT Task specification

III . Mode specification

The mode specification consists in selecting for each task the behavior expected in the mode

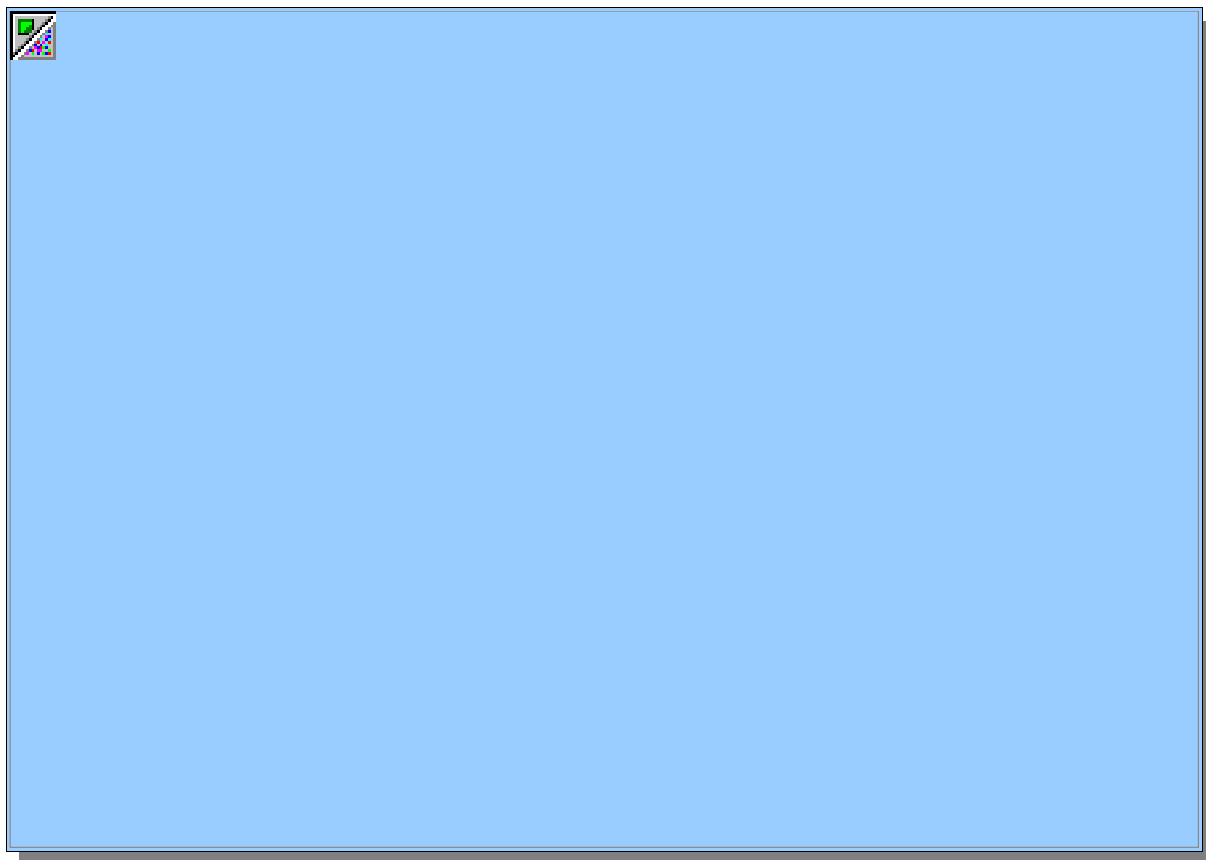
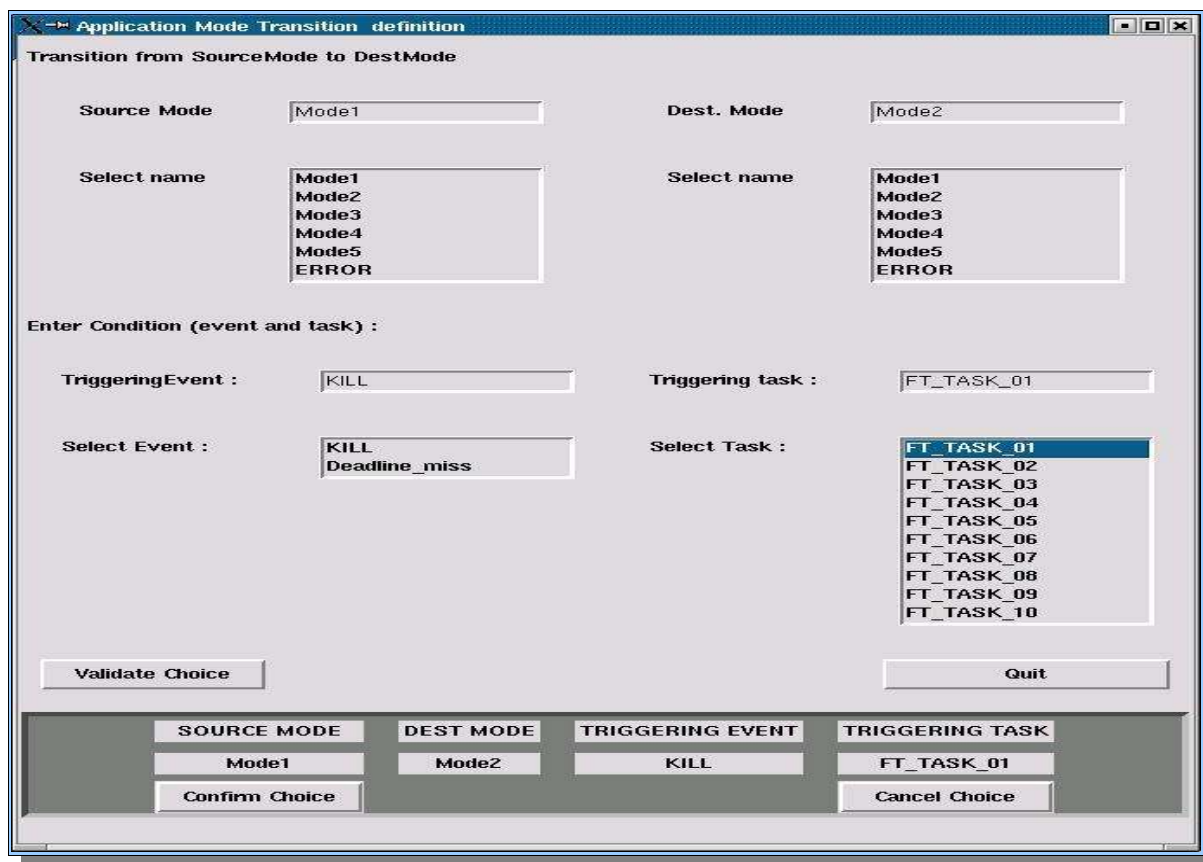


Figure 10-4. FT Mode specification

IV Mode Transition specification

The mode specification consists in selecting for each task the behavior expected in the mode



Application Mode Transition definition

Transition from SourceMode to DestMode

Source Mode: Dest. Mode:

Select name:

Mode1
Mode2
Mode3
Mode4
Mode5
ERROR

 Select name:

Mode1
Mode2
Mode3
Mode4
Mode5
ERROR

Enter Condition (event and task) :

TriggeringEvent : Triggering task :

Select Event :

KILL
Deadline_miss

 Select Task :

FT_TASK_01
FT_TASK_02
FT_TASK_03
FT_TASK_04
FT_TASK_05
FT_TASK_06
FT_TASK_07
FT_TASK_08
FT_TASK_09
FT_TASK_10

SOURCE MODE	DEST MODE	TRIGGERING EVENT	TRIGGERING TASK
Mode1	Mode2	KILL	FT_TASK_01

Figure 10-5. FT Application Mode Transition specification

V 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.

VI User coding

FT facilities for degraded mode management of real_time embedded are available for HardRTLinux environments only. All application tasks are thus RTLinux tasks created within one single application module that can be dynamically loaded into the system.

The prerequisites are thus a running OCERA RTLinux kernel with Posix Trace and FT_components installed (installation aspects are treated in an other section).

A user application must thus consist in a RTLinux module. As usual this module must contain declarations, one init_module function and one cleanup_module function.

a User coding : declarations

The declaration part of the module must include in addition to usual rti headers the following ft specific ones:

- header files from the ft_components : ft_api_common.h and ft_api_appmon_appli.h
- one header file for the application model (generated by the FT_builder) :ft_appli_model.h
- one header file for user defined appli declarations : ft_appli.h

Example 10-1. declarations

```
#include <rti.h>           // include rti linux
#include <pthread.h>       // include posix threadx
#include "ft_api_common.h" // include api commun ft
#include "ft_api_appmon_appli.h" // include api ft_appli_monitor
#include "ft_appli_model.h" // include modele application model (header file)
#include "ft_appli.h"      // include user application code
```

In addition to these includes, the user has to define the routines corresponding to the various behaviors of the ft_tasks. This point is treated later on in this section.

b 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: is primitive initializes data structures related to ft_tasks within internal T_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: is primitive creates and starts ft_task threads. Two arguments are provided : the t_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.

Example 10-2. init_module

```
int init_module(void) { // init module RTLinux
    #include "ft_appli_model.c" // include application model
    for (ap_task_id=1; ap_task_id < APPLI_TASKS_MAX_NB+1; ap_task_id++) {
// tasks building loop
    // FT_task initialization
    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 scheduling 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 identificateur de la tâche
                                ap_task_mode); // ft_task_mode (one of
FT_TASK_NORMAL,FT_TASK_DEGRADED,FT_TASK_NOT_STARTED)
    }
}

```

VII 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.

Example 10-3. cleanup_module

```

void cleanup_module(void) {
    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
    }
}

```

VIII 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 to corresponding to the code to be run for a degraded behavior of the task. In both cases, the code correspond to a usual rtlinux periodic task with a main infinite loop and a body starting by a wait for the next period.

However, there are slight differences since the normal behavior is supposed to be run at application start, while the degraded behavior is supposed first to be suspended and then to dynamically take over normal behavior in case of fault.

In the following example, we have defined a normal behavior routine which is actually run by several ft_tasks(10). These tasks are periodic. On the 100th period iteration, one of them (the 9th) deliberately kills itself in order to show how behavior switch is handled. On the 300th period iteration of each task, we print status information The structure used for the normal routine is strictly the usual structure of rtlinux tasks.

Example 10-4. 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 #
        if ((no_cycle == 100) && (ap_task_id == 9)) { // Particular case : normal thread of ft_task 9 is destroyed at period
100
            rti_printf("\n\nApplication : thread %d cancelling (normal_behavior_%d)", pthread_self(), ap_task_id);
            pthread_cancel(pthread_self());
            break;
        }
        if (no_cycle == 300) { // Print info on thread on reaching 300th period
            rti_printf("\n\nApplication : thread %d no_cycle = %d (normal_behavior_%d)", pthread_self(), no_cycle,
ap_task_id);
        }
        nloops=5000; // just consuming time
        for (j=0; j<nloops; j++);
    }
    return (void *) 0;
}

```

Coding degraded behavior routines is slightly different. Indeed, usually, the thread with the degraded mode is suspended until an error occurs on the thread with normal behavior. So two synchronization steps are required, one in order to wait for the occurrence of error which will wakeup the thread with degraded behavior; and one for waiting for the next ready time for the thread. So, two wait instructions (pthread_wait_np) are required, the first one corresponds to the wakeup condition (a fault occurred on normal thread), it is set at the very beginning of the routine ; the second one corresponds to the next period resume for the degraded thread, the thread has been made periodic and has to wait for the next period, so a second pthread_wait_np is required in the beginning of the main loop.

This is illustrated by the following example, which shows a degraded routine that just prints its status on occurrence of 300th period.

Example 10-5. degraded behavior routine

```

void *ap_degraded_behavior_routine(void *arg) { // degraded behavior routine
    pthread_wait_np(); // First wait corresponding to suspension of thread with degraded behavior
    ap_task_id = (int) arg;
    rti_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 #
        if (no_cycle == 300) { // Print info on thread on reaching 300th period
            rti_printf("\n\nApplication : thread %d no_cycle = %d (degraded_behavior_%d)", pthread_self(), no_cycle,
ap_task_id);
        }
        nloops=5000; // just consuming time
        for (j=0; j<nloops; j++);
    }
    return (void *) 0;
}

```

IX Application compiling

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

- posixtrace

- ft_components : ftappmon and ft_controller

This provides a general OCERA Makefile that can be used to compile the application.

The application code given as an example is located in the following directory : ft/ftcontroller/examples/ftappli The compilation of the ftappli module, is achieved by applying the following commands at the OCERA (or ftappli) directory level :

- Clean the \$OCERA_DIR/components/ft/ftcontroller/examples/ftappli directory:

```
$ make clean
```

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

- Compile the ftappli module:

```
$ make all
```

The ftappli.o module is now available under the \$OCERA_DIR/components/ft/ftcontroller/examples/ftappli directory.

X Application running

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 ft/ftcontroller/examples directory level :

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

- Be a root user

```
$ su  
Password:  
#
```

Actually, it is necessary to be a root user. Further, the user has to be a normal user.

- Install and execute all the modules:

```
# make test
```

- Get the modules execution traces:

```
# exit  
$ dmesg > dmesg
```

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

The Makefile code is the following

```
test:  
    @echo "WP6 - Fault Tolerance Components !"  
    @echo "Version 02 - July 2003"  
    @rmmod ftappli  
    @rmmod ftappmonctrl  
    @rtlinux stop  
    @echo ""  
    @echo "Now we start RTLinux"  
    @echo "Type <return> to continue"  
    @rtlinux start
```

```
@read junk
@echo "Now the ftappmonctrl module is started"
@insmod $(O_FT_DIR)/ftcontroller/ftappmonctrl.o
@echo "Now the ftappli module (application) is started"
@insmod $(O_FT_DIR)/ftcontroller/examples/ftappli/ftappli.o
@echo "Now wait 30 seconds ..."
@sleep 30
@echo ""
@echo "Now the FT application and components are stopped"
@sync
@rmmod ftappli
@rmmod ftappmonctrl
@rmmod rtl_ktrace
@rtlinux stop
@echo ""
@echo "Done!"
@echo ""
```

XI Example of application trace

The following trace corresponds to an application with 10 ft_tasks.

Two application modes are defined :

- Mode M1 : in which all ft_tasks have a normal behavior
- Mode M2 : in which ft_tasks 1,3,6,8 have a normal behavior; in which ft_tasks 2,4,5,7,9,10 have a degraded behavior;

A mode transition is defined from M1 to M2 on occurrence of pthread_kill on ft_task 9

Example 10-6. execution trace of ft appli example

```
mbuff: kernel shared memory driver v0.7.2 for Linux 2.4.18-rtl3.2-pre1_ocera_01
mbuff: (C) Tomasz Motylewski et al., GPL
mbuff: registered as MISC device minor 254
RTLinux Extensions Loaded (http://www.fsmlabs.com/)
*****
FT_Controller
*****
FT_Appli_Monitor
*****
FT_Appli
*****
...
Application : thread -143261696 cancelling (normal_behavior_9)// Simulation : kill of task 9 normal thread
FT_Controller : System call : task 19 (-143261696) syscall PTHREAD_KILL (2) // pthread_kill Detection
FT_Controller : Kill the normal thread !!! // Local controller action : cancellation of task 9 normal thread
ft task id --- 9
ft normal thread kid --- -143261696
FT_Controller : Wake up the degraded thread !!! // Local controller action : wakeup of task 9 degraded thread
ft task id --- 9
ft degraded thread kid --- -283738112
FT_Appli_Monitor : Function ft_notify_failed_thread // Event Notification to Application Monitor
FT_Appli_Monitor : before switch ft_current_appli_mode= {1 , M1} // Current Mode : M1
FT_Appli_Monitor : ft_new_appli_mode= {2 , M2} // New target Mode : M2
FT_Controller : Kill the normal thread !!! // Controller action : normal thread cancellation for task2
ft task id --- 2
ft normal thread kid --- -766181376
FT_Controller : Wake up the degraded thread !!! // Controller action : wake up of degraded thread for task 2
ft task id --- 2
```

```

ft degraded thread kid    --- -146964480
FT_Controller : Kill the normal thread !!!
ft task id                --- 4
ft normal thread kid      ---
...
Application : thread -812810240 no_cycle = 300 (normal_behavior_1) // Task 1 normal thread is still active
Application : thread -146866176 no_cycle = 300 (normal_behavior_3)
Application : thread -1039106048 no_cycle = 300 (normal_behavior_6)
Application : thread -767754240 no_cycle = 300 (normal_behavior_8)
...
Application : thread -144080896 no_cycle = 300 (degraded_behavior_10) // Task 10 degraded thread is active
Application : thread -283738112 no_cycle = 300 (degraded_behavior_9)
Application : thread -802455552 no_cycle = 300 (degraded_behavior_7)
Application : thread -283934720 no_cycle = 300 (degraded_behavior_5)
Application : thread -147718144 no_cycle = 300 (degraded_behavior_4)
Application : thread -146964480 no_cycle = 300 (degraded_behavior_2)
Application : CLEANUP application threads !!! // Threads cancellation for all appli tasks
FT_Appli_Monitor : CLEANUP ft_appli_monitor thread !!! // Cancellation of ft_monitor thread
FT_Controller : CLEANUP ft_controller thread !!! // Cancellation of ft_controller thread
unloading mbuff
mbuff device deregistered

```

8.1.3 Degraded Mode management: architecture overview

The current implementation architecture is based on OCERA hard real-time platform which consist mainly in 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 In the following image we show the main development process for FT applications.

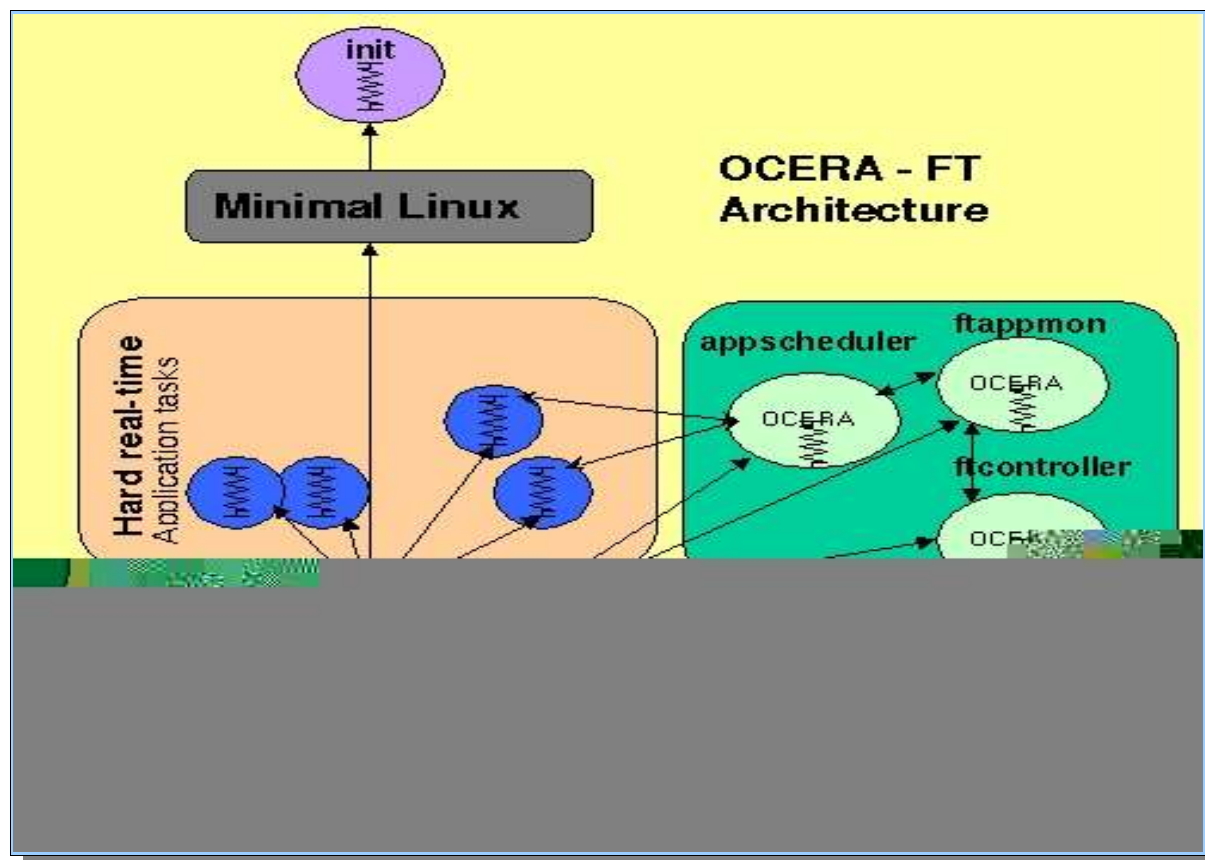


Figure 10-6. FT architecture overview

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.

1 FT API

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.

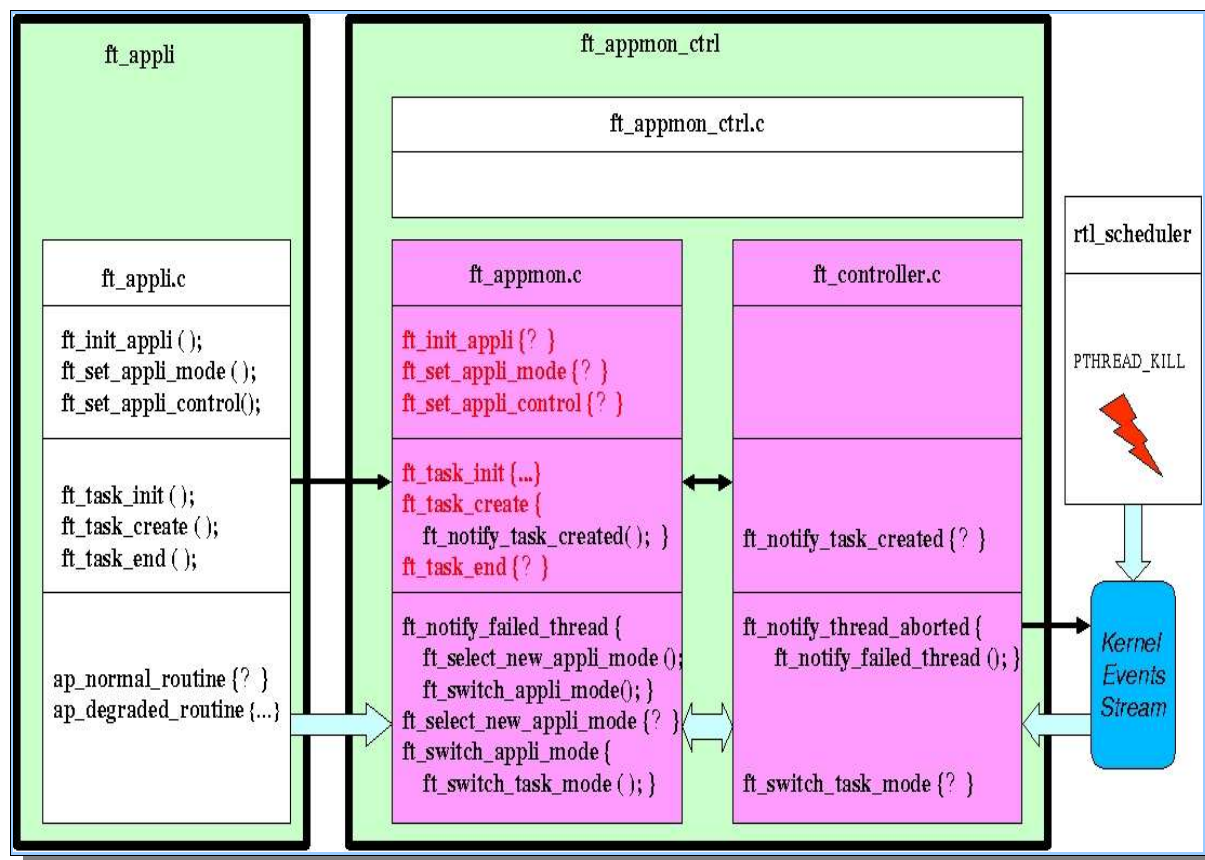


Figure 10-7. FT API overview

Chapter 9 Ada

9.1 Overview

The Real-Time Linux OS (RTLinux) is an attractive platform for real-time programming, since real-time tasks can be guaranteed bounded response times, while a number of applications, IDEs, GUIs, etc. are also available for the same platform.

In RTLinux, real-time tasks are implemented as kernel modules, implemented in "C". Special care must be taken when writing these modules: a bug in a single task can make the whole system to hang or crash, since these modules are executed in the kernel memory space.

This is clearly an area where Ada can be of great help: Ada is strong typing, consistency checking, robust syntax and readability, and the availability of high quality compilers, encourage the writing of correct software and allow to catch bugs early in the implementation.

RTLGNat is a modification of GNAT Ada compiler to allow to write RTLinux modules in Ada.

9.2 ADA RT implementation in OCERA: RTLGNat

9.2.1 Presentation of RTLGNat

RTLGNat is a port of the GNAT Ada compiler for the Real-Time Linux operating system. An Ada program can be compiled as a Linux loadable kernel module, where the program's tasks run as RTLinux threads.

As a result, the Ada program is run in the Kernel space with more priority than any other running Linux application, therefore achieving Real-Time performance UNDER CERTAIN CONDITIONS that will be explained in more detail in section "Real-Time performance".

9.2.2 Features

The current version of RTLGNat implements the core language and the Real-Time Systems Annex (annex D).

9.2.3 Requirements

A PC machine with:

- Linux Kernel 2.4.18 or later for the i386 or above. Earlier versions of the kernel will probably do as well, but we have not tested this point. The Linux Kernel is available from <http://www.kernel.org>
- GNAT 3.14p or 3.15p Ada compiler for Linux. Most Linux distributions include an "unofficial" version of the GNAT compiler as an optional package, but only the official distribution of the compiler should be used with RTLGNat. The official site for the GNAT compiler is <ftp://cs.nyu.edu/pub/gnat/>
- The "bigphysarea" patch for the kernel. Available from <http://www.polyware.nl/~middelink/En/hob-v4l.html>
- Real-Time Linux 3.1, 3.2pre1, 3.2pre2 or later (as long as the POSIX interface is preserved). RTLinux is available from <ftp://www.rtlinux-gpl.org/pub/rtlinux>

9.2.4 Real-Time performance

The goal of RTLGNat is to be able to run concurrent real-time applications with real-time guarantees. The RTLGNAT approach is to run the run-time system and the hard real time tasks as a kernel module, therefore avoiding the interference from other processes.

The way the operating system (RTLinux) provides this guarantee is basically by virtualisation the interrupts for Linux. When an interrupt occurs, it is RTLinux who captures it first. If necessary, RTLinux reissues the interrupt to Linux after it has been captured (and possibly processed) by itself. This scheme allows to run the real-time tasks in an Ada program with a higher priority than Linux.

Nevertheless, there are still some factors that may affect this ideal scenario. One of them is caching. This is not a problem of RTLGNat, but you must take the effects of cache faults into account in your application analysis in order to achieve a deterministic behavior.

Another source of indeterminism, and this is more problematic, is the fact that some applications and drivers may violate the RTLinux scheme for enabling and disabling interrupts. These applications enable and disable interrupts by means of the assembly cli and sti instructions, instead of using the RTLinux macros STI and CLI. Therefore, they actually (not virtually) enable and disable interrupts. This may occur at any time, which makes the system unpredictable. Known examples of such applications are the XWindows system and some commercial driver modules.

Unfortunately, we do not have a list of such applications, but you can guess whether a program executes the actual "cli" instruction with this simple command: `objdump -S module-or-program-name | grep cli` (thanks to Juan Zamorano for the fun about this command ;-)

9.2.5 Known problems and limitations

The limitations imposed by version 0.1 of RTLGNat have been removed. See section "Version history" for more details.

9.2.6 Compilation notes

Please, note that makefiles must be called with -j1 flag during the installation of RTL-Gnat 1.0, to avoid parallel execution of several compilations in multiprocessor platforms. This is the default behavior.

9.2.7 Version history

- 1.0 Second release. September 2003.
- More exhaustive coverage of the run-time system.

- Works with both GNAT 3.14p and 3.15p.
- The RTGL layer has been re-implemented. Now the code is better organized and there is added support for hardware exception handling.
- Full Ada.Text_IO support (for the virtual files supported by RTLinux.)
- Problems with floating point conversions in RTLGNat 0.1 have been solved.
- 0.1 First release. April 2003.

9.2.8 Main sites

You can download RTLGNAT 0.1a from the following URL:

- <http://rtportal.upv.es/>
- <http://www.ocera.org/>

9.2.9 ADA Library

9.2.10 Compiling

To compile you need the RTLGNat .

9.2.11 Sample program

The following example create two real time tasks:

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

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

task body Std_Task is
Next_Time : Time;
Period : Time_Span := Microseconds (100);
Next_While : Time;
Period_While : Time_Span := Microseconds (10);
begin
accept Call;
Next_Time := Clock + Period;
loop
Put ("I am "); Put (Id); New_Line;
Next_While := Clock + Period_While;
while Next_While > Clock loop
null;
end loop;
delay until Next_Time;
Next_Time := Clock + Period;
end loop;
end Std_Task;
Std_Task1 : Std_Task(1);
Std_Task2 : Std_Task(2);
dev : Integer;
```

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

9.3 . *What's next*

In this chapter you have seen how to configure an OCERA system and the tools you need to build every things. Now you will learn more about what you can do if you get into trouble with the usage of debuggers and tracing tools.

Chapter 10 Chapter 11. Debugging and tracing

10.1 Debugger

You can use GDB to debug your application. Insight, the GDB GUI to debug your application.

THIS CHAPTER IS TBD (To Be Done)

10.2 Analyzers

You can use GDB agents to trace your application. You also can use kiwi to view the real-time tasks.

THIS CHAPTER IS TBD (To Be Done)

10.2.1 Supervision tools

1 The /proc Filesystem entries

The different components bring information to the user's space, and so to you through the /proc interface. You can use this interface to get informations on:

See the OCERA Programmer's Guide to see how you can use the /proc interface for your own purpose when writing a real time application.

10.2.2 Setting POSIX traces

THIS CHAPTER IS TBD (To Be Done)

10.3 *What's next*

It is near the end of this book, you have seen every thing you need to know about developing an application with the OCERA framework. The next chapter will show you a few sample applications you can poke to build you own application.

Chapter : 11 RtxFS

THIS CHAPTER IS TBD (To Be Done)

Chapter : 12 SA-RTLinux

THIS CHAPTER IS TBD (To Be Done)

Chapter : 13 Cross compilation

THIS CHAPTER IS TBD (To Be Done)

Chapter : 14 Qualifying an OCERA system

14.1 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: Software error can cause discomfort.
- Level E: Software error has no effect.

14.2 How to keep the criticality level when developing an application

14.2.1 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.

14.2.2 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.

14.2.3 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 Assessment 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.

Chapter : 15 Performance

15.1 *Hard Real time performances*

Using OCERA, you can expect the following performances:

Table 5-1. Hard Real time performances

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

15.2 *Soft Real time performances*

Using OCERA, you can expect the following performances:

Table 5-2. Soft Real time performances

type of measure done	result	commentary
----------------------	--------	------------

Interrupt latency		
-------------------	--	--

Task switch latency		
---------------------	--	--

Semaphore latency		
-------------------	--	--

Number of tasks		
-----------------	--	--

CBS reservation		
-----------------	--	--

15.3 General performance

Table 5-3. General performances

type of measure done	result	commentary
----------------------	--------	------------

Filesystem size		
-----------------	--	--

File size		
-----------	--	--

Real time Ethernet bandwidth		
---------------------------------	--	--

CAN BUS bandwidth		
-------------------	--	--

15.4 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.

15.5 What's next

In this chapter, you learned what are the performances you can expect from a system running Linux/RTLinux and the OCERA components. In the next chapter you will start to see how you can make it run for real.

Chapter : 16 Appendix A. control command with CAN

THIS CHAPTER IS TBD (To Be Done)

Chapter : 17 Appendix B Real Time Ethernet

THIS CHAPTER IS TBD (To Be Done)

Chapter : 18 Appendix C. Robotic application

THIS CHAPTER IS TBD (To Be Done)

Chapter : 19 Appendix D. Streaming video

THIS CHAPTER IS TBD (To Be Done)

Chapter : 20 Glossary

THIS CHAPTER IS TBD (To Be Done)

Chapter : 21 Index

THIS CHAPTER IS TBD (To Be Done)

Chapter : 22 Bibliography

[Aldea02] Mario Aldea-Rivas and Michael González-HArbour, 2002, 14 th Euromicro Conference on Real-Time Systems (ECRTS'02), POSIX-Compatible Application-Defined Scheduling in MaRTE OS.

[Abeni98] Luca Abeni and Giorgio Buttazzo, IEEE Real-Time Systems Symposium, Madrid, Spain, 1998, Integrating Multimedia Applications in Hard Real-Time Systems.

[Sha90] L. Sha, R. Rajkumar, and J.P. Lehoczky, 1990, IEEE Trans. on Computers, 39, 1175-1185, Priority Inheritance Protocols: An Approach to Real-Time Synchronisation.

[Baker91] T.P. Baker, 1991, The Journal of Real-Time Systems, 3, 67-100, Stack-Based Scheduling of Realtime Processes.

[Liu73] C.L. Liu and J.W Layland, 1973, Journal of the ACM, Scheduling algorithms for multiprogramming in a hard real-time environment.

[PTS] Posix Test Suite.

[BPA] Bigphysarea.

THIS CHAPTER IS TBD (To Be Done)