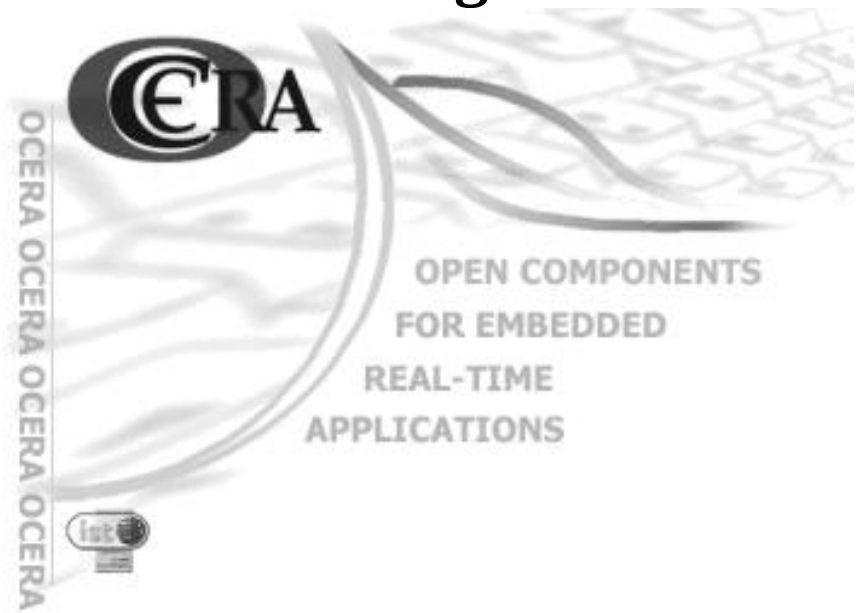
WP8 - Integration



Deliverable D8.2_rep – Integration V2 and configuration tool

WP8 - Integration : Deliverable 8.2_rep – Integration V2 and configuration tool
by Giuseppe Lipari, Davide Pagnin, Luca Marzario, Paolo Valente, Ismael Ripoll

Published April 2004

Table of Contents

Document Presentation

Project Coordinator

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

Participant List

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

Document version

Release	Date	Reason of change
1_0	19/04/04	First release
2_0	15/12/04	Second release

Chapter 1. Introduction

1.1 Summary of contents

In this deliverable, we summarize the work that has been done in workpackage 8, regarding the integration of the OCERA components and modules to produce a single software package. In this chapter, we summarize the objectives of the work and a description of the cooperative development process. In Chapter 2, we describe the base hw/sw platform for OCERA and the work performed on the components to integrate them together. In Chapter 3, we describe the configuration process. Finally, in Appendix A, we describe how to manage the CVS repository to produce a new release.

1.2 Objectives

The main objectives of the Integration workpackage is to provide a software package that contains the OCERA software, and a configuration tool to configure compile and install the OCERA software.

OCERA consists of many components: the Linux kernel, the RT-Linux executive, a set of hard real-time components (the outputs of WP 5), a set of QoS components (the output of WP 4), a set of fault tolerance component (the output of WP 6), and a set of communication components (the output of WP 7). All these components must coexist, without conflicts in the same software package.

At the same time, OCERA aims at providing flexibility to the user. The objective is to customise the software package to the needs of the application. It must be possible to develop time-critical hard real-time applications as well as QoS based soft real-time applications. In some case, in the same application, hard real-time tasks and soft real-time tasks must cooperate.

1.3 Status of the integration

We have released the software package OCERA, version 2.0, which is available for download at the Sourceforge web site (www.sf.net) and at the ocera web site (www.ocera.org). The package includes the Linux kernel, with some modification (patches), the RTLinux executive, and all the components developed by the OCERA partners. Also, it includes a configuration tool (Qconf), and development tools to aid the development of applications on top of OCERA.

1.4 Cooperative development

The development of OCERA software is not over yet. While we want to provide support and bug fixes for the current release (2.0), we also would like to continue developing more components and enhance the current existing components. However, to reduce the effort it is important to keep the modification (bug fixes and enhancement) as much centralized as possible.

The cooperative development of OCERA was carried out using the CVS provided by the Sourceforge web-site. The CVS repository contains the code of both Linux and RTLinux (with all the patches already include) plus the code of all the components and tools developed by the OCERA consortium.

Although the CVS system is not the most advanced system for cooperative development, it allows a good degree of flexibility and was considered sufficient for the scope of this project. In particular, the CVS system allows the management of versions, branches and releases.

According to the plan discussed in Deliverable D8.1_plan, the main idea is to use one single repository where all the development is carried out. When the consortium agrees on releasing a new version of the software, a two branches are created: on one branch (the bugfix branch), the stable version of the released software is maintained. On this branch, no major development is done, only bug fixes. On the other branch (the development branch), the next version of the software is developed, which can include adding new features, re-writing part of the code, and so on.

The process of making these branches is described in Appendix A, and it can be used as a reference for the future releases of OCERA.

Chapter 2 Structure of the software

2.1 Development Platform

For the first release of OCERA (1.0.0), we chose to develop the software based on the Linux kernel, versions 2.4.x. In particular, the consortium decided to support the 2.4.18 release. When the choice was made, the 2.4.18 release was the more stable and widespread among the main Linux distributions addressed to production servers. Although a completely new version of Linux (2.6.x) has been recently released, the chosen version is still widely used in production platforms and it is considered to be stable. In the future, we plan to port most of the software developed in OCERA to the newest, up-to-date version of Linux.

Another important choice was the software development environment, and in particular, the compiler and related programming utilities. At the beginning of the project, the consortium chose to develop the OCERA software on a Debian "Woody" distribution, and using the GNU suite of development tools, specifically the gcc 2.95.3 compiler (later became gcc 2.95.4).

It is important to point out a problem caused by the choice of the compiler suite. In fact, the most recent development of the GNU compiler suite (versions 3.0 or higher) introduced substantial changes to the possible command-line options and programming constructs. This change was due primarily for standardization reasons: according to the latest evolutions of the ANSI C and C++ standards, some construct that were previously discouraged but permitted, are not allowed anymore. Initially, such old construct were marked as "deprecated" but allowed anyway, to keep the compatibility with the previous versions of the compiler. However, with the latest releases of the compiler (3.2.3 and 3.3.2) such constructs are not allowed anymore.

Such a change has a major impact on the OCERA software. As a matter of fact, the release 2.4.18 of the Linux kernel (as downloaded from the www.linux.org web-site) cannot be compiled correctly with gcc 3.2.3 or higher.

The choice made by the OCERA consortium was to test OCERA Linux kernel, and, more in general, the whole project, also with a gcc reference release of the series 3.x.x. Therefore, some effort has been made to make the OCERA software compilable with these releases of the compiler.

As a consequence, at the time of the first release of OCERA (1.0.0), the whole project has been compiled and tested with both gcc 2.95.3 and gcc 3.3.2, and it has proven to work correctly within the limits of the components that we could test. In fact, it must be noted that, due to the extent and the high number of options of the 2.4 Linux kernel, we could not perform exhaustive tests. We tested only the main components and options that are commonly used in the base configuration. We expect that more testing will be provided by the users of the OCERA kernel. As soon as new incompatibilities are found, we plan to correct them.

2.2 Update

At the date of release of this deliverable, we have released a second version of OCERA, (2.0.0), by updating the Linux kernel version to 2.4.27 and the RT-Linux version to xxx. Also, all components have been updated to the last version.

2.3 Structure of the software

The OCERA software distribution is organized in a directory tree. The main directories are:

- **kernel;** this directory contains both the Linux OS and the RTLinux executive, including their configuration files.

- **components**; this directory contains all components developed by the partners of the project, and is divided into sub-directories including the scheduling, QoS, communication and fault tolerance components. It must be noted that the components in the scheduling directory are already patched in the kernel. Therefore the code inside this directory need not to be compiled and is distributed here as a reference. See Section 2.5 below for more details.

- **emdebsys** contains the configuration tool, qconf, and a tool for producing images for the target system. This second tool has been developed independently and has been imported into the project.

- **app**, contains application software. In the next future it will contain the project case studies.

- **doc**, contains the software documentation.

It must be pointed out that the main result of this integration work was to re-organize all this software so that the user can issue all important commands (configuration, compiling and installation) from the root directory. To this end, all makefiles, and in particular Linux and RTLinux makefiles, had to be modified properly.

In particular, one objective was to make it possible to release a new version of a component independently of the OCERA release. As discussed in deliverable D8.1_plan, we distinguish two kind of releases:

- a main OCERA release consists in a software package containing all OCERA software

- a component release consists in a package containing just one single component.

To make it easy a component release, we confined all configuration files and makefiles for a component in its own directory. In this way, upgrading just one component is as easy as unpacking the package file in the right place. This was possible thanks to the configuration system, which will be discussed in Chapter 3.

2.4 The Linux kernel and related patches

To provide and support all features expected by the OCERA kernel, the Linux kernel needed to be modified accordingly. Such modifications are known as “patches”. Such modifications improve substantially some aspect of the kernel and prepare it for the features provided by our components. As an example, the QoS components, which are implemented in the Linux kernel, need a precise time reference to work correctly. The standard Linux kernel does not provide a fine-grained time measurement: the basic operating system tick is set equal to 10 msec. Although such constant can be modified (and the kernel re-compiled with the new constant), reducing the operating system tick too much can result in a higher overhead. Therefore, we decided to apply the “High Resolution Timers” patch. This patch provides precise time references and timing event handling with very high precision.

However, applying a software patch is not an easy task. In fact, when many patches need to be applied on the same software, many conflicts and incompatibilities can arise.

In order to provide an integrated and working product, without leaving to the developer or to the user the problems related to the applications of several patches, we selected and applied various patches to the kernel, and solved any related conflict.

In particular, the following patches have been applied:

Big Physical Area

With this patch, it is possible to reserve a contiguous portion of memory, even very large, inside the kernel. Such patch is needed to support the WP5 components. In particular, such patch permits to implement the Dynamic Memory Allocator.

Low Latency

The execution of long instruction sequences surrounded by spinlock, inside the kernel or inside the kernel drivers, increases significantly the latency of the kernel. In fact, such critical sections run at the maximum privilege level and with interrupts disabled. The Linux kernel (especially the many device drivers) has many of these critical sections. As a consequence, in some case the latency of a kernel operation can be as higher as 100 msec. This patch introduces special points of lock release, chosen in such a way to not break the consistency of the kernel or of the drivers. At the same time, such release points contribute to reduce the maximum and mean latency of the kernel. Such a patch is particularly useful for the component that work at the Linux level, like for example the QoS components.

High Resolution POSIX Timers

Linux kernel timers do not comply with POSIX standards and, more generally, have a rather limited precision, in the order of the tens of msecs, whereas, thanks to the rapid processors evolution and, above all, the increase of clock frequencies, it is now possible to use timers with nanosec precision. This patch enables to use timers compliant with POSIX standards, and with a higher precision than the timers usually employed in Linux kernel. Such a patch is used by all components that work at the Linux level, like for example the QoS components.

RTLinux support

This patch introduces the hooks needed to enable the use of RTLinux as a Linux module, and, consequently, the use of other modules of RTLinux. Such patch is absolutely necessary for all hard real-time components, like the scheduling components developed in WP5.

Generic Scheduler support

This patch introduces the hooks needed to permit the use of a generic scheduler in the Linux kernel. A new scheduler can be loaded as a kernel module, and can be a substitute to the standard Linux scheduler. This patch is necessary to the QoS components.

In as much as possible, the above mentioned patches enable the possibility to employ some new kernel functionalities. However, all patches can be conditionally compiled. During the configuration phase, it is possible to select which one of the above patches must be enabled. Hence, using such features is not mandatory and, if desired, they can be excluded from compilation.

In this way, we add additional flexibility to the OCERA kernel. The user can choose exactly what it is needed for the application, and customise the kernel to a very fine-grained level.

2.5 The RTLinux executive components

These components include mainly modification, enhancements and new features to the RTLinux executive.

During the analysis phase of the RTLinux components, we detected several sources of potential problems:

1. Several RTLinux components had to modify the same source files (mainly the `rtl_sched.c` file) for different reasons (several non-related schedulers, timers, signals, etc...).
2. There is a large number of specific RTLinux components proposed and implemented (8 components developed in the first phase and 6 in the second phase).
3. Each component should be as independent as possible from RTLinux to allow easy porting to other OS's.
4. Dependencies among developed components. Some components can only be developed after more basic components has been finished.

In order speed up the development of each component, allowing developers to work in parallel without the mess of coping with bugs of others components, ee decided to develop each component in a completely isolated source tree and then merge all the code in the integration phase.

To avoid or minimize the problems during integration the following procedure to write components has been followed:

1. Start developing the component using the clean original RTLinux-3.2-pre1 source code.
2. If needed, apply the required modifications to the original code. Some components depend on the functionality provided by others. For example to implement POSIX timers, the POSIX signals component has to be already installed.
3. Follow the design style of RTLinux as much as possible. That is, do not change or rename any original RTLinux source file, and follow the RTLinux directory organization.
4. All new code will be surrounded at least by “C” preprocessor conditional macros, which will allow the user to select these new features.
5. Once the code of the new component has been completely written and tested, then the final component will be packed.
6. A “**public component**” will be a tar file (or a directory structure) with the following contents:
    1. `README` file with at least the following information: author/s, description, usage and licenses.
    2. `patch/` directory with the patches that this component introduces to Linux and RTLinux vanilla sources. This patch files will only contain modification of already existing files, that is no new file or directory will be included in the patch file.
       File names of the patch files are generated following the next pattern:
       [*TargetKernel*]-[*TargetVersion*]_[*ComponentName*]-[*ComponentVersion*]`.patch`
       For example: `rtlinux-3.2pre1_signals-0.1.patch`
    3. Newly created files will be located in a local copy of the RTLinux directory tree. The idea is that a simple “cp -r” should properly install the new files in the RTLinux target tree.
    4. `Makefile` with at least the “install” target that will copy the new component source files to the location of the RTLinux-3.2-pre1 code. This Makefile can be directly used during the integration process or be used as a script guide to do the integration.

It is important to note that RTLinux components are not patch files, but complete stand

alone source code with small patch files (which only contains specific RTLinux code). Components packed in this way are highly portable because the dependent code parts are located in the patch files.

2.6 Hardware platforms

At the beginning of the project the OCERA consortium committed to provide support for the Intel x86 architecture. Therefore, all software development and testing was aimed at providing a working software for the Intel architecture. Nevertheless, the Intel architecture is not the most popular in the embedded domain. Therefore, we considered the problem of providing some limited support for other hardware platforms. As a matter of fact, the Linux kernel already provides some support for many hardware platforms. We decided to provide a limited support for the Motorola PowerPC platform (processors MPC8240, MPC8245 and 750cx PPC processors, although it was tested just on MPC8240 by Unicontrols) and StrongArm (and its evolution, the Xscale processor). The reason is mainly due to the fact that the two case studies of the OCERA project are based on these two platforms. In particular, the case study provided by Unicontrol will be implemented on a Motorola PowerPC, whereas the case study provided by Virtual Tools will be implemented on a Compaq Ipaq, which has a StrongArm processor.

Therefore, some additional effort has been made to support cross-compilation to other potential processor architectures. The structure of the directories has been slightly re-organized for supporting cross-compilation. The operation has been done in a general way, so to facilitate future support for additional hardware architectures. The results of such preliminary work, although largely incomplete, are encouraging.

The porting of the OCERA software on the PowerPC platform, and in particular a VMP1 board with a MPC8240 processor, required to add a small patch to the base OCERA distribution. The resulting kernel has been tested (by partner Unicontrol) and it results to be usable, except for the application of some minor patches to OCERA base code. Unfortunately, the porting of the scheduling components could not be finalized since there is no open-source version of RTLinux available for PPC. Therefore, it is not yet possible to provide the full range of features and services provided by OCERA.

Similar encouraging result have been obtained on the StrongArm and Xscale processors. In this second case, the testing was provided by Scuola Sant'Anna on the Xscale, and by Virtual Tools on the StrongArm. Unfortunately, as in the PPC case, the development of Linux for the ARM architecture has been carried out in a tree (arm-linux) different from the main one, and the integration of the patches for ARM, at the moment, is not compatible with the use of the same tree with the other architectures.

Also in this case, within the next OCERA release, we will pursue the objective of integrating in an single tree the patches needed for the compilation on both this architecture and the previous ones.

Another problem is that it was not possible to provide the full features of OCERA for the ARM processor family. In fact, at this moment the RTLinux part of the project has been ported to the ARM processor family only in the stand-alone configuration. Therefore, at this moment it is not possible to provide a configuration of OCERA including both Linux and RTLinux for ARM.

2.7 Scheduling components

In this release (OCERA-2.0) not all components have been included, only those components that has been shown to be useful and have been well tested. The components that are not included in the release are available as separated components being possible to be added by the user.

The component list included into this release 2.0

Application-defined Scheduler	No
Dynamic memory allocator	Yes
POSIX Execution-Time Clocks	Yes
POSIX Barriers	Yes
POSIX Message Queues	Yes
POSIX Signals	Yes
POSIX Timers	Yes
POSIX Trace	Yes
Constant Bandwidth Server (CBS) Scheduler in RTLinux	No
RTLinux ide/filesystem driver	No
RTLinux Terminal	Yes
RTLinux UDP/IP	No
POSIX compliant FIFOS	Yes
Stand-Alone RTLinux	Yes
GNAT (Ada 95 compiler) porting to RTLinux	Yes
XtratuM	No

A Rational of these components is described in the next paragraphs:

ADS

The Application-defined Scheduler (ADS) is component powerful but with a complex and ex tense API which requires a deep knowledge to use it. The experience showed that this component can only be successfully used by kernel programmers, that really knows the meaning and effects of each ADS system call.

There is a new proposal (Mario and Michael) of the ADS API designed to improve efficiency. In the original proposal scheduling decisions were done by a special thread called scheduling thread, therefore every scheduling decision had to: 1) wake up the scheduler thread, 2) send the relevant scheduling information to the scheduler thread, 3) the scheduler thread analyse the information and choose the scheduled thread to be executed next, and 4) send the appropriate actions to the system scheduler to effectively lest only ready the chosen thread. As can be seen, every scheduling decision adds an overhead equivalent to two contexts switches.

The new ADS proposal describes the scheduler in a similar way as UNIX manages device drivers. The scheduler can be seen as an abstract object type (a C structure that contains pointers to functions) that provides a set of predefined functions. To add a user scheduler policy, the end user has to write the required functions of the scheduler and register in the system this data structure as a new scheduler facility. This way, it is possible to directly call the user functions from kernel code which removes the context switch overhead.

We think that the original ADS showed to work correctly, but has been obsoleted by the new

API proposal. Also a deep analysis of the structure of a library of schedulers and a complete and efficient library should be also provided to really use this component.

IDE

The RTL/IDE component is still in beta stage. We decided not include the code because it is not flexible enough. For example, the hard disk partition in hard-coded in the source code (/dev/hdc3). Also, it is better to have a highly stable code with this kind of components before they are released. A simple bug may destroy valuable data.

UDP/IP

There are several communication stack implementations: RTNET, rtsock and rtl-lwIP. Each stack was designed to achieve different goals, and has different card driver support. Some stacks provide a complete TCP/UDP/ICMP/IP support, even HTTP, while others are not so API compliant but faster. RTL UDP/IP is a simple and fast implementation intended to be used by small applications. Since the selection of one implementation or another mostly depends on the specific application requirements, we preferred to let the end user do the selection.

Circular dependency problems due to the way RTLinux is loaded.

An important change introduced in OCERA-2.0 is the unification of all RTLinux system modules into a single RTLinux runtime module. Original RTLinux runtime system was split in several Linux kernel modules:

rtl.o : Core functionality, interrupt virtualisation and management, print to console, etc.

rtl_time.o: Timer virtualisation and management (8254 and APIC). This module only provides the low level timer hardware drivers.

rtl_sched.o: Most of the POSIX threads functionality. SCHED_FIFO scheduler, thread creation, signals, timers, mutex, semaphores, etc.

rtl_posixio.o: Standard UNIX/POSIX file handling. open, read, write, etc.

rtl_fifo.o: Special communication facility. It provides buffered stream of bytes similar to the unix pipes.

mbuff.o: Shared memory between Linux processes and RTLinux threads.

rt_comm.o: Serial port driver.

This way of arranging the runtime of the RTLinux has several problems:

1.- The end user has to select the functionality he/she wants two times: first at the configuration phase, by selecting the appropriate options; and second at the startup by not forgetting to load the corresponding module.

2.- It is not clear from the configuration menu in which object file the final code related to the selected options is finnaly located. For this reason, many developers have troubles with RTLinux because they forget (or better, don't know and do not have a simple way to know) to load the appropriate module.

3.- Circular dependencies are hard to solve. Usually, a module is a driver that requires some basic functionality and provides some new functionality. This direct server client dependency implies that the "server" module has to be loaded before the "client" module. But OCERA-RTLinux is getting complex and it is not always possible to have this simple dependence chain.

The POSIX trace component jointly with the POSIX improved RTLinux FIFOS component introduced an intrinsic circular dependency. The problem is caused due to the fact that the

code scheduler has to dump the system events into a file (or something similar as a rt-FIFO). The scheduler has to send throw fifos the events produced in the system, therefore the rtl_fifo.o module has to be loaded before rtl_sched.o. On the other hand the new version of FIFO module provides both the asynchronous and synchronous operation which requires to suspend the calling thread by calling functions located in the rtl_sched.o module, therefore rtl_sched.o has to be loaded first.

The solution implemented in OCERA-2.0 has been to merge all the OCERA RTLinux modules into one single "rtl_kernel.o" module. The linker (ld) can successfully solve all the dependencies. Besides the circular dependency problem, this solution also simplifies the use of OCERA-RTLinux to new programmers.

There are minor drawbacks that has been addressed:

1.- All original examples and regression tests were designed to explicitly load and unload the required modules. Most of the original regression tests are invalid because they consisted in loading (insmod) and unlading (rmmod) each separate module many times to check whether the system gets

2.- Experienced RTLinux programmers may get confused. For this reason, we have implemented the single module system as an optional characteristic that the user can select in the configuration window.

If one single RTLinux module is selected, the whole RTLinux system is compiled into a file called "rtl_kernel.o".

Dynamic memory allocation

The use of TLSF (previously known as DYDMA) in the core of RTLinux and the use of TLSF malloc in all internal functions of RTLinux, permits to use any RTL system call from any execution space.

Original RTLinux-GPL relayed completely on the Linux memory management, both to handle virtual memory and to manage dynamic memory. For this reason, any API function that requires to allocate or deallocate memory can not be called while in RTLinux-GPL domain, that is, that API functions can inly by invoked from init_module() and cleanup_module() functions. Most of the functions that setup or create new objects (pthread_create, timer_create, rtf_resize, etc.) were Linux only functions.

The use of the TLSF allocator in all RTLinux-GPL code, makes the system orthogonal and removes the strange (and not well documented) usage restriction of some functions.

2.7 QoS components

With regard to the QoS components, we decided to reduce the dependencies between the components and the Linux code as much as it is possible. Hence, we did not try to integrate such components inside the Linux kernel tree, even if they modify the Linux scheduler.

The main reason for this choice was to try to reduce the dependency between the component version and the Linux kernel version. Also, we wanted to minimize the number of modifications to Linux in order to let the user easily get rid of the QoS components in case they are not needed.

As a result, the only functionality needed to let the QoS modules work is the patch generic_scheduler, for obvious reasons.

This choice was, for example, a key factor that allowed us to easily port the development of the code to platforms different from the x86, without the need to face, at least at the beginning, the necessity of making the compilation of the kernel itself, but relying, from time to time, on the Linux tree more suitable for the target platform.

The QoS components have been tested under the following platforms: x86, PowerPC, ARM (XScale and StrongARM). It must be noticed that, under XScale platform, the modules permit to exploit (through Algorithm GRUB) the possibility to reduce the frequency of the clock via simple hardware instructions; the introduction of similar functionalities on other processor architectures is currently under study for the next releases of OCERA.

2.8 Communication components

A similar choice, even more extreme, has been made for the communication components, in particular for the LinCAN driver and related tools, and for the ORTE components. The results of this work has been very satisfactory as the communication components could be tested under different platforms and, above all, have been tested on Linux kernels belonging to the 2.2.x, 2.4.x and 2.6.x release series.

The extreme need of flexibility required by these components was a stimulus to adopt an innovative configuration method based upon OMK, a language which extends the functionalities of make, the base compilation tool on GNU/Linux systems and, more in general, on GNU/Unix compatible systems.

Unfortunately, as unwanted side effect of this choice, we discovered then that some not enough recent version of the “make” command, even if theoretically supporting OMK extensions for configuration files, produced unrecoverable errors during the compilation phase, caused by bugs of the tool itself.

In order to overcome this problem, we chose to provide, together with OCERA source code, also an enough up-to-date version of make, as the utility needed for the compilation of the communication and prototyping components.

However, to safeguard less expert users, the configuration and compilation phase has been made automatic and, hence, no special user intervention is needed to enable the use of such utility.

2.9 Fault tolerance components

These components basically consist of a middleware layer on top of the scheduling components and of the Linux components. This layer aims at providing fault-tolerance mechanism to the applications in a transparent way. Also, an important product of this workpackage is to provide some tools to configure and automatically generate the code for an application that requires fault-tolerant behavior.

Mostly, these components are based on features provided by the scheduling components, as POSIX trace, POSIX barriers, etc. Therefore, from the point of view of integrating these components in the tree, the only problem is to make sure that the correct dependencies between the fault tolerance components and the scheduling components were set. We will discuss the dependencies and the configuration process in more details in the next chapter.

Chapter 3 The configuration tool

3.1 Structure of the configuration tool

One of the objectives of the Integration workpackage is to provide a tool that helps the user to configure the system, by selecting the most appropriate options and components. The configuration system should be as much simple and intuitive as possible, without renouncing to power and flexibility.

To this end, we chose to use the Qconf graphical tool (there is also a graphical tool based upon gtk libraries called gconf, which can be used in place of Qconf), which is the standard configuration tool adopted by the new Linux kernel version, the 2.6.0 release. Besides the graphical utilities, the "only text" interactive mode remains available, to enable to work on systems lacking of graphical libraries.

Please, notice that in this chapter we will not explain the graphical interface usage, as such information will be available in the User Guide (deliverable D10.7). However, to clarify the structure and organization of the configuration process, we will sometime show some graphical picture showing the relevant issues.

Qconf uses a new configuration specifications language lkc (a description of this language can be found at http://www.xs4all.nl/~zippel/lc/), which overcomes the limits of CML1 and CML2 languages, by making the configuration programs independent from the tree they are used to configure. This is the language used in Linux kernel 2.6.x.

This choice was appropriate since lkc offers many powerful features necessary for such a complex and configurable system. In addition, it already provides compatibility with the future releases of the Linux kernel. As a side effect, this fact guarantees that the lkc language and the Qconf tool is supported by the Linux community. In fact, after the first phase of the integration, where the first version of Qconf was used, we discovered some bugs and missing features in Qconf. Meanwhile, a new version of Qconf was being developed, and then we imported it into the OCERA project in December 2003, which permitted us to solve all the known problems.

3.2 Organization of the configuration files

The requirements in writing the configuration files were the following:

1. To separate the configuration of standard Linux and RTLinux from the configuration of OCERA. In fact, many developers are used to standard Linux configuration, and it is necessary to keep the same organization of the options. Same holds for RTLinux developers.

2. To organize the configuration options of the OCERA software in the most appropriate way for the user. We expect that most users of OCERA are not at all aware of the features of the OCERA software. Also, OCERA is a complex system, since it includes several components, each one can be configured separately. Finally, components may have dependencies with other components, or they can be mutually exclusive. Therefore, presentation of the option tree is important to guide users across all possible options of the software.

3. To permit to separately release components. It is not feasible to release the whole OCERA software every time one component changes. The entire OCERA package is released only when some major change has been done. This means that we need then to

separately release a component, so that, to a certain extent, the user need only to download an updated version of the components in which she/he is interested.

In order to achieve this goals, we organized the configuration files in a proper way. We are now going to discuss how the previous requirement have been fulfilled.

In figure 1 we show the general appearance of the configuration options.

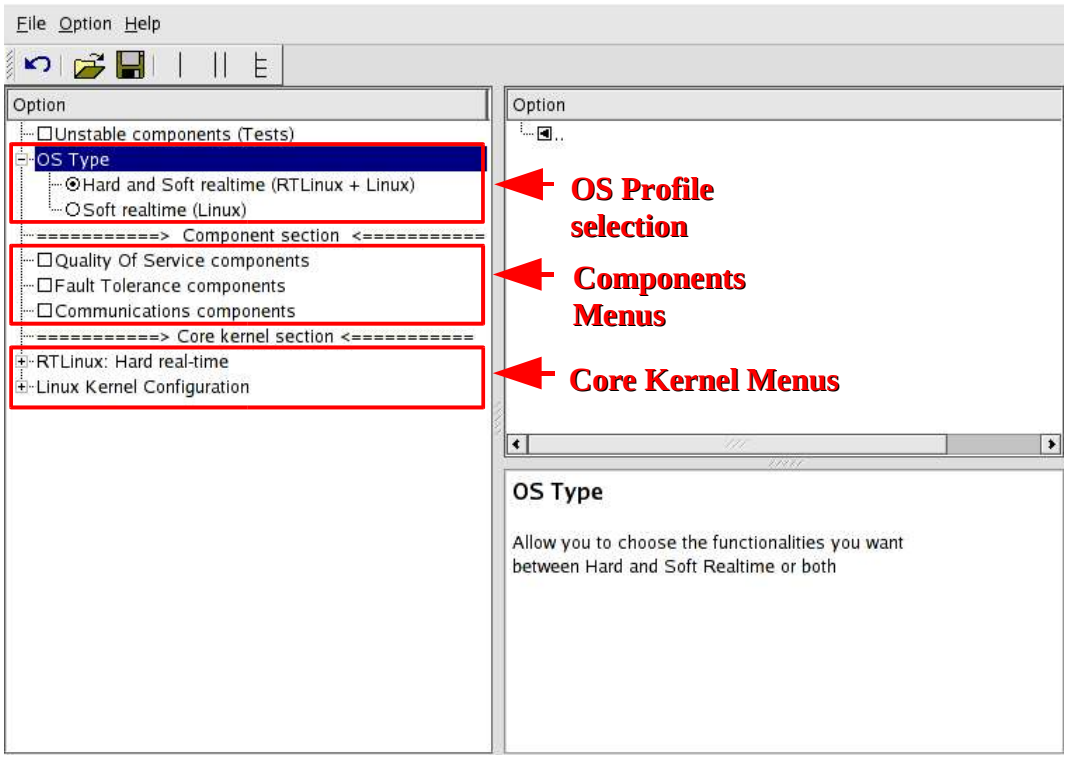


Figure 1 General appearance of configuration tree

We organized all options into three section. The first section contains the operating system profile. Currently, it is possible to choose among two mutually exclusive options: the first one enables all options that are related to the RTLinux, while the second one disables RTLinux altogether. The idea is that the user is immediately presented with the basic choice: if she/he wants to develop a hard real-time application or a soft real-time application. This simplifies the approach to the whole system.

All options related to the OCERA components are grouped in the second set of menus (QoS, Comm, FT) just after the choice of the operating system profile, and inside the menus related to the components. Again, the user is presented immediately with the basic choices of what kind of features she/he needs to select.

This choice is important because it helps the user to understand which specific option he/she is going to use and which part of the system will be influenced by such options. Besides dependencies, all the additional configuration options related to the components itself are easily selectable from the corresponding submenu.

Finally, the last group of menus is concerned with Linux options and RTLinux options. Linux options subtree is left unchanged, so the user is presented with exactly the same options as in the Linux menu. The RTLinux option tree, instead, has been re-organized a simpler way.

RTLinux options

In Figure 2 we show how the RTLinux menu is organized. Basically, we decided to include the scheduling components options in the RTLinux menu. In this way, it is very clear that the scheduling components are basically extensions of the RTLinux kernel. The options are shown on the right-upper part of the windows. In the figure, you can see that the "POSIX Signal" group of options has been selected. In the right-lower part of the window, an help text is shown that summarizes the features of the component and some caveat.

An effort has been made to help the user understanding what the options are used for. In most cases configuration option is provided with a short description text, which tries to be simple but at the same time complete.

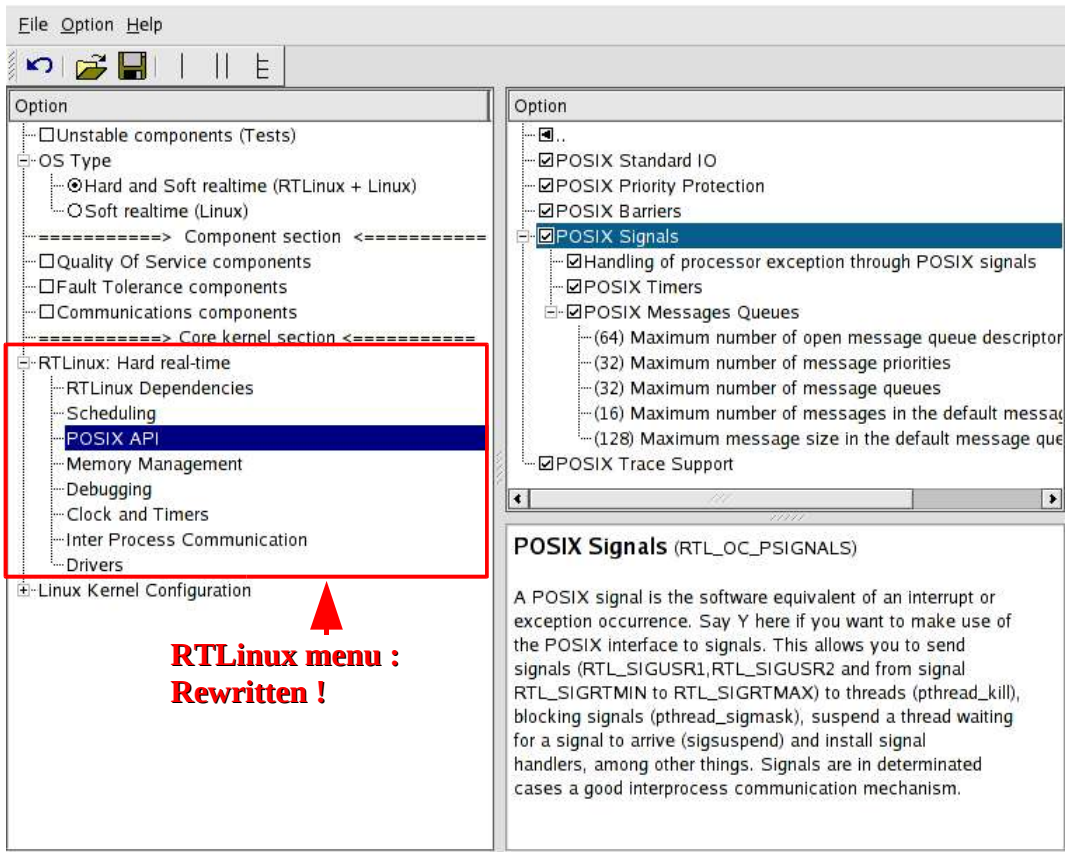


Figure 2 Details of the RTLinux menu, including scheduling components options

In this way, the user can easily remember what is the meaning of each option. Of course, it is not possible to explain all details of the components in such a small text group. However, it is possible to get some information that could convince the user to read the more length documentation. Also, most user and developers of the open source community like to use the approach "download and try it". In this way, at least they get an idea of what each component is up to.

Component sub-menus

Notice that in the left part of the window in Figure 2, under item "RTLinux: Hard real-time", a list of dependencies is reported (item "RTLinux dependencies"). Every component has such label: basically, the idea is to summarize in one place all options the selected component depends upon.

This feature is highlighted in Figure 3, where we show the QoS components's submenu. In the left part of the window. As you can see, we selected the "QoS dependencies" item, and

on the right-upper part of the window, a list of options to be selected is listed. For example, in this case, is the user wants to select any of the QoS components, the “loadable module support” option of the Linux kernel must be selected. Also, the user needs to select the “Generic scheduler” option.

Finally, again in the right-lower part of the window, an explanation of these dependencies is reported, along with a list of suggestions for the options to be selected.

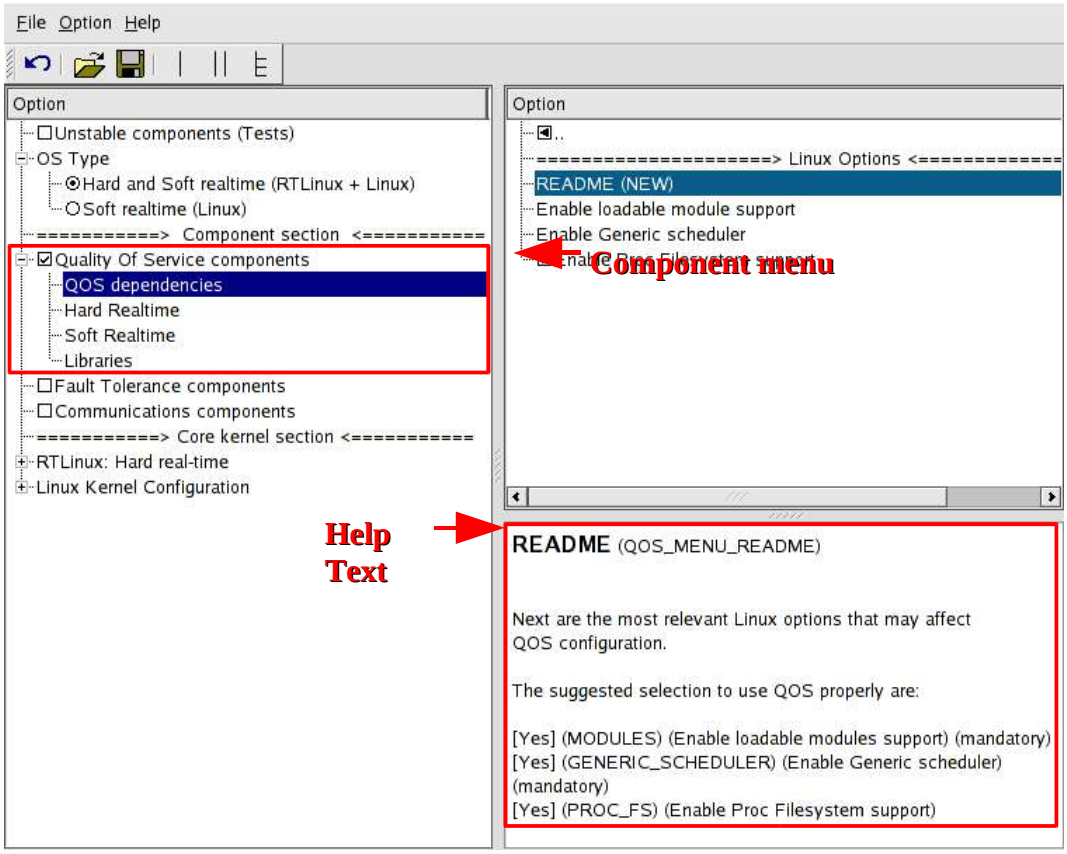


Figure 3 Example of component submenu and help text

The result of this work is the current version of the configuration tree. The description of the options and their use will be described in deliverable D10.7.

Appendix A. Managing branches and releases with CVS

How to check out a branch:

If the developer needs to work in a particular branch, because for example she/he needs to fix some bug of a particular release, type the following command:

```
cvs checkout -P -r ocera-2_0-branch -d ocera-2.0 ocera
```

where

- -P for pruning empty directories
- -d ocera-2.0 for naming the local repository in a consistent way in respect of the work that you may do with it

How to checkout a release:

If you want to checkout a particular official release (or a bugfix release or a snapshot) you may want to use this command:

```
cvs checkout -P -r ocera-2_0_0-release -d ocera-2.0.0 ocera
```

IMPORTANT WARNING:

If you plan to do some kind of development into this tree, remember that the release tag is a symbolic tag and thus you are not in a branch, this means that commits will eventually fail. For development you have to checkout a branch (or the main tree) and develop under it.

Make a release and create a bugfix branch

1. First you have to be sure that the status of the main tree is the expected one for the official release you want to make, in particular you may need to update as needed the version numbers in the relevant files. As of OCERA 2.0.0 release, we have modified kernel/linux/Makefile and kernel/rtlinux/Makefile

   Perhaps a RELEASE file into the root of ocera tree,is a better and unique point, to have the version of the whole project.

2. For issuing a release from the MAIN tree:

   ```
   cvs rtag -Fa ocera-2_0_0-release ocera
   ```

   (create tag for the release, this applies to MAIN tree and does not need a local repository, because of rtag! -F is useful for relocating the same tag if needed -a is needed for looking also in the attic)

3. After release creation, we want to create a branch that will separate development of new features for subsequent release from bugfixing of the stable release :

   ```
   cvs rtag -Fa -b -r ocera-2_0_0-release ocera-2_0-branch ocera
   ```

(create a sticky tag for the 2.0 bug fixing branch)

Now we have two branches: the main branch, ready for development of the 1.2.0 release the bugfixing branch, ready for development of the 2.0.1 release.

Make a tarball for a release

When you issue an official release (or a bugfix release, a snapshot, etc.) you may want to offer a tarball containing only the project files to end-users, you may want to use this couple of commands:

```
cvs -d $CVSROOT export -r ocera-2_0_0-release -d ocera-2.0.0 ocera
tar -czvf ocera-2.0.0.tar.gz ocera-2.0.0
```

Make a bugfix release

After some time, many bugfixes may have been found their way into CVS in the bugfix branch, when the times come you can issue a bugfix release with this command:

```
cvs rtag -Fa -r ocera-2_0-branch ocera-2_0_1-release ocera
```

(create the 2.0.1 release and applies the tag only to the proper bugfixing branch).

How to merge fixes from bugfix branch to the main tree

The procedure is slightly different, depending whether it is the first merge or a successive merge.

Instruction for the FIRST merge:

1. You need an updated main tree working local repository, then issue the command:

   ```
   cvs update -j ocera-2_0-branch
   ```

   (this merges all the changes we have made in the branch from the branch point on)

2. Then you have to tag the bugfix branch, in the case you are going to merge fixes again into the development main tree:

   ```
   cvs rtag -Fa -r ocera-2_0-branch ocera-2_0-mergedtomain ocera
   ```

3. Commit the merge:

   ```
   cvs commit ocera
   ```

Instruction for the SECOND merge:

1. You need an updated main tree working local repository, then issue the command:

   ```
   cvs update -j ocera-2_0-mergedtomain -j ocera-2_0-branch ocera
   ```

2. Commit the merge:

   ```
   cvs commit ocera
   ```

3. Move the merging reference to the current state of the branch:

   ```
   cvs rtag -Fa -r ocera-2_0-branch ocera-2_0-mergedtomain ocera
   ```