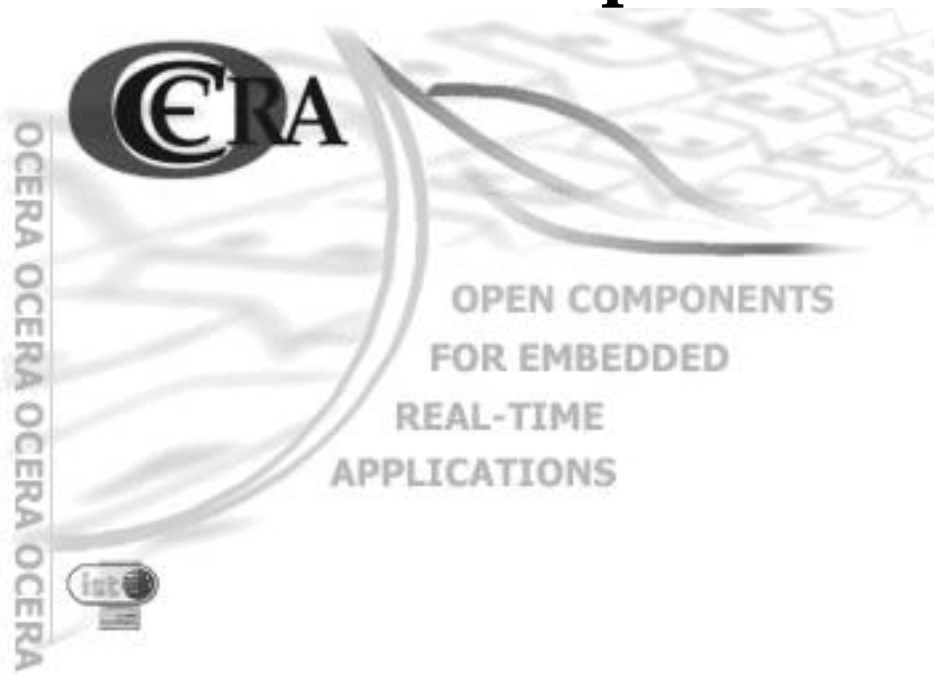
WP2 - Architecture Specification



OCERA Architecture and Component Definiton

WP2 - Architecture Specification : OCERA Architecture and Component Definiton

by Ismael Ripoll, Alfons Crespo, Adrian Matellanes, Zdenek Hanzalek, Agnes Lanusse, and Giuseppe Lipari

Table of Contents

List of Tables

List of Figures

Document presentation

Table 1. Project Co-ordinator

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

Table 2. Participant List

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

Chapter 1. Introduction

The goal of the OCERA project is to provide support to real-time applications in embedded systems based on the Linux kernel. This support will be provided by a set of components that will be:

- flexible: new schedulers will support a wide variety of applications from hard to soft real-time;
- configurable and scalable from a small to a fully featured system;
- robust, providing fault-tolerant and high performance;
- portable to several hw/sw configurations.

The results of the project will be:

- a library of open source software components for the Linux kernel, that support real-time embedded systems at the design and at the implementation phase.
- a contribution to the improvement of the Linux kernel by participating in the evolution of an open community of developers.

The OCERA consortium will join the real-time Linux community by following the standard development and design methodology currently used in open source projects. In this way the OCERA consortium and the Linux community will both benefit of mutual collaboration; also, the results of the OCERA project will be maintained and developed after the end of the project.

In this project, we will to provide support for:

- critical real-time applications that need a very low response time and can be considered as "trusted" (that is, we can guarantee their correctness);
- less critical and more complex real-time applications, that may not be trusted (that is we are not sure about their correctness).

Since critical applications need low response time and are "trusted", we will use the same approach followed by the RTLinux executive (see Section 2): the critical applications run in kernel space and can directly access the hardware. However, we want also to treat less critical applications by providing real-time support in user space. This is particular useful when deadline with untrusted applications.

This documents is organised as follows: next section presents a general overview of the main Linux developments related with real-time, and the different approaches used; following section deals with the OCERA components definition; followed by the architecture section, with a general overview of the different scenarios where OCERA components will be used; finally a section with a concrete description of each component and the role and place it will have in the general architecture; last section provides a glossary of the terms and concepts used in all the OCERA documentation.

Chapter 2. Real-Time in Linux

Linux is a full-featured UNIX implementation, conforming to the POSIX standard. The Linux kernel was not originally designed for real-time applications. Althought kernel developers are actively working in improving the responsiveness of the kernel, Linux is still not suited to support hard real-time applications with tight timing requirements for at least two reasons:

- Kernel responsiveness. The default mechanims used by Linux for protecting its internal kernel structures can cause long non-preemptable sections.
- The lack of real-time mechanisms, like priority inheritance, sufficient timing resolution, etc.

Several mechanisms have been proposed for supporting real-time in Linux. They can be divided in two distinct classes [Dankwardt00]:

- Mechanisms that use Linux for accessing the hardware. Following this approach, the latency of the Linux kernel must be reduced as much as possible. It is also necessary to modify the kernel by introducing real-time mechanisms like priority inheritance, dynamic scheduler, high resolution timers etc.
- Mechanisms that by-pass Linux for accessing the hardware. In this case, the hardware interrupts must be virtualized. The interrupts that are used by critical hard real time tasks are directly handled by-passing the standard Linux kernel. The other interrupts are forwarded to Linux when no real-time task is active. Also, cli and sti instructions must be modified to avoid that Linux disables interrupts for long intervals of time.

There are several on-going projects in both approaches. The so-called "low latency patch" and "preemption patch" belong to the first one. Also, several research groups proposed modifications to Linux for introducing real-time mechanisms. Examples of this approach are Linux/RK by TimeSys, RED-Linux, etc.

The second solution can be implemented in different ways. RTLinux [RTLinux.org] and RTAI [RTAI] implement a real-time executive (which consists of a interrupt handler plus a scheduler and some real-time mechanism) and real-time tasks as kernel modules that are dynamically linked with the Linux kernel and run in kernel space.

A different approach is used by other systems, like L^4-Linux [L4], in which a small microkernel is used for running both a modified version of the Linux kernel and real-time tasks in user space.

RTLinux and RTAI use similar mechanisms to acquire direct control of the hardware, by intercepting the interrupts and modifying a small portion of the Linux kernel code. On the other hand, L^4-Linux differs from previous approaches in that Linux runs completely in user space as a task on the L^4 microkernel. Therefore, the L^4 microkernel has full control of the "guest" operating system. As a consequence, by introducing a small overhead, it is possible to implement memory protection and other security mechanisms.

The OCERA architecture is based on the RTLinux architecture. Figure 2-1 presents a schematic overview of the relations between Linux kernel and RTLinux. RTLinux is located just above the hardware to represent the tight control that RTLinux has on it. Also it is important to note that while RTLinux has direct control of the interrupts (dark angled arrows), Linux works with virtual interrupts delivered by RTLinux. Hardware devices can trigger an interrupt which will be received by RTLinux, but will not be delivered to Linux until all real-time tasks are idle.

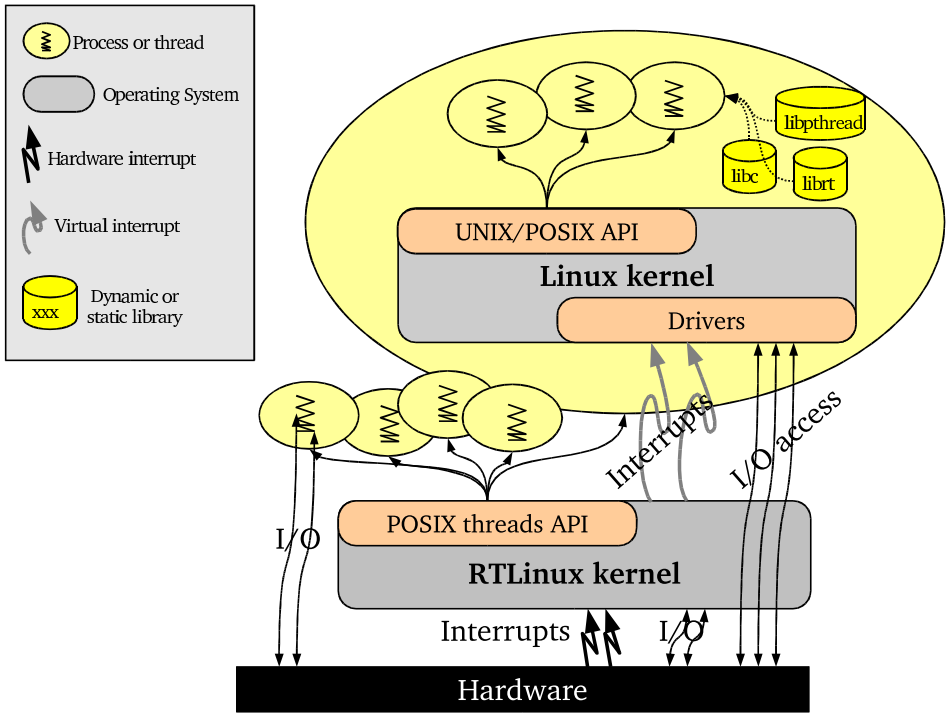


Figure 2-1. General Linux and RTLinux overview

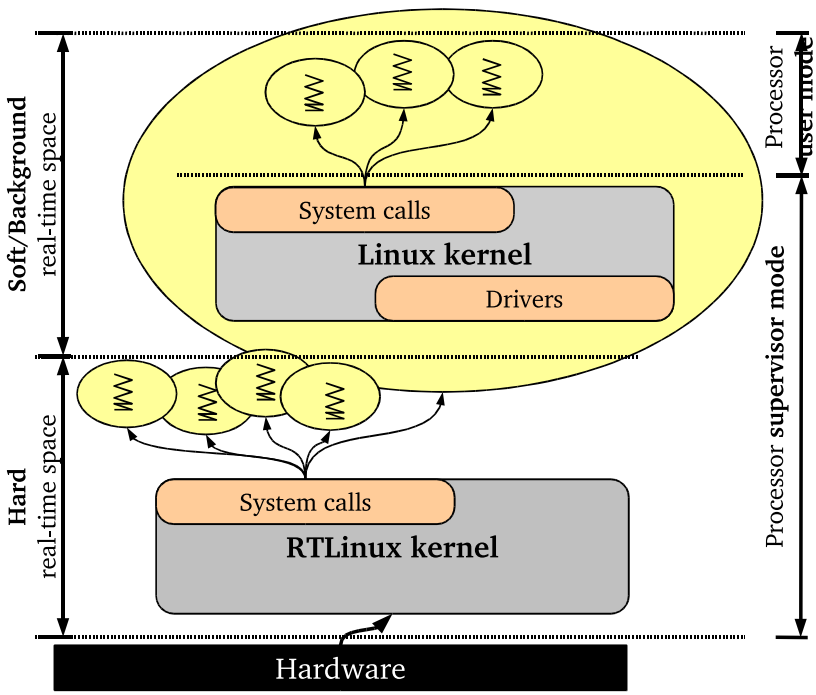


Figure 2-2. Different view of the execution environments

Although RTLinux and Linux run both in kernel space, when RTLinux is installed, the code of Linux is modified (applying a patch to the sources files and recompiled) to prevent Linux from disabling interrupts. As a result, Linux has no direct control of the interrupt and timer subsystems. For this reason, RTLinux is used for hard real-time execution, and Linux jointly with its user applications is used for background (or non-hard) real-time activities. In OCERA, we plan to provide components both at the RTLinux level and at the Linux level.

As a summary of the pros and cons of the RTLinux approach to real-time: RTLinux applications run faster; can work with the hardware like most programmers are costumed to do (MSDOS® style); but buggy task can compromise the system integrity due to the lack of protection.

2.1. Concurrent execution paradigm

An operating system can provide concurrent execution in two different forms: processes and threads. In general, each process has a protected virtual address space, a set of open files, one single execution flow (program counter), etc. Processes are self-contained units. A thread, on the other hand, consists of an execution context and a state (i.e., a program counter, a stack, and local variables). Usually, a thread is contained within a process; thus, a thread has global access to all states within its containing process. A process can contain more than one thread. Threads contained by different processes may only communicate through inter-process communication mechanisms.

The main differences between processes and treads as real-time is concerned are summarised in the following table:

Table 2-1. Process versus thread

Process	Thread
* Memory protection among processes. * Easy to distribute in a distributed system. * Well defined and complete programming API.	* Easy and small support implementation. * Fast context switch. * Intrinsic shared memory. * Efficient communication. * Finer grain parallelism.

It is common to use both methodologies in a real-time embedded application. Threads are better suited to program the low level, hard real-time activities, while processes can be used for human interaction and tasks that do not require short timing response.

Most of the POSIX real-time extensions are based on the thread model, an example of this is that solution to priority inversion is only available in thread API (`PTHREAD_PRIO_PROTECT` and `PTHREAD_PRIO_INHERIT` protocols). POSIX profiles define subsets of the operating system API which fulfil the requirements of specific targets. There are four profiles: "Minimal System", "Controller", "Dedicated System" and "Multipurpose System". The two first profiles do not have process support, only thread support is required. RTLinux follows the "Minimal System" standard.

There are three different libraries that provide thread support on Linux. The first implementation was done By Xavier Leroy and is known as the LinuxThreads. It is integrated and distributed jointly with the standard "C" library. Currently there are two new competing implementations [Cooperstein02] of the POSIX thread standard called: “New Generation POSIX Threads” (NGPT) and “Native POSIX Thread Library” (NPTH). These implementations improve both, compatibility with POSIX standard and performance.

OCERA components will assume the thread model if not explicitly stated otherwise. The term “task” will be used to refer to both process or thread.

Chapter 3. OCERA components

Most computer applications can be designed following a clear and standard software engineering methodology like object oriented methodology or client/server model. The code that is part of the operating system can not be easily categorised or described in a unique way, specially when the type of operating system is an RTOS. In this case, the intrinsic characteristic of an RTOS: efficiency and predictability of the code are mostly opposed to clearly divided and well organised code. It do not mean that the internal code of the RTOS is bad written or chaotic, what it means is that the RTOS can not be structured in a well defined and clearly separated blocks of code with clear inputs and outputs. For example, when RTLinux is loaded (inserted as a kernel module) it takes the control of the interrupt system by modifying the Linux code while Linux is running, which is one of the less advisable programming methods but provides the fastest result.

One of the definitions of what is an operating system is that it is the piece of software that hides all the complexity of the underlaying hardware to provide a clear and orthogonal programming environment to user applications.

An OCERA component is defined as follows:

> "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 and Communications"

This is a fairly general definition that will be extended and detailed next.

As a piece of software we mean:

- a modification of the Linux kernel or RTLinux executive, which will be released as a patch file against a specific kernel version or integrated into the final OCERA kernel;
- a module which can be loaded (with the `insmod/modprobe` commands) and provides new functionalities and may use some of the already installed services;
- a library, dynamic or static, which can be linked with the user application;
- or a standalone thread or process (for example a debugging program).

3.1. Component classification

OCERA components can be classified according to two criteria:

- Protection
- Level

According to the "protection" criteria, we can identify components that run in user space and components that run in kenel space. User space components are Linux applications running in their own address spaces at the lowest privilege level. Because of the protection mechanism enforced by the Linux kernel, these components are not able to crash the system, even if they misbehave and try to access random memory locations.

Components running in kernel space are: the (patched) Linux kernel (including its own device drivers), the RTLinux executive, and the hard real-time tasks (also referred as RTLinux tasks or RTLinux applications).

According to the "level" criteria, kernel space components can be located in the RTLinux layer (as modifications or additions to the RTLinux executive) or in the Linux layer (as modifications to the Linux kernel, which are independent from RTLinux).

Note that some functionalities can be provided both in user space and in kernel space (for example, the QoS manager developed by SSSA will be available both as a kernel

module or as an user space daemon), and other functionalities can be implemented at different levels in kernel space (for example, the CBS scheduler will be implemented both in the RTLinux layer - by UPVLC - and in the Linux layer - by SSSA). For this reason, each component documentation will contain precise information about its protection level (user space / kernel space) ant its location (RTLinux layer / Linux layer).

It is worth to note that the RTLinux layer can be logically divided in three sub-layers:

1. **Low-level RTLinux**: This kind of components are highly related with the current RTLinux capabilities or internal algorithms, thus *it requires to modify the current RTLinux source* code in order to provide the new functionality or to improve an available one.

   This kind of component will be distributed in a patch-form and hopefully incorporated in the main stream of the implied kernel source.

2. **High-level RTLinux**: *It only needs the current API of the RTLinux kernel* or an extended API offered by other kernel component in order to implement its new functionality. It does not require to modify the existing kernel source code or any low-level kernel component.
3. **RTLinux Applications**: This kind of component uses the kernel API to provide a new service. It does not require to modify the existing kernel source code or any kernel component. The main characteristic of these components are that they are implemented as an application-level processes/threads, offering some kind of service to other processes/threads (as a kind of a classical UNIX daemon).

In a similar way, the Linux layer is also split in two layers:

4. **Low-level Linux** : Like the low-level RTLinux executive components, these components modify the current kernel and has to be distributed as patch files.
5. **High-level Linux** : Components located inside the Linux kernel that use but do not modify the Linux kernel code. Device drivers are components of this category.

The third level (Linux applications) coincides with user space applications.

Most of the components will fit in one of these categories, but others will require the modification of several layers. For example, the CBS component at the RTLinux executive will also require small modifications of the Linux kernel.

3.2. Design guidelines

It is mandatory to follow some design and implementation guidelines to obtain good quality code.

List of features that must meet OCERA components:

- Open source, GPL or GPL like license. Some components may be incorporated into the RTLinux/Open distribution and in that case they will be covered by the RTLinux/Open License.
- Uniform API. The API will be POSIX like in the case that new functionalities. POSIX current tendency is to minimise the use of complex data structures in favour of object style attributes (`set/get` pair functions for each individual attribute).
- Small memory footprint. It must be optimised to minimise memory usage but paying more attention to the execution speed and predictability.
- Integrated into the main OCERA distribution.

- Minimum dependencies. The component and the accompanying examples will be as stand alone as possible.
- Optional. To reduce the memory footprint of the final target system, the user will have the option to choose only the components needed. Therefore, components will be implemented as conditional compiling or separate modules.
- Component record description. Each component will be described filling a standard record form.
- Well documented. The following questions will be considered when preparing the documentation:
  - What is the component useful for.
  - Small review of similar or related facilities in other RTOS.
  - How it can be used: is it a patch, a stand alone module, a thread.
  - Configuration parameters if any.
  - Complexity analysis, both temporal and spatial memory worst case analysis.

- Installation instructions, in the case that the component could be used independently of the OCERA distribution, dependencies with other components from OCERA or external developers will be specified.
- Usage examples.
- Regression tests, used to validate the correct implementation of the component.
- Cross partner validation. Every component will be developed by a partner and reviewed by a different one.

Chapter 4. OCERA Architecture

The OCERA architecture does not focus on a single and specific class of real-time or embedded systems, but it is aimed to provide a complete and flexible framework capable to be adapted to a wide range of applications. This section will describe the different OCERA Architecture configurations that OCERA components will support.

4.1. Hard real-time system configuration

In this configuration the whole application runs in the RTLinux layer, where hard real-time performance can be guaranteed. OCERA Linux will be a stripped kernel version with the minimal functionality and memory footprint just to boot the system and execute simple background tasks.

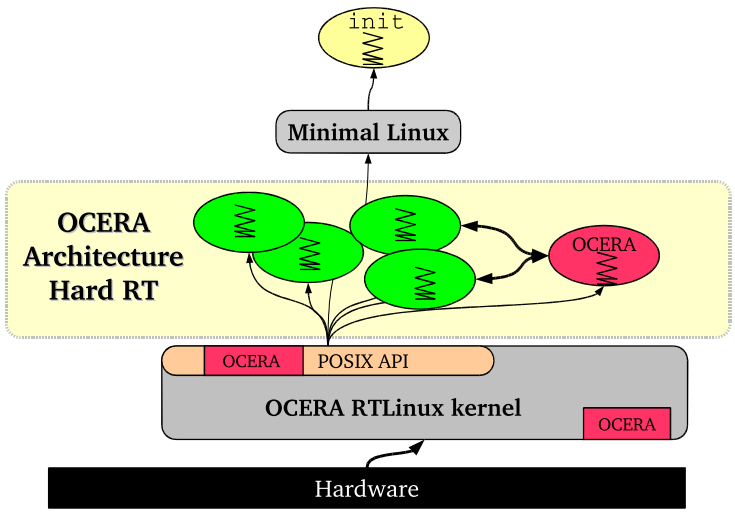


Figure 4-1. OCERA Architecture, hard real-time support

In the Figure 4-1 figure, the Linux kernel has been intentionally drawn smaller to show that it will be a reduced kernel, and only marginal tasks will run on Linux layer.

4.2. Hard and Soft real-time system configuration

Since most complex applications require a hybrid solution, requiring at the same time both hard and soft real-time, this configuration of the OCERA architecture allows the user to organise the real-time tasks in two groups: hard real-time tasks and soft real-time tasks. This special feature is provided by a QoS component which reserves a fraction of the processor time for the Linux layer without endangering the the correct and predictable execution of the hard-real-time tasks.

Hard real-time tasks will give precise and deterministic control of the system, very high time accuracy and very low latency. Hard real time performance will be provided at the RTLinux layer. The main design criteria at this level is predictability and low overhead (what is necessary to build a real-time system); therefore, this layer lacks most of the

general facilities found in conventional OS but provides as much determinism as the underlying hardware provides.

The Linux layer, on the other hand, does not provide so much time accuracy, its execution depends on both the RTLinux work load and on the Linux unbounded behaviour. To minimise latency and deadline misses the OCERA Linux kernel will integrate several patches (low latency patch, preemptable patch, etc.), as well as the OCERA developed components.

The Linux layer is mainly used for serving user-space applications: in user-space, tasks can access the full range of Linux services, like drivers, debugging tools, network, graphics, file system, etc.

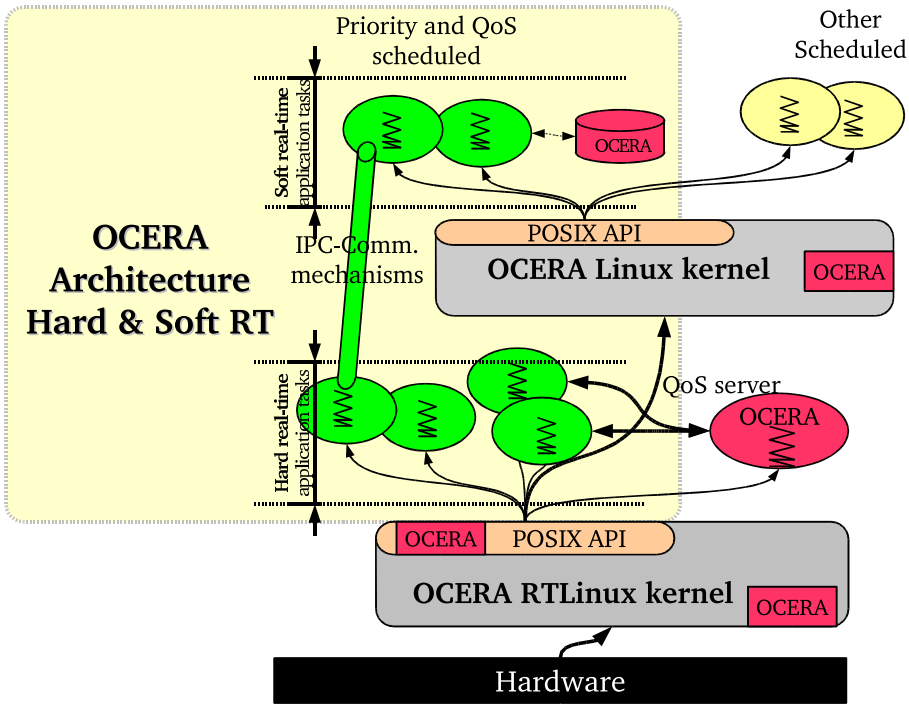


Figure 4-2. OCERA Architecture, hard and soft real-time support

Although the same API cannot be provided at both levels, some OCERA components are aimed at improving the API compatibility of both execution layers; being, at the same time, as close as possible to the POSIX real-time extensions. The benefit will be twofold:

1. It will be possible to develop real-time applications in user-space. At this level, we can use many debugging facilities like gdb, dynamic memory debuggers, etc., and the system will not hang on application bugs.
2. For some application, like applications with soft or firm timing requirements, the decision about which part of the application will run in the kernel space or in user space has not to be taken at the initial design phase, but can be delayed after the implementation, at the deployment phase.

4.3. Soft real-time system configuration

The RTLinux layer is also optional in the OCERA architecture. It can be removed if it is not required, which results in a embedded system with a single enhanced Linux OS.

This configuration is preferred in systems that do not have tight timing constrains, or where the absolute time requirements are long enough to be correctly met by tasks running in user space.

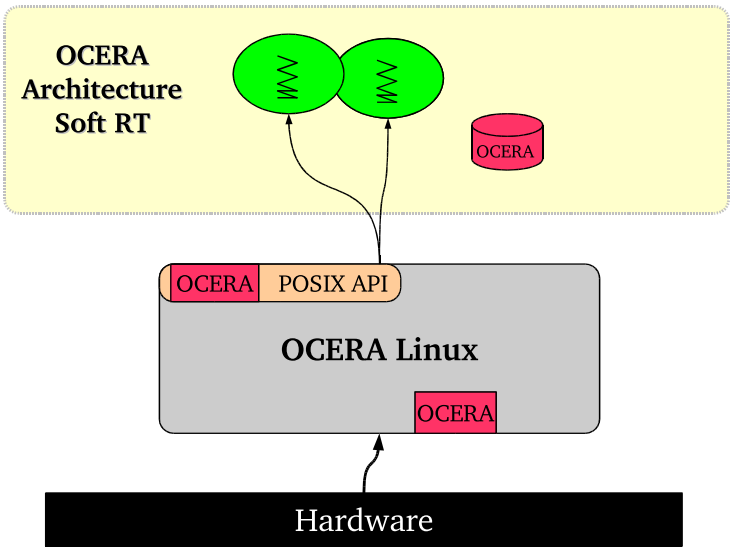


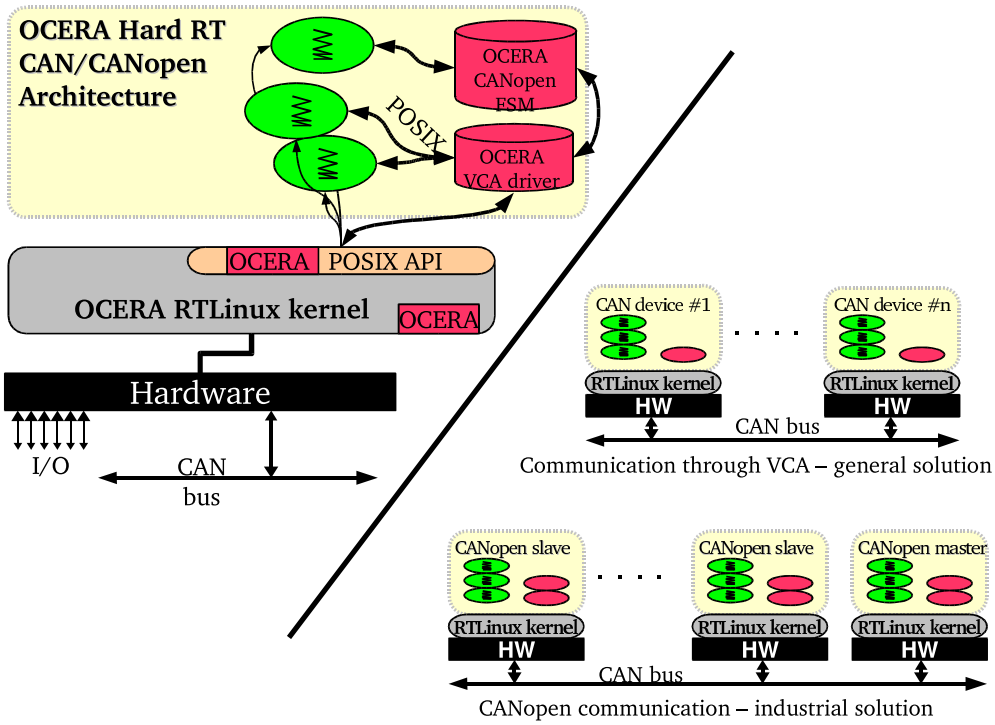
Figure 4-3. OCERA Architecture, soft real-time support

This architecture should be the first choice when starting a new application design since it is closer to the standard development methodology of general Linux applications (applications that do not directy handle hardware and have no timing requirements), and also it is a more robust execution and debugging environment.

4.4. Distributed architecture

Real-time distributed system under consideration consists of application tasks, operating system and communication support. Hard and soft real-time application requirements have strong impact on the architecture of such systems. If given real-time requirements have to be met then communication support has to be able to guarantee specific characteristics of the message delivery paradigm. The matter is quite complex since the message delivery is realized via shared media.

Specifically the deterministic protocol behaviour, the message priority and the guaranteed message delivery time are required in hard real-time distributed systems. Concise verification of such systems considering all possible states is needed when analysing hard real-time applications. Consequently CAN bus is well-suited communication support for hard real-time distributed systems and it is widely used in this application area. "OCERA Hard RT CAN architecture" is accessed via POSIX compliant VCA (Virtual CAN API). VCA offers minimal set of functions enabling to open/close/configure CAN device and to send/receive CAN message.



Since a complex distributed application possibly consists of non-Linux products (e.g. sensor - driven by programmable 8-bit micro-controllers) and various third party products (e.g. operator interface - closed non-Linux system that can be parameterised but not programmed), it is needed to use application level protocol in OCERA architecture. Choice of CANopen is based on a fact, that it is open, well documented and widely used in factory automation. "OCERA Hard RT CANopen architecture" is formed by CANopen communication standard in the same manner as it is used in typical factory automation areas. It is based on Finite State Machine (FSM), which enables to access data stored in device Object Dictionary (OD) to other CANopen devices connected to CAN bus. VCA is used to connect CAN bus and FSM.

On the other hand the network throughput and message delivery time given as statistic values (e.g. probability distribution functions) are sought in soft real-time distributed systems. Performance measures based on simulations or stress tests are usually appropriate for evaluation of these systems. Consequently RT-Ethernet is well-suited communication support for soft real-time distributed systems since it has high performance, guaranteed throughput (isolated from external traffic, publisher/subscriber architecture resulting in restricted traffic from upper layers), and wide range of supporting hardware. "ORTE (Ocera Real-Time Ethernet) architecture" is implementation of RTPS protocol originally developed by RTI. As a consequence it is composed of one "Manager" per each node and one "Managed application" per each local or remote user application. Publisher/subscriber mechanism is used to share data between user applications.

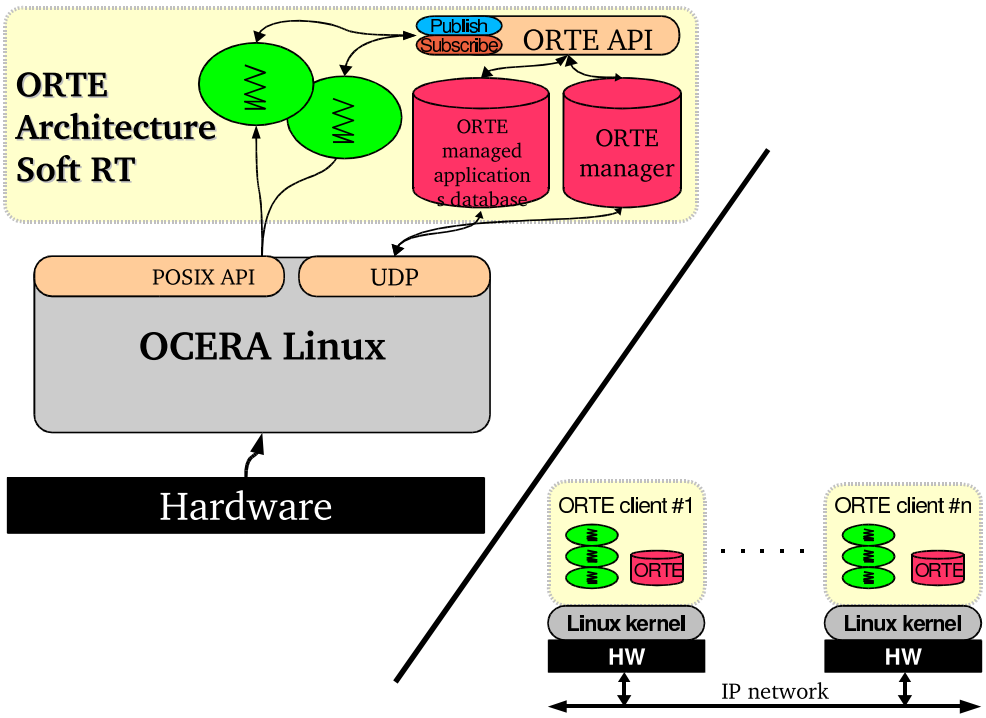


Figure 4-4. ORTE architecture

4.5. Hardware platforms

The OCERA architecture will be developed on the Intel™ x86 platform, and ported to ARM® and PowerPC® embedded targets.

x86 computers are the default hardware used on desktop systems and can also be found on some industrial applications. For that reason the x86 will be the reference hardware platform to implement and validate the OCERA components. Since most of the systems where the OCERA architecture will be used are embedded and industrial systems, we decided to support the other two processors architectures. The ARM processor family is widely used in handheld and embedded computers, and PowerPC is recognised as an high-quality industrial processor.

Chapter 5. Fault-Tolerance Management

The goal of the fault-tolerance in OCERA is intended to provide basic functionalities to develop robust applications. In order to achieve this objective, specific components will be developed allowing task monitoring, redundancy management and dynamic reconfiguration.

5.1. Applications characteristics

OCERA applications can be deployed to one or more nodes configuring a distributed system, where nodes can be connected through CAN/Open bus or RT-Ethernet network.

The application properties tackled within the project will concern: safety, liveliness and timeliness.

- Safety means that nothing bad can happen, or in other words that if something goes wrong, we can insure that a safe state can be and is reached.
- Liveliness means that the system required services are provided.
- Timeliness means that services are delivered on time.

We will not consider security issues.

5.2. Faults considered

There are several sources of faults or errors in this kind of embedded applications.

The following hypotheses will be made concerning hardware devices:

- they will be fail-silent,
- byzantine behaviours will not be considered,
- critical hardware components will have a mechanism allowing for safe stop in case of silent node.

Hypotheses on communications

- communications are atomic,
- in case of a silent node. Communications are silent.

Additionally, it is assumed that kernel crashes results in a silent node.

Prevention of such errors can be done by providing good practice programming rules that can be used for the development of tasks or drivers. Code analysis tools and specific compilation techniques with appropriate segmentation strategies can also reduce this risk. By the end of the project guidelines will be written to help developers write safe rt-tasks.

Fault-tolerance in OCERA will focus on the management of timing errors and on errors raised by the RTLinux executive. By timing errors we consider missed deadlines for time-critical tasks and watchdogs fires. We will mostly handle deadline missing and provide patterns for default actions (for example provide a default value for a periodic acquisition task restart for the next period and controlling that this does not occur more

than a given number of successive iterations). Other patterns will be developed for different types of tasks. Specific watchdogs will be used to detect fail-silent components and will thus reveal a miss-functioning in the system.

The fault-tolerance facilities will analyse the possible consequences of such errors and apply actions decided by the developer at design time and stored in run-time databases.

An application can define several operational modes that will be predefined at design time. An operational mode is an attribute of the on-board software as a whole, which purpose is to adapt the system (set of tasks) behaviour to various sets of external conditions. Tasks may have several alternative behaviours. A mode will encompass a set of tasks and a statement on which behaviour must be adopted for each task. For instance some applications may have an initialisation mode and then several possible functioning modes. Switching from one mode to another can be decided by the application on reaching a certain state or can be decided by the Application fault-tolerant monitor.

With respect to timing fails, several strategies are proposed: the task is late because of temporary overload, a task is blocked, a task is waiting for a failed node. They will also depend on the nature of the task: periodic acquisition task (in this case a default strategy can be defined by the developer), command task towards an actuator is not responding (failure can be assumed), computational task (imprecise computation strategy can be implemented). Within OCERA a global framework consisting of a set of building blocks will be developed. These building blocks will implement basic strategies in the beginning of the project and will be enriched as the project progresses.

5.3. Fault-Tolerance in OCERA

The approach chosen in the project is to provide a declarative way of managing fault-tolerance. It is based on a first stage which consists in collecting fault-tolerance requirements and default strategies at design and use this information to instantiate run-time fault-tolerance components that will monitor the application and take appropriate decisions at run-time on abnormal situations. The gathering of information and the configuration phase will be supported by a design/build tool set that will settle the fault-tolerance infrastructure.

As stated in the introduction, we will focus on the development of a framework offering basic building blocks aimed at two well identified goals. The first goal will be to provide mechanisms allowing mode management in order to increase applications robustness and achieve the safety requirements. The second goal will be to implement transparent redundancy mechanisms between nodes, which will provide continuity of service.

Two sets of components will thus be developed:

- The first one devoted to mode management consisting of two complementary components: i) The FT controller will collect low level information on ongoing tasks and notify the application monitor on tasks completions or abnormal situations. The FT controller will apply local emergency actions on faulty tasks. ii) The Application FT monitor will take global decisions on tasks reconfiguration decisions. These decisions will result in new constraints for the QoS scheduler (stop tasks, start new tasks implementing alternate behaviours, etc...)
- The second set of components will implement transparent redundancy facilities for critical tasks (declared critical by the user). Such tasks will be automatically replicated, using a passive replication method) and a specific protocol will be defined to permit synchronisation of replicas states. The controlling structure for the management of redundancy will rely on two kinds of components.

  These components will consist on: a task redundancy manager that will monitor redundancy of a cluster of replicas and decide when to activate or deactivate a replica;

and a task replica manager that is charge of synchronization and communication of a particular replica with its master for check-pointing.

As a conclusion, the expected results from the fault-tolerance work_package will consist in a set of design/configuration and run-time components permitting to enhance applications robustness. Fault-tolerance characteristics targeted are the following :

- A faulty-task will have a degraded behaviour defined at design time and implemented as a dormant task (with state updates)that will possibly be activated on error detection. This degraded behaviour will not support a subsequent failure but can be used to implement a graceful stop.
- A faulty-task or a faulty component that compromise a service may activate a degraded service implemented through a mode change activation of several tasks. Several cases will be considered : the service can be fulfilled at least partially by a different combination of tasks or not, the service is mandatory for the overall application or not. Different strategies of reconfiguration will be implemented.
- Redundancy mechanisms will permit to insure the continuity of service of a redundant task. In this case, if the master is faulty, the control will be passed onto the slave and no degraded mode will be enabled unless the last copy becomes faulty.

Chapter 6. Component description

In this section, each partner: SSSA, UPVLC, CTU, and CEA will describe by using the OCERA architecture organisation, what are the components (where are located, what do they provide, etc)

6.1. Resource Management

The aim of the Resource management components is to provide Quality of Service guarantees to real-time tasks. While a hard real-time task must be guaranteed to meet all its deadlines, a soft real-time task has less strict temporal requirements and some deadline can be missed without compromising the functionality of the application. However, it is important to "keep under control" how many deadlines are missed, and how late a soft task is going to be. A typical QoS requirement for a task is the probability of deadline miss over a certain interval; another possible requirement is a bound on the "tardiness" of a task (i.e. how much it is late in proportion to its period). According to this model, we can consider a hard real-time task as a task with a special case of QoS requirement, in which the probability of deadline miss must be 0 over any interval.

The Resource Management components are especially useful for soft real-time tasks, like for example multimedia applications. These applications are usually implemented as normal Linux processes. In fact, Linux offers many libraries, drivers and tools for this kind of applications, like sound and video drivers, a standard network protocol stack, etc. These services cannot be found in RTLinux and it is infeasible to port them.

Therefore, in the OCERA Architecture, the Resource Management components are provided at the Linux layer as additional services of the Linux kernel. They can be used both in the soft real-time configuration (if no hard real-time service is needed), or in the mixed hard/soft configuration (when both hard real-time and soft real-time tasks coexists in the system). The use of the QoS manager in the mixed configuration can be useful for those applications that consists of a hard real-time part and a soft real-time part; the critical part can be executed as a set of threads in the RTLinux executive, and must be "trusted", because a fault in a thread can compromise the functionality of the whole system; less critical parts can be implemented as Linux processes, and can use memory protection, provided by the Linux kernel, and temporal protection, provided by our QoS component.

However, in the mixed hard-soft configuration, some kind of guarantee must be provided at the hard real-time level to Linux activities. In fact, Linux is currently scheduled as background activity in the RTLinux executive, i.e. only when there is no active hard real-time tasks. If Linux is scheduled in background, no QoS guarantee is possible at the soft real-time level, because the amount of execution time allotted to Linux is not distributed evenly and Linux can be delayed by real-time tasks for large time intervals.

As a consequence, if we want to provide QoS guarantees to soft real-time tasks in the mixed hard-soft configuration, we have to reserve some bandwidth to Linux, and schedule it according to a reservation algorithm. This can be seen as a problem of hierarchical scheduling: at the hard real-time level, a scheduler selects which tasks has to be executed next; if Linux is selected, its scheduler selects which process has to be executed next.

A description of the Resource management components follows.

Low level Linux components

- Generic Scheduler Patch
  - It is a small patch for the Linux kernel that provides useful hooks to the Linux scheduler. These hooks will then be used by our scheduling module for implement-

ing sophisticated real-time scheduling policies. This patch has to be minimally invasive for to limit the overhead and to minimise the need to upgrade it for new versions of the Linux kernel.

- □ Responsable: SSSA
- □ Validator: UPVLC

- Integration patch
  - □ This patch will take into account the introduction of the "preemtpion patch" and "high resolution timers" in the Linux kernel. These new services are very useful for improving the responsiveness of time-sensitive applications in Linux: however, they are not compatible with the RTLinux patch. Since these new services are most likely to be introduced in the next Linux stable release, there is the need to take into account this problem by modifying the RTLinux executive. Therefore, this component will be a patch to the RTLinux executive that makes it compatible with the future versions of Linux.
  - □ Responsable: SSSA
  - □ Validator: UPVLC

High level Linux components

- Resource Reservation Scheduling module
  - □ It will be a dynamically loadable module for the Linux kernel, that will provide a resource reservation scheduler for soft real-time tasks in the user space. It will be based on the Constant bandwidth Server. This module is the core of Workpackage 4, and it will provided in different versions during the course of the project: a first version will simply provide the CBS scheduler for uniprocessors; a second version will include mechanisms for reclaiming the spare capacity; a third version will take into account SMP (simmetric multi-processor systems).
  - □ Responsable: SSSA
  - □ Validator: UPVLC, CEA

- Quality of service Manager
  - □ This component will provide a mechanism for identifying the temporal characteristics of a task and to adjust its scheduling parameters so to maximise its quality of service. It will be based on the concept of "feedback scheduler", that is a controller that measure the QoS experienced by the task and modify its parameters accordingly. It will be provided in two versions, as a dynamically loadable module and as a daemon process running in the Linux kernel (i.e. as a Linux application).
  - □ Responsable: SSSA
  - □ Validator: VT, UPVLC

Linux applications

- User API
  - □ This component is a set of one or more libraries that will provide a convenient API to the user to access the Resource Management services. This API will be as similar as possible to the POSIX API provided by the RTLinux executive: in this way, it

will be possible to move a RTLinux thread from the hard real-time level to the soft real-time level, and vice versa, with little effort.

- Responsable: SSSA
- Validator: UPVLC

6.2. Real-Time Scheduling

Components at the Low-level RTLinux

- POSIX Signals
  - Description:The Real-time signals facility is a deterministic signal extension that allows asynchronous signal notifications to an application to be queued with some application-specific data. This component will provide a mechanism by which a process or thread may be notified of, or affected by, an event occurring in the system. Hardware exceptions and specific actions by processes are some of the possible use of these events.
  - Responsable: UPVLC
  - Validator: CEA

- POSIX Timers
  - Description: High resolution clock access is require to program precise threads activations. This component 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. This mechanism can also be programmed in a periodic way.
  - Responsable: UPVLC
  - Validator: CEA

- POSIX Barriers
  - Description: A barrier is a simple and efficient synchronisation utility. It allows to synchronise multiple threads before trespassing an specific point. A barrier can be implemented inefficiently by mean of a mutex or a condition variable, but the proposed implementation will relay on special processor instructions to achieve low overhead.
  - Responsable: UPVLC
  - Validator: CEA

- POSIX Tracing facilities
  - Description: Tracing facilities can be used for instrumenting the RTOS and the real-time applications, allowing debugging, maintenance and performance measurement tools. Also, it can be used to register specific application events and dynamically take decisions about the system considering how it is working. It is a new POSIX debugging facility not available in other RTOS.
  - Responsable: UPVLC
  - Validator: CTU

- Application-defined Scheduler support (ADS)
  - Description: This component allows the user application to define its own scheduling algorithm. This new API will provide the capability of implementing new scheduling policies without modifying the kernel scheduler. Using this facility an application thread can decide how other threads should be scheduled. Multiple application schedulers could coexist in the same real-time application. It is a new RTOS API proposal (POSIX-like).
  - Responsable: UPVLC
  - Validator: SSSA

Components at the High-level RTLinux

- POSIX Message Queues
  - Description: This component provides a prioritised message passing facility for the real-time and embedded environments. It should be highly optimised in multithread environments (only one process and one address space). The following features have to be considered: message priority scheme, asynchronous notification and fixed size messages.
  - Responsable: UPVLC
  - Validator: VT

- RTLinux Dynamic Memory Management
  - Description: dynamic memory management is a desirable feature not available in RTLinux. POSIX system calls as `malloc()` and `free()` have to be implemented to provide to hard real-time threads and drivers the dynamic memory allocation facilities. An efficient way to manage memory considering aspects as resource reservation will be implemented. This component will provide a highly customisable and fully deterministic manager that allows hard real-time threads and low-level drivers to allocate memory dynamically.
  - Responsable: UPVLC and SSSA
  - Validator: CTU

- RTL-ADA Porting
  - Description: C is the language to develop RTLinux applications but Ada is one of the few languages with real time abstractions. This component will permit to compile Ada programs for RTLinux using some of the facilities provided by the OCERA Components as Application Scheduler.
  - Responsable: UPVLC
  - Validator: VT

Components at the Application-level RTLinux

- Earliest Deadline first (EDF)
  - Description: The EDF is a basic scheduling algorithm with a solid theory background, mainly used in multimedia applications. This implementation relies on the

Application-defined Scheduler component, which makes it almost independent of RTLinux code, and very easy to port to other OS.

- □ Responsable: UPVLC
- □ Validator: SSSA

- Constant Bandwidth Server (CBS)
  - □ Description: Using the facilities provided by the Application-defined Scheduler component, this component implements the CBS scheduling algorithm explained before. As was the case with the previous component, this implementation is highly portable since it do not modify the RTLinux executive.
  - □ Responsable: UPVLC
  - □ Validator: SSSA

6.3. Fault-Tolerance

Fault-tolerance components provide user's support for developing fault-tolerant applications. The functionalities offered will concern degraded mode management in a first stage and transparent redundancy implementation and management in a second step.

Components at the Low and High level RTLinux

At this level no fault-tolerant component will be specifically developed but the POSIX Tracing, POSIX Signals, POSIX Timers and Application Defined Scheduling components described in the Scheduling section will contribute to fault-tolerance thanks to the logging and signalling facilities they will offer.

Components at the Application-level RTLinux

- FT controller
  - □ Description: The FT controller will be in charge of collecting information on the critical tasks behaviours and controlling their lifeliness and timeliness. This low level component notifies the Application FT monitor (at the Application-level Linux) of abnormal situations and will activate emergency actions when required. In connection with the Low-level RTLinux executive, it will detect fail silent situations through watchdogs and transmit deadline misses to the Application FT monitor.
  - □ Responsable: CEA
  - □ Validator: UPVLC

- Application FT monitor
  - □ Description: The AFT monitor consists of a module that records Task Information and defines reconfiguration strategies. It will collect information from the system and tasks. It will decide of reconfiguration on notification of abnormal situations (from the FT controller) and ask RT scheduler to stop tasks and provide QoS Scheduler with a new task set (this set may include already existing suspended tasks).
  - □ Responsable: CEA
  - □ Validator: UPVLC

- Task Replica Manager
  - Description: This component will locally monitor interactions of a local replica of a task with the Task Redundancy manager. It will operate chek-pointing, and local activation / deactivation of a replica. Passive redundancy will be implemented.
  - Responsable: CEA
  - Validator: SSSA

- Task Redundancy manager
  - Description: The task redundancy manager monitors redundancy of a cluster of replicas, and decides when to activate or deactivate a replica.
  - Responsable: CEA
  - Validator: SSSA

Components at the Application-level Linux

- Design tool
  - Description: This design tool allows the user to express non-functional features. The first step will be the definition of a description language in order to specify explicitly timing characteristics and constraints of tasks, possible alternatives for tasks, actions to be done on temporal faults, and on errors. The possibility to specify tasks graphs will be offered. Support for imprecise computation will be offered. A way to specify critical tasks requiring redundancy will also be defined. The tool will permit to gather this information along with information on mapping requirements for tasks. Ways of specifying modes at task and application level will be considered. A result of this language definition task might be a UML profile for fault-tolerance. The acquisition tool itself will be developed using standard GUI programming tool.
  - Responsable: CEA
  - Validator: CTU

- Buiding tool
  - Description: The building tool uses information gathered by the design tool and configures the OCERA platform, it will provide tasks information to kernel schedulers, instantiate FT mechanisms (AFT monitor and FT controllers), and adapt tasks code. When redundancy will be tackled it will produce code for tasks duplication and check-pointing mechanisms.
  - Responsable: CEA
  - Validator: VT

6.4. Communication

Components at the High-level RTLinux

- CANopen device
  - Description: OCERA RT CANopen device is a software solution based on OCERA RT Linux and VCA capable to exchange its data with any industrial CANopen de-

vice following the CANopen communication standard. It can be configured to work as CANopen master, CANopen slave or CANopen NMT master. Type of CANopen device is specified by loading appropriate Electronic Data Sheet (EDS) into device Object Dictionary (OD).

- □ Responsible: CTU
- □ Validator: UC

Components at the Application-level RTLinux

- Virtual CAN API (VCA)
  - □ Description: The Virtual CAN API introduces CAN network to the application threads. Application threads can reside in the soft real-time space or in the hard real-time space. VCA offers minimal set of functions enabling to open/close/configure CAN device and send/receive CAN messages.
  - □ Responsible: CTU
  - □ Validator: UPVLC

- EDS parser and CAN/CANopen analyzer
  - □ Description: This component captures the traffic on CAN bus and analyze it on the level of CAN massages. A CANopen device Electronic Data Sheet (EDS) can be loaded into analyzer. In that case the analyser can send and receive SDO communication objects and show its impact in a device Object Dictionary (OD). The analyser olso offers basic CAN open NMT functionality.
  - □ Responsible: CTU
  - □ Validator: UC

Components at the Application-level Linux

- Real Time Ethernet (ORTE) device
  - □ Description: The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. This protocol has already been submit to IETF as an informational RFC and has been adopted by the IDA group. RTPS is new application layer protocol, which is built on top of standard UDP stack. This protocol stack adds real-time capabilities to standard Ethernet technology. Publisher/subscriber mechanism is used to share data between user applications. The component will be a library linkable against the user applications.
  - □ Responsible: CTU
  - □ Validator: SSSA

- Real Time Ethernet analyzer
  - □ Description: This component enables to developer to capture network traffic and to analyse it on the level of Ethernet frames, IP and UDP datagrams and RTPS messages. Real Time Ethernet analyzer is not stand alone application, but it is a plug-in module for Ethereal network analyzer.
  - □ Responsible: CTU
  - □ Validator: SSSA

Verification

- CAN model by timed automata /Petri Nets
  - □ Description: This component is theoretical study offering methodology tool support for analysis of distributed system consisting of n independent processors and deterministic communication bus (CAN). In order to verify distributed RT system, application designer needs to create model of application tasks and to interconnect this model with communication bus model provided by this component. Finally he/she needs to define system properties to be verified (deadlock, missed deadline etc.). The approach is illustrated in the form of examples in PEP verification tool.
  - □ Responsible: CTU
  - □ Validator: CEA

- Verification of cooperative scheduling and interrupt handlers
  - □ Description: This component is theoretical study offering methodology and tool support for model checking of real-time applications running under multitasking operating system. Theoretical background is based on timed automata by Allur and Dill. As this approach does not allow to model pre-emption we focus on cooperative scheduling. The cooperative scheduler under assumption performs rescheduling in specific points given by "yield" instruction in the application processes. In the addition, interrupt service routines are considered, and their enabling/disabling is controlled by interrupt server considering specified server capacity. The server capacity has influence on the margins of the computation times in the application processes. Such systems, used in practical real-time applications, can be modelled by timed automata and further verified by existing model checking tools. The approach is illustrated in the form of examples in the real-time verification tool UPPAAL.
  - □ Responsible: CTU
  - □ Validator: CEA

Chapter 7. Glossary of terms

Please, add all the terms that you think it may be useful to be here:

User space

The execution environment (characterized by restricted privileges, addres-space protection, etc.) in which Linux applications run.

Kernel space

The execution environment of the Linux kernel (maximum privilege, no address-space protection, etc.)

Linux module

A object file which can be dynamically linked (and unlinked) into the running Linux kernel with the `insmod` command. A module can access to all the Linux kernel functions and data structures (if they are exported).

Linux kernel

The kernel, as released by Linus Torvalds at kernel.org. The Linux kernel version currently used in the OCERA project to is 2.4.18. At the end of the project, the version will probably be upgraded (depending in the kernel evolution) and all the components will be ported to the new version.

RTLinux layer

The term “RTLinux layer” is used for identifying the RTLinux exectuive and the set of real-time tasks using it.

Linux layer

The term “Linux layer” is used for identifying all the code running in kernel space that does not depend on RTLinux.

OCERA component

A piece of software that brings some new functionality or feature in some of the fields: Scheduling, Quality of Service, Fault-Tolerance and Communications. Depending on the type of facility and its role, a component can be: a patch, a stand-alone module, a library, or a thread.

OCERA framework

The development environment provided to the final user for building and installing applications using OCERA components.

RTLinux executive

The Linux patch and the set of kernel modules that provide the RTOS functionality out of the scope of Linux kernel. In the strict sense, it is not an operating system, since it can not boot nor have many of the facilities required take full control of a computer. RTLinux only manages the set of hardware devices required to provide deterministic timed behaviour.

Open RTLinux or RTLinux/Open

The version of RTLinux released by FSMLabs covered by the Open RTLinux license. It is a small RTOS which coexists with Linux kernel and intercepts the low level interrupts and processor control instructions which allows to have the control of the computer at any time independently of the Linux kernel state.

OCERA Linux kernel

The Linux kernel containing existing, as well as the OCERA developed, patches to enhance the real-time capabilities. This kernel will be considered as a Soft Real-Time system.

RTLinux/GPL

Same than RTLinux/Open. FSMLabs use both terms interchangeably.

RTLinux/Pro

The commercial version of RTLinux developed and distributed by FSMLabs.

Task

A executing unit, which can be a normal process or a thread.

User Space Task (or User Space Application)

Task (or application) running in user space, that uses the Linux services only by invoking system calls.

RTLinux Task (or RTLinux Application)

Task (or application) running in kernel space, that directly uses the RTLinux services

Bibliography

[Cooperstein02] Jerry Cooperstein, 07/11/2002, The O'Reilly Network, *Linux Multi-threading Advances*.

[RTLinux.org] Der Hofrat, *Open Source RTLinux Repository*.

[RTAI] Paolo Mantegazza, *RTAI Home page*.

[L4] *DROPS - The Dresden Real-Time Operating System Project*.

[Dankwardt00] Kevin Dankwardt, 11/2000, Linux Devices.com, *Comparing real-time Linux alternatives*.