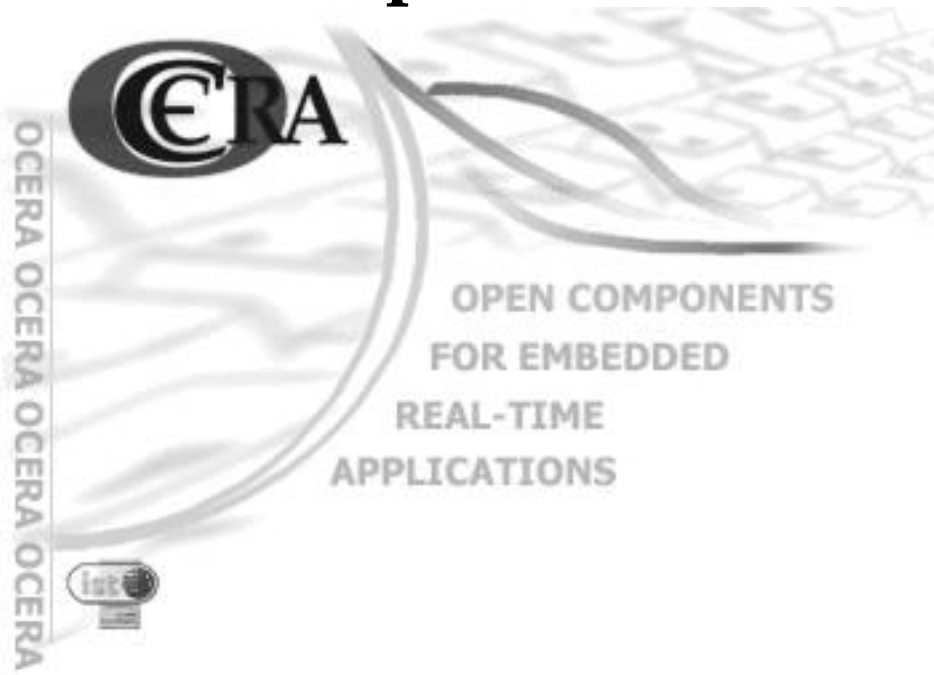
WP5 - Real-Time Scheduling Components



Deliverable D5.3 - Definition of new scheduling functionalities

WP5 - Real-Time Scheduling Components : Deliverable D5.3 - Definition of new scheduling functionalities

by Alfons Crespo, Ismael Ripoll, Patricia Balbastre, Miguel Masmano, Andres Terrasa, Agustin Espinosa, Vicente Esteve, and Alejandro Lucero

Published July 2003

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

1.1. Overview

In the first stage of this working package, most of the components where related to improve POSIX compatibility. Components of this stage will provide new features and services.

Most of the work that will be done during the next six months will be focused on the development of the new components, but also some effort will be used maintain and to keep up to date first phase already developed components. Among others it will be released new versions of signals which will include arithmetic exception signals; a new version of the Ada95 support that works with the latest version of the GNAT compiler (3.15); a code clean-up and better documented version of the dynamic memory allocator; etc.

Although not directly related to POSIX, special care will be taken when designing the API of the components to be POSIX compliant. For example, drivers will provide the classical `open`, `read`, `write`, `ioctl`, ... set of access functions. Specific features will be selectable via configuration options, when possible. The main idea is to provide new functionality with standard interfaces, so that the final user do not get stuck with RTLinux and OCERA components.

This deliverable is organised as follows: next section provides a brief overview of each component; in each of the remaining chapters it is described and justified the utility of each proposed component, also the status and some implementation issues are outlined.

1.2. Summary of new components

Execution Time Timers (ExecTimers)

Execution Time is the time spent executing a process or thread, including the time spent executing system services on behalf of that process or thread.

Among other uses, this kind of timer may be used to detect execution time overruns in the application, and to limit their effects. It can also be used to detect under-load situations.

Lightweight POSIX Trace (Lptrace)

The main proposal of the Lptrace component is to subdivide the original POSIX Trace standard in several small implementation options, in order to allow the operating system implementor to customize the desired support to applications and to allow a more efficient implementation of the tracing system itself. In order to do so, the standard has to be fully revised and then adapted to the needs of the Minimal profile.

System Metrics (Metrics)

This component will be a library capable of extracting system metrics from POSIX trace streams. System metrics are temporal or logical properties, per thread or system related. Worst case execution time, response time, longest critical section, priority inheritance correctness, etc.

RTLinux Terminal (RTTerminal)

RT-Terminal will be a new component that will allow to RTLinux applications display data directly on the console screen and read directly from the keyboard the pressed keys.

RTLinux IDE Device Driver (RTLide)

RTLinux is a small executive that lacks support of almost any kind of device. RTLinux do not have direct access to permanent storage systems like hard disks. In

order to use this kind of devices, an RTLinux application has to be helped by a Linux program running on top of Linux, that has full access to Linux kernel drivers. That is, the access to the hard disk is executed in background and by a non-real-time OS.

This component will port the IDE driver code from the Linux kernel to RTLinux. This low level driver, jointly with the file systems and the disk scheduler developed in the RTLfs component will provide to RTLinux a truly real-time file system.

RTLinux Disk Scheduler and FS (RTLfs)

This component will be split into two separate subcomponents: 1) the file systems structure; and 2) the request scheduler.

The file system will be a design of a very simple real-time file system designed to fulfil the not complex requirements of small embedded systems: no subdirectory support, predefined and fixed number of files and file space preallocated.

The disk scheduler will attend requests from Linux kernel (that will be served in background) and RTLinux tasks.

RTLinux Stand-Alone (RTLsa)

Stand-Alone RTLinux will be a Linux independent implementation of RTLinux. Stand-Alone RTLinux will be a bootable executive with the following main features: low memory overhead; scalability; easy porting; faster due to lower memory contention.

RTLinux Stand-Alone Memory Protection (RTLsaMprot)

One of the most important characteristics of a real-time system is robustness and fault-tolerance. This component implements two different memory protection models: 1) kernel memory protection; and context memory protection. The fist model assumes that the application has only one single process, and the executive is protected against application write access. The second model permits to have several separate and protected execution spaces, its functionality is almost the same than that of conventional systems.

RTLinux Stand-Alone debugging tools (RTLsaDebug)

Several debugging tools will be provided by this component: GDB agent, extensions to the GDB agent, and simple trace (non-POSIX).

The GDB agent will be communicated with the debugger (GDB) via the serial port. In order to permit the debugging of critical sections, and interrupt handlers, the driver of the serial port has been re-implemented to operate by polling (do not using interrupts).

Porting RTLinux Stand-Alone to ARM processor (RTLsaARM)

The ARM is a processor widely used in hand-held and embedded systems. It is a low power consumption but with a fast and advanced architecture.

This component fulfils two objectives: it provides support of a widely used processor; and it will improve the code structure of the original RTLinux Stand-Alone component to permit the easy porting to other processors by isolating the basic hardware dependant parts of the OS.

RTLinux UDP/IP (RTLUDP)

This is an implementation of the UDP/IP stack from scratch (not derived from BSD code as many other TCP/UDP/IP implementations). The main design criteria is efficiency, while features and compatibility are secondary.

This stack will be prepared (interfaced) to work with the network device drivers developed in the project **EtherBoot**, which has a large base of supported drivers.

The stack will also be connected with the Linux networking stack, providing a high level communication mechanism between RTLinux and Linux on the same machine.

Chapter 2. Execution Time Timers (ExecTimers)

2.1. Description

Scheduling analysis techniques assumes a very pessimistic behavior. For example, for each task in the system, it is supposed a known worst-case execution time (WCET). Nevertheless, estimating WCET is a difficult task due to the different execution paths within a program and today's computer architectures. Specially, in the context of concurrent programs in which cache misses are frequent after interrupt service routines or context switches.

Unfortunatelly the tasking model of most concurrent hard real-time systems, not enforces the bound on the execution time of tasks. Without bound on execution time, a task could execute more than estimated, causing other tasks to loose its deadlines. This, on most hard real time application, may result in catastrophic consequences.

The assumption that a task will consume its WCET in all the activations does not have to be necessarily true, which implies that the real utilization of the CPU is less than the estimated in the schedulability test.

In this component the implementation of execution time timers within the task scheduler is proposed. This kind of timers may be used to detect execution time overruns in the application, and to limit their effects. It can also be used to detect underload situations (when tasks do not execute with WCET in every activation), and the extra time detected can be used for processing future events (execution of optional computation, non-periodic tasks, ...).

Execution Time Timers work in the following way, when they are used to detect overruns: when a thread is activated, a timer is armed. The timer's expiration time is set to the thread's estimated worst-case execution time (plus some small amount to take into account the limited resolution and precision of the CPU-Time clock). The timer will only expire if the thread suffers an execution time overrun, sending the appropiate signal.

On the other hand, if we want to use Execution Time Timers to detect underload situations, then the timer has to meassure the actual time spent. Therefore, if the thread finishes its actual execution before the timer expires, it can be obtained the time between the actual time, and the time when the timer was armed, that is, the real execution time of the thread in the current activation.

2.2. Layer

Low-level RTLinux component.

2.3. API / Compatibility

The execution time clocks interface defined in the proposed standard POSIX.1d is based on the POSIX.1b clocks and timers interface used for normal real-time clocks. The new interface creates a new function to access the execution time clock identifier of the desired thread: `pthread_getcpuclockid()`. In addition, it defines a new thread-creation attribute, called cpu_clock_requirement, which allows the application to enable or disable the use of the execution time clock of a thread, at the time of its creation. Once the thread is created, this attribute cannot be modified. To use CPU-time clocks for threads, we must set the cpu_clock_requirement attribute to the value `CLOCK_REQUIRED_FOR_THREAD`.

```
#include <time.h>
```

```
int pthread_getcpuclockid(pthread_t *thread, clockid_t *clock_id);
```

2.4. Dependencies

This component depends on psignals and ptimers components.

2.5. Status

Design.

2.6. Implementation issues

POSIX defines the execution time as the time spent executing a process or thread, including the time spent executing system services on behalf of that thread. Due to the fact that in RTLinux all threads run in kernel space, system calls are implemented as simple function calls fully executed in the context of the calling thread. This OS characteristic allows to implement CPU-Time execution timers in a very simple and efficient way.

The implementation of CPU-Time clocks and timers in RTLinux requires modification of the data structure that defines each thread, the thread control block, modifications the scheduler code to include the necessary steps to update each thread's CPU-Time clock and modifications to the timers management code to operate with the CPU-Time clocks associated timers.

The information that must be added to the thread control block consists of:

- A boolean to indicate whether the thread has a CPU-Time clock is enabled or not; it is set at the time of thread creation using the value specified in the thread attributes object used to create the thread.
- A structure with the information needed for the CPU-Time clock, including the clock identifier, the time of the last activation of the thread, and the total CPU-Time consumed by that thread.
- An Abstract Data Type (array, list, ...) containing pointers to the timers associated with that thread's CPU-Time clock;

The modification required to support CPU-Time clocks and timers consists of adding code at the point where a new thread becomes the running thread. This point in RTLinux is located at the scheduler function (int rtl_schedule()) before the context switch is performed.

2.7. Validation criteria

This OS facility allows to implement today's scheduling algorithms such as CBS and sporadic server schedulers in a more efficent and reliable way, specially at application-level with the use of POSIX-Compatible Application-defined Scheduling.

The use execution time timers helps to increase the fault-tolerance of critical applications due to a system timing malfunction not detected by the off-line analysis techniques.

As in most of the low level components, component implementation must fit two objectives: minimize the overhead introduced in RTLinux runtime and try to achieve the best efficiency. In addition to this, an acceptable timer resolution respect the available hardware should be reached.

The overhead introduced to RTLinux runtime due to execution time timer management should be negligible when no CPU-Time timers are used and similar to normal POSIX timers when several exection time timers are armed.

Finally, execution time timers should guarantee the immediacy of the timer expiration notification (similar to a context witch).

2.8. Tests

During the analisis and design phase several test have been planned to check implementation correctness and the validation criteria. Furthermore, the POSIX 1003.1d Test Suite covers, among others, execution time clock tests, therefore, it will ported to RTLinux to validate implementation.

To measure the overhead introduced by execution time timers, Baker's utilization test should be applied. Also timers accuracy tests developed while implementing POSIX timers will be adapted to measure execution time timers precision.

Chapter 3. Lightweight POSIX Trace (Lptrace)

3.1. Descritpion

The experience obtained in the POSIX Trace (Ptrace) component in the first stage of the project has lead us to the belief that tracing mechanisms are very useful to the real-time application designer, even in small MRSP kernels, but also that the original POSIX trace standard is clearly over-weighted for such kernels. As an alternative of building custom tracing mechanisms, we think that the best solution to this situation is to define a lightweight tracing component which: 1) is as close as possible to the original POSIX Trace standard; 2) presents a functionality which is compatible with the hardware and software requirements of the Mininal Realtime System Profile; and 3) can be efficiently implemented (considerably reducing the overhead of the original Ptrace component).

The main proposal of the Lptrace component is to subdivide the original POSIX Trace standard in several small implementation options, in order to allow the operating system implementor to customize the desired support to applications and to allow a more efficient implementation of the tracing system itself. In order to do so, the standard has to be fully revised and then adapted to the needs of the Minimal profile.

Finally, the Lptrace component will support all the possible small implementation options defined above, including some support which was missing in the Ptrace component (as the trace to log files, or devices). However, it will be possible to incorporate (or remove) each option by defining (or not) the appropriate labels and recompiling the component.

3.2. Layer

The tracing support has two main parts: the implementation of the trace subsystem and the instrumentation of different parts of the kernel in order to generate system events. Both parts are placed in RTLinux Lowlevel.

In addition, some tracing support will be available to Linux, at the Linux Application Level.

3.3. API / Compatibility

The API provided by this component is equivalent to the API of the Ptrace component.

3.4. Dependencies

Open RT-Linux version 3.2-pre1

3.5. Status

The revision of the POSIX Trace standard in order to detect which restrictions should be placed and the identification of the small implementation options has concluded. Next steps are: (1) implementation of this component (that is, reimplementation of the Ptrace component following the guidelines of the revision), and (2) exhaustive overhead testing of the Lptrace component and comparison with the overhead in the Ptrace component.

3.6. Implementation issues

The main issue of this component is to implement the kernel trace subsystem in such a way that it incorporates only the code necessary to support the implementation options chosen at compile time. For example, if the trace subsystem is chosen to support one trace stream only, it should not suffer from any overhead related to having multiple streams.

In this sense, the module will have an exhaustive use of conditional compiling directives.

3.7. Validation criteria

There are two validation criteria: code size and overhead. The trace system has to contain the code strictly necessary to support the options chosen before the component is compiled. On the other hand, neither the trace subsystem nor the instrumentation in the rest of the kernel should suffer any overhead penalty for implementation options which are not being supported.

3.8. Tests

The first group of tests are aimed towards functional aspects of the component (testing that it is logically correct). The same tests of the previous Ptrace component will be used here.

A second group of tests will compare the code size of the trace subsystem when different sets of implementation options are supported, and also with the size of the trace subsystem in the Ptrace component.

The final group of tests is equivalent to the second one but related to the overhead produced by the component, specially in the most sensitive functions at run time: the function to trace an event and the different functions to retrieve an event.

Chapter 4. System Metrics (Metrics)

4.1. Descritpion

This component will be a library capable of extracting system metrics from POSIX trace streams. A system metric is the duration of the execution of a system service, for example the suspension of a thread which has been called the function `clock_nanosleep()`, or a temporal property such as the response time of an application task.

In addition to the generic extraction mechanism, the Metrics component will provide the set of predefined system metrics which are required to analyze the schedulability of a real-time application which is implemented using the services offered by the POSIX Minimal Real-Time System profile.

The component library will offer two operation modes: off-line and on-line. In the off-line mode, metrics will be computed from a pre-recorded trace stream (previously stored in log file). In the on-line mode, metrics will be computed at run time from an active trace stream, from which events are retrieved on-line.

4.2. Layer

The Metrics component will be built on top of the POSIX tracing services, and so, it will not be part of the OCERA RT-Linux kernel. Nevertheless, the kernel must be properly instrumented in order to extract the required system metrics.

As a result, the component's extraction mechanism and the library of system metrics will be implemented at the RT-Linux High Level, while the instrumentation required to extract the events which will be placed at the RT-Linux Low Level.

4.3. API / Compatibility

The API definition is part of the development of this component.

4.4. Dependencies

The current POSIX Trace (Ptrace) component and the Lightweight POSIX Trace component (Lptrace). The latter is a new component proposed in this stage of the project.

4.5. Status

The extraction of some temporal metrics from trace streams has been validated already in a previous work not related to the OCERA project. The API to this component is now being designed and implemented. The final step will be to define a complete list of system metrics to incorporate to the component.

4.6. Implementation issues

The internal implementation of this component will be based on finite-state machines, which will allow the extraction of metrics to be done in constant or linear time with each event considered. This permits on-line extraction of properties if necessary.

As a result, this component will contain a collection of finite-state machines, each one devoted to extract one metric, plus a generic extraction mechanism which will make the automata evolve.

4.7. Validation criteria

The first validation criteria is that the API to be proposed should be easy to use for applications.

Internally, this component should be capable of extracting all the relevant metrics needed to analyze the schedulability of a POSIX real-time application in OCERA RT-Linux.

Finally, the main source of overhead to the kernel will be the generation of POSIX trace events. In order to minimize this overhead, this component assumes that the Lightweight POSIX Trace (Lptrace) component will be preferably used.

4.8. Tests

The first group of tests will include the development of programs which use the API of this component in order to extract metrics of simple applications. This will test the logic of the extraction mechanism and the consistency of the automata computing the different metrics.

A second group of tests will include the implementation of complex POSIX real-time applications in order to validate the utility of the metrics proposed in the component.

The final group of tests will be specifically designed to test the overhead of the component at run time, in order to quantify the cost of on-line extraction of metrics.

Chapter 5. RT-Terminal (RTTerminal)

5.1. Descritpion

Currently, RTLinux human user interface capability is limited to directly print strings via functions: `conprn()` or `rtl_printf()`. Direct user input from keyboard is not possible. Complex interaction with the human user (data representation, and user input) has to be done by a non real-time Linux application communicated with the RTLinux application via shared memory or RT-FIFO's. These communication mechanisms are non-portable specific RTLinux facilities. Also, the access to the video and keyboard can not be done in real-tiem since it is delayed until no real-time task is active and when Linux became active.

RT-Terminal is a new component that allows to RTLinux applications display data directly on the console screen and read directly from the keyboard the pressed keys.

RT-Terminal will provide direct access to the console to RTLinux applications and at the same time keep compatibility with the existing Linux console drivers. To maintain compatibility with current Linux system, RTLTerm will intercept the Control-Alt-F9 key combination to switch the control from RTLinux console to normal Linux processing.

RT-Terminal will also be used by the Stand-Alone RTLinux component. Since standalone can not use the linux kernel to print on the screen, it has use this new component to perform this task.

5.2. Layer

Low level, between the linux kernel and the hardware

5.3. API / Compatibility

The implemented API will be basic POSIX standard (open, close, read, write) and a subset of the Xterm (VT100) escape sequences in order to provide a compatible but fast and compact terminal control.

5.4. Dependencies

RT-Terminal depends of Linux Kernel, since in order to catch the keyboard interruption, it patchs the Linux Kernel.

5.5. Status

Currently RT-Terminal is in alpha status.

5.6. Implementation issues

Two different behaviours, selectable at compile time via a configuration option, will be provided:

- Screen and keyboard is managed by Linux (at background or slack time). Threads output buffered and only effectively printed on the screen when Linux became active.
- Read and write functions are implemented completely in RTLinux, and the output is performed immediately before the write function returns to the calling thread.

Both implementations will provide the same API, so the only difference will be when characters are printed or read from keyboard, whether in non-real-time or in real-time.

When the RT-Terminal is inserted inside RTLinux, it is registered as a POSIX device and it is visible to RTLinux applications as a standard RTlinux device on "`/dev/tty`".

In order to implement the RT-Terminal, the implementation can be divided in two halves.

- Keyboard reading, it can be implemented using one of the following methods: (1) intercepting keyboard interrupts, reading the value of the keyboard chip register and if it is not a key for the RTLinux handler, then write-back the key-scancode to the keyboard and call the normal Linux handler; (2) consists on patching the kernel keyboard handler interrupt so that if the pressed key is the expected key, the handler will send it to the RT-Terminal.

  Since some PC keyboard controller are not fully compatible (scancode can not be written back to the controller), the RT-Terminal will be implemented using the second method.

- Screen writing, it will be implemented taken the control of video memory and writing directly above it. Screen writing must be a non-blocking operation.

Initially RT-Terminal begins in Linux mode, all the data written by the RTLinux application is introduced into a buffer and all the pressed keys are passed to the Linux Kernel. When user press the F9 key, the RT-Terminal mode changes to RTLinux mode, the current screen are stored in a buffer and all the buffered informations are showed on the screen, in RTLinux mode all the pressed keys are passed to the RTLinux application that read from the RT-Terminal.

5.7. Validation criterias

RT-Terminal is a component that must not affect the correct behavior of a hard real time application, i.e. all the applications that had a correct time behaviour before using RT-Terminal, they must have it after using RT-Terminal.

In order to validate screen writing it will be tested that all the data sended to the RT-Terminal by means the `write()` function are showed correctly on the screen.

The validation criteria for keyboard reading is that the RTLinux appication must receive all the data that user is sending to it throught the keyboard.

5.8. Tests

Many tests which use intensively the screen and also read from the keyboard, must be done in order to validate RT-Terminal. Several families of test can be implemented:

- Screen tests: These tests must use intensively the screen, representing all types of characters, one possible test can be, several high priority RTLinux threads running and printing on the RTLinux without blocking theirself.
- Keyboard tests: These tests must get to the RTLinux application user information. These test have not to use RTLinux facilities in order to represent data on the screen, they can use `rtl_printf` or `con_prn` in order to do it.
- Screen + Keyboard tests: Once both RT-Terminal features have been tested in a separated form, both must be tested together. These test will demand user information and later they will representate it or a derived of it on the screen.

Chapter 6. IDE Device Driver (RTLide)

6.1. Description

The access to the device is at the end of the path for read/write requests to a storage device. Here a device driver is needed. This device driver has the knowledge about how IDE devices work, and what kind of structures and commands must be used. As Linux has a device driver for IDE disks, and the functionality of our device driver does not change with regard to how it works inside Linux kernel, we could use the Linux code. But the problem is how this device drivers is related with other structures of Linux kernel, as we explain in File System component description. So, we need to implement a IDE device driver for our purpose.

6.2. Layer

This is a High-level RTLinux component.

6.3. API / Compatibility

IDE device is not user oriented. Only disk scheduler can use it. Since disks have three ways to read/write blocks, single sector, multiple sectors(16 maximum) and DMA(128 sectors maximum) will have three different routines to call from disk scheduler.

6.4. Dependencies

This component depends on initialisation functions of Linux kernel for PCI and IDE layers. It only makes sense to use it with the RTLinux Disk scheduler and file system component.

6.5. Status

A simple IDE device driver prototype is working with single sector operations, which has been used to study how Linux processes and RT Linux tasks could share the same IDE disk.

At the moment, the status is design phase, since the DMA functionality depends on global structures used in the file system component.

6.6. Implementation issues

The device driver must provide RT tasks with DMA support, since our design is based on bandwith DMA offers. Indeed, Linux makes a conservative DMA configuration: if the device is not known, the DMA is not activated. Obviously the successful of our design is based in DMA functionality, so a more aggressive configuration is required.

Since the way the IDE device will work with RTLinux is the same as works with Linux, we could use the Linux implementation code. However, Linux IDE device driver implementation is very related to the block layer, with includes buffer and page cache structures. Linux requests are processed inside the IDE device driver function, and when the disk processes the request, an interrupt service routine runs after the IDE disk has activated an interrupt. All of this functionality depends on Linux buffer heads, a critical component in Linux Block Layer design. This architecture was designed taking into account magnetic disks limitations and with the idea of multiple users working with disks concurrently and in a predictable way. This is not true in the RT Linux environment, so

buffer heads are not necessary, which means we will have to write this functions without the Linux buffer heads and request structures.

On the other hand, we have to avoid Linux and RTLinux accessing to IDE disk at the same time, with a reserved part for each one. This is easy if we work with two different disks, one for Linux and the other for RTLinux, but if we want to use the same disk, some interconnection between Linux and RTLinux must be established. Indeed, it would be interesting that the real time file system was accessible from Linux. To do this, Linux device driver will not communicate with the device directly. Instead, the request will be sent to the RTLinux thread that controls the IDE device physically. When the IDE disk process a request, an interrupt is raised. This interrupt is catched by RTLinux, and then a software interrupt is raised to Linux which will run the Linux original interrupt routine for IDE disks. Some changes are needed inside these Linux routines since they access to IDE registers and they make some assembler instructions, so some patches would be needed.

A very important disk functionality are DMA operations. Since the RTLinux file system and disk scheduler are designed to support efficiently the storage operations of a video stream, DMA becomes the most important point in the system. Last Ultra DMA modes support 100MB/s. Such bandwith are unreachable in PIO modes (Programmed Input Output). DMA in Linux also works with buffer heads, which implies a new implementation of DMA routines.

Initialization of IDE disks is done by Linux kernel at boot time. RTLinux only needs the structures that Linux has initialised, so no special initialization routines are necessary. If we want to have an IDE device driver in a stand-alone RTLinux system, we can copy these routines to do the job.

6.7. Validation Criteria

Since this component has not utility without the file system and disk scheduler, validation criteria would focus on how the complete block layer works. The interface with the RTLinux tasks is located in the file system, so validation criteria for IDE driver could be the validation criteria for the file system.

6.8. Tests

See Tests section of the file system and disk scheduler component.

Chapter 7. RTLinux Disk scheduler and file system (RTLfs)

7.1. Description

The aim of this component, together with the IDE Device Driver component, is to port the hard disk IDE driver to RTLinux. This implementation will consist of three distinct elements: the file system, the disk scheduler and the disk device driver. Although we implement these elements separately, to support future developments of only one of them, they are not completely independent: for example in the file system design we need to know what mechanism supports the disk device (as DMA) to take advantage of it, or the disk scheduler can need information of which are the response times of the disk device to achieve the requeriments. Obtaining real time capabilities inside the block layer requires some kind of political decisions about when a read/write request must be processed, and which request to choose when there are more than one ready. Therefore, one part of this component will be the disk scheduler, which takes the decissions about how priorities are assigned to RT-Tasks to access to the IDE disk. The other part of the component is the implemetation of real-time filesystem to RTLinux.

7.2. Layer

With this component we do not change the RT Linux core, only we add new functionalities. Therefore, it is a High-level RTLinux component.

7.3. API / Compatibility

The interface with the system is through the file system. Every read/write request from RTLinux or Linux will be sent to disk scheduler, using a well-defined function. The parameters of this function will be: type of operation, initial sector, number of sectors and buffer to data.

7.4. Dependencies

It is strongly related to the IDE device driver component.

7.5. Status

The component is in the design phase.

7.6. Implementation issues

The implementation will focus on the disk scheduler and the file system. Following are the key issues in the implementation of these two parts of the component:

Disk scheduler functionality

The disk scheduler receives requests from RT tasks or Linux kernel, to take decisions about what request must be processed first and to send data to the disk device driver using IDE API. The request structure is different from the Linux one, since in Linux this structure is strongly related to the buffer and page cache.

The scheduler will be implemented as a RT-task, which will be wake up by some other RT-tasks for read/write. In the first prototype, the disk scheduler task works in a periodic way, which is not the best option.

File system functionality

This is a list of points to study in the design and implementation of the file system:

- Concurrency: The number of tasks using the file system at the same time. RT Linux is not a general purpose operating system as Linux, where it is usual to have a lot of tasks working at the same time. In RT Linux normal case there are a low number of concurrent tasks, possibly one or two. Obviously, mechanisms to share the file system between several tasks is a must, but the number of concurrent tasks supported is important since is necessary a memory structure per task and open file. If there are a low number of open files to search inside this structure, listing will be fast. But, for a high number of them, it is better to use another data structure as AVL's which implies more complex code.
- Simplicity: the key in RT Linux is simplicity. It is not necessary to build a full real-time operating system. In this way, the RTLinux core is easier to maintain. If we don't want to break this approach, the file system design must be simple, avoiding complex implementations. We are not thinking in making a file system for all kind of requirements, only to support video streams storage.
- Space allocation: Space allocation is critical to achieve the requeriments, and the file system must offer contiguous blocks to files. Two approaches can be followed: i) chunks of 64KBytes since DMA operations has this upper limit. ii) Extents with a long number of consecutive blocks, where one file can have a lot of these extents. The decision about what to choose depends on whether it is more important to read or to write.
- API with users/tasks: A POSIX interface would facilitate the use of the file system.
- Buffers allocation, tasks or file system? : DMA operations must have contiguous memory. The problem here is where must be this memory allocated? When a read/write call is done, the task gives a pointer to a buffer which will be used into DMA operation. Must this memory space be allocated by the file system or must be owned by tasks? This leads to other important questions: if the tasks own this memory they must know how to work with it (it is not possible to rewrite memory which is being used by DMA). If the memory is owned by file system, how much space must be reserved by the file system for this purpose? Must this space allocation be dynamic? This second approach implies data will be copied two times, which can not be an efficient solution for real time systems.
- Read and write functionality: must these functions block the caller task? This is very related to who is the buffer's owner.
- Interface with the disk scheduler: how to send the request to disk scheduler? This function must support mutual exclusion.

7.7. Validation Criteria

Since this component has not utility without IDE device driver component, validation criteria will concern to both components, and it will be focused on the file system. The goal of the RTLinux block layer is to store video streams efficiently. The validation criteria will be how many concurrent video streams can be written at the same time, and how it affects reading a video stream stored previously at the same time that other video streams are written. As one of the aims was to share the same IDE device between Linux and RT Linux, another validation criteria would be what is the performance of Linux when some RT-tasks are working with the IDE device.

7.8. Tests

We will test what is the behavior of the system when one or several video strems are being written to disk. For this, we will make simulations of the different MPEG modes, and this will help us to limit the system posibilities. It will also be tested which is the system behavior when a task reads a video stream stored when, at the same time, other tasks are writting data. Finally, to see how Linux is affected whith this implementation, tests will be done with Linux doing several input/output oriented tasks at several levels (desktop user, server machine).

Chapter 8. Stand-Alone RTLinux (RTLsa)

8.1. Description

RTLinux is a executive developped to work jointly with Linux. RTLinux takes the control of the real-time critical hardware devices (timers, interrupt controller and interrupt management processor instructions) while leaving to Linux the control of the rest of the system (device drivers, memory management, etc.). RTLinux is implemented as a modification (patch) of the Linux kernel, which intercepts the low level devices, and a set of modules that provide the API. RTLinux code takes the control of the system **once Linux has booted** the system.

In what follows, the original RTLinux executive (which depends on Linux) will be called "standard RTLinux" and the RTLinux provided by this component will be called Stand-Alone RTLinux.

Stand-Alone RTLinux will be a Linux independent implementation of RTLinux. Stand-Alone RTLinux will be a bootable executive with the following main features:

- Low memory overhead. The core systems contains only following modules: the code of RTLinux, the minimum code needed to boot the system and basic virtual memory management.
- Scalability. Users will be able to customize RTLinux funcionalities to reduce kernel memory usage.
- Porting will be possible to systems without hardware for virtual memory support. Therefore, Stand-Alone RTLinux could be ported to a wider range of architectures.
- Less TLB and memory cache misses since only real-time applications are being executed.
- One problem of RTLinux is the use kernel modules or applications that execute directly interrupt management processor instructions (`cli` and `sti`), for example, some xfree86 drivers disable interrupts or lock the PCI bus for long time periods. Stand-Alone RTLinux removes this problem since the target system will be compiled to a single static and bootable kernel.
- We are not longer limited by the Linux memory manager. Standard RTLinux memory management relays completely on the Linux so it is not possible to implement custom virtual memory management algorithms. Stand-Alone RTLinux has full control of the MMU.

8.2. Layer

Low level. It will be a full reimplementation of RTLinux.

8.3. API / Compatibility

Like RTLinux, Stand-Alone RTLinux will be POSIX 1003.13 compliant.

8.4. Dependencies

In this moment, Source Code needs Linux include directory to compile like RTLinux since some Linux code (as spinlocks) are taken directly from Linux include files.

8.5. Status

Stand-Alone RTLinux users can use a wide RTLinux POSIX subset. The low level API that RTLinux rt_core provides are in alpha phase. The main features implemented are:

- □ Basic Preemtive Scheduler.
- □ Periodic timer support.
- □ IPC. (Semaphores, Conditional Variables, Mutexs, Barriers, Signals)
- □ Posix Input/Output API via the RT-Terminal component.

8.6. Implementation issues

In the x86 architecture, Stand-Alone RTLinux works in Protected Mode without page management although users will be able to enable the page managment facility turning on the Memory Protection support in the configuration menu (See SARTL-MP component). It uses a flat memory design looking for lower memory access overhead (no address translation) and also it enhances architecture portability. Linux boot code has been inserted in the RTLinux arch directory to provide system with a basic Real Mode initialization code.

`init_tasks()` function is called after the system initialization and can be used to create user threads and initializate IPC variables. This function takes the role of `init_module()` in the standard RTLinux.

8.7. Validation criteria

The basic system is expected to fit in 100Kb RAM. The core real-time kernel parameters like context switch time, interrupt response, and the execution time of applications should be improved since the processor runs with paging disabled.

8.8. Tests

Standard RTLinux provides a set of regressions tests, Stand-Alone RTLinux will be pass all theses regression tests not related with Linux (like FIFO's, shared mem, etc.).

Chapter 9. Stand-Alone RTLinux Memory Protection (RTLsaMprot)

9.1. Description

One of the most important characteristics of a real-time system is robustness and fault-tolerance. Applications has to be developed using good programing methods, but the RTOS can also improve the robustness by catching and limiting the effect of programming bugs while the system is running. General purpose operating systems use memory protection (usually page-based) to implement isolation between processes.

Memory protection is not implemented on all embedded RTOS due to two main reasons: (1) some embedded processors do not provide the necessary hardware support; (2) the overhead introduced by the protection mechanism may by not negligible. Also, when there is only one single application running in the system, memory protection is of limited utility, since there is no other application to protect from.

POSIX thread model defined that each normal (weighted) process can contain one or more threads (light-weight). Standard RTLinux executive was designed to run one single normal process, that is, a single execution space with multiple threads. In this scenario, memory protection can only by applied to code (both kernel and application) and kernel data, since all threads must share the code (exec and read) and data (read and write).

This component implements two different memory protection models:

- Kernel memory protection. It was designed to be extremely efficient and portable, and also to be compatible with the original programming model of RTLinux where all the application threads share all the data.
- Context memory protection. This second mechanism is a more complex and powerful implementation that permits to have several protected execution contexts where in each context are executed one or more threads. This second implementation provides protection for several isolated processed where each process runs several threads. It is also possible to run device drivers in one of the protected contexts.

Current general purpose OS and RTOS implements strong memory protection with a very restrictive memory settings, for example the kernel space can not be accessed (in any way) by applications. The main reason for this restrictive implementation is to protect the system sensitive data like passwords from malicious users. Most RTOS systems do not need to be protected from external or even internal malicious users since this kind of systems do not provide remote or even local user interfaces. For this reason, our component will implement memory protection to protect, or intercept, against programming faults.

9.2. Layer

This component is located at the Low level Stand-Alone RTLinux Layer. The Scheduler and the user API have been modified.

9.3. API / Compatibility

We won't need any User Level API. Memory Protection will be a user transparent fiability mechanism.

9.4. Dependencies

None.

9.5. Status

At this moment. Users have context and kernel memory protection.

9.6. Implementation issues

Different approaches has been studied and analysed based on the following man objectives:

- Low time overhead. Use the more simple and less intrusive memory mechanism.
- Low memory fragmentation. Some processors provide protection mechanisms jointly with the virtual memory mechanisms like paging or segmentation. Each of these models has pros and cons.
- Easy portability. It should be a mechanism available on most processors.

Finally, we choose paging as the MMU facility to implement memory protection.

It is important to note that RTLinux is a simple executive with no separation between RTLinux kernel and application. In fact, both the RTLinux executive and the application run in supervisor mode, in the same address space and share the same stack, that is all the code will be linked together in a single executable with no special trap or interrupt to enter into kernel code but simple function calls. Our idea was to maintain the same execution environment and avoid using separated user and supervisor modes.

For the first model, where only the system has to be protected, we will use a single page table for kernel and application, where virtual addresses are the same than physical addresses (page tables are only used as the method to protect pages, not to implement real virtual memory), therefore the same code can be executed with paging activated or deactivated. When entering kernel code, paging is disabled and when returning to the application paging is enabled.

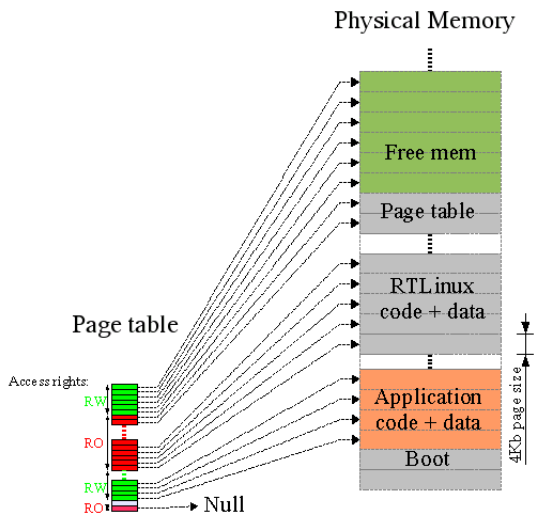


Figure 9-1. Single application memory map (first model).

The IA-32 (486+) architecture has a special feature that permits to control whether the protection bits of the pages are honoured by the processor or not without the burden (invalidating TLB) of disabling and re-enabling paging system. It is a single bit called WP in the CR0 control register. This feature was initially introduced by Intel engineers to facilitate the "copy-on-write" strategy used in most modern UNIX systems as a way to improve the creation (`fork()`) of new processes.

The second model will be implemented in a more conventional way using several page tables, one per context. In this case, context switch has to be modified, and so the overhead is somewhat higher that in the first model. Although memory protection is a user transparent facility and there are no user API available, users must be able to insert code and data in a particular context. For that, users will use prefix strings in the object file names and initialisation code will be able to detect and initialise the memory environment properly.

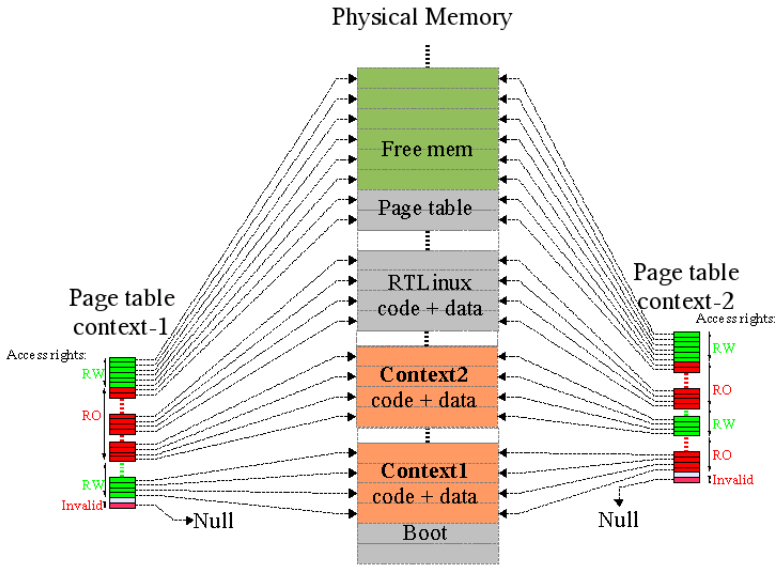


Figure 9-2. Two contexts example (second model).

9.7. Validation criteria

9.8. Tests

Several tests will be implemented in order to check the different address space accessibility. Also a set of fault injection tests will be developed.

Chapter 10. Stand-Alone RTLinux Debugging Tools (RTLsaDebug)

10.1. Description

Although testing do not demonstrate the absence of bugs, debugging and tracing are mandatory tools that must be available in any system. These tools are of especial importance in real-systems since a logical or temporal fault may have catastrophic consequences.

Standard RTLinux implements several debugging utilities: simple console print, simple trace system, POSIX trace (provided by OCERA), and GDB debugging. The GDB debugger is executed by Linux in the same system that RTLinux is running; the RTLinux GDB agent and the GDB are communicated usign RT-FIFO's.

Since the debugging tools were needed to develop SA-RTL, they were developed at the same time the Stand-Alone RTlinux were developed. The debugging tools were needed to develop SA-RTL. Debugging is done remotely from a host machine.

The set of debugging tools, besides the RT-Terminal that implements the basic console input/output interface, are:

- GDB agent. The GDB agent in the target SA-RTL is communicated with the GDB process, running on the host, via a serial line. The GDB agent and communication were implemented in a way that **permits the debugging of any kind of code**: application, kernel and event interrupt handlers (where interrupts are disabled).
- The GDB agent will be extended with facilities: measure execution time between breakpoints; generate hardware interrupts; manage processor status, like disable/enable memory cache; etc.
- Simple trace events. This is not the complex and powerful POSIX trace but a very simple and compact way to emit events from the kernel or the application.
- A script to convert the traced data to the format used by Kiwi (a graphical tool to display temporal traces) to visualise the trace.

It is important to note that the debugging tools has been intensively tested and validated during the implementation of SA-RTL.

10.2. Layer

Stand-Alone RTLinux low level layer.

10.3. API / Compatibility

GDB Agent provides all the GDB functionality plus added features to measure time elapsed between breakpoints.

Simple tracer only needs one macro:

`RTL_TRACE(TRACE_ID,TRACE_VALUE)`

Insert event information of event TRACE_ID in the tracer buffer. This buffer will be downloaded to the host machine via the serial port with the GDB interface.

10.4. Dependencies

None.

10.5. Status

At this moment, Stand-Alone RTLinux GDB agent can handle the base mandatory command subset.

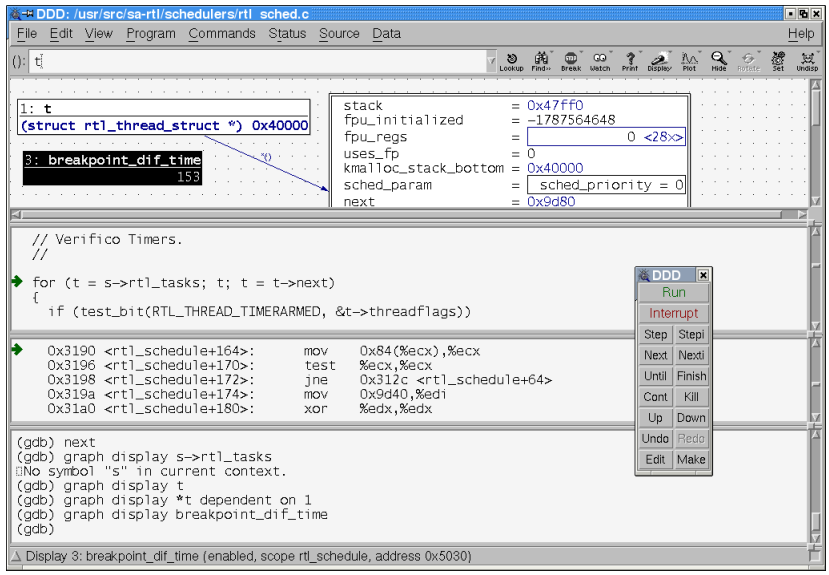


Figure 10-1. Working with the GDB agent using the DDD interface.

The simple trace is implemented and a script to convert traces to the VCD (Value Change Dump, it is a common trace file format mainly used in hardware traces).

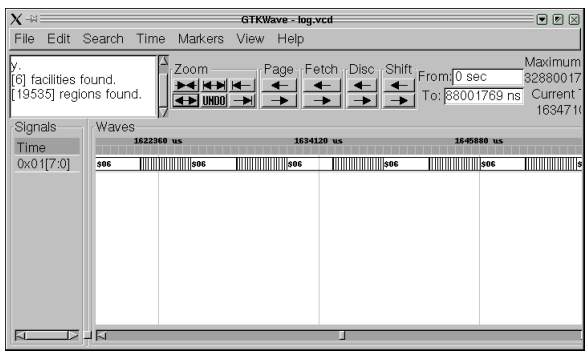


Figure 10-2. A VCD trace sample viewed with Gtkwake

10.6. Implementation issues

GDB debugger communicates with GDB Agent in the remote (target) system using the serial port. To allow GDB agent to debug Kernel code in a safe way, that is even with interrupts disabled, the serial line drives has been programmed not using interrupts its interrupt capabilities but by polling.

Breakpoints has been enhanced to store in a global variable the value of the TSC register (Time Stamp Counter), a internal 64 bits processor register which counts the number of CPU cycles since the las CPU reset. This way it is possible get a fine grain measure of the time elapsed between two consecutive breakpoints.

10.7. Validation criteria

It is new facility.

10.8. Tests

The best way to test this component is by using it. The GDB Agent and tracer was successfully used during the Stand-Alone RTLinux development.

Chapter 11. Porting RTLinuxSA to ARM processor (RTLsaARM)

11.1. Description

This component will provide the RTLinux Stand-alone (originally designed to work on x86 machines) for the Strong ARM processor. The board used to develop the porting will be CerfCube 1110 developed by Intrinsyc.

Currently imec (http://www.imec.be/rtlinux/) ported RTLinux V3.0 to the Strong ARM processor. This porting includes all the Linux kernel. Since the ARM processor family is mainly used in the embedded market, it is interesting to provide an RTOS with a small footprint.

This component will port the basic core system mechanisms, that is, no memory protection only the system facilities like threads, and IPC.

11.2. Layer

Below low level. It is a core hardware management code porting, it will not modify internal facilities, like the scheduler, but will replace hardware specific code.

11.3. API / Compatibility

Like RTLinux, Stand-Alone RTLinux will be POSIX 1003.13 compliant. It will be transparent to the user.

11.4. Dependencies

It will depend on the RTLinux Stand-Alone component.

11.5. Status

Early development stage. Only a simple "hello world" booting program has been done.

11.6. Implementation issues

The RTLinux parts that has to be ported are: interrupt management, hardware clock management, and simple memory initialisation.

11.7. Validation criteria

It is a new porting.

11.8. Tests

Standard RTLinux provides a set of regressions tests, Stand-Alone RTLinux will be pass all theses regression tests not related with Linux (like FIFO's, shared mem, etc.).

Chapter 12. RTLinux UDP/IP (RTLUDP)

12.1. Descritpion

Currently, standard RTLinux/GPL has two different IP stacks: RTSOCK and LwIP (LwIP was ported to RTLinux by a college also working in the UPVLC, in a project sopported by the national reseach department, MCYT).

Rtsock is not a device driver for network cards. Instead, packets flow through the Linux kernel using the standard Linux drivers, up/down the standard layer 2 and layer 3 protocols, and then packets are diverted into an RTLinux task. Currently only UDP sockets are supported.

LwIP is a complete and full featured TCP/IP stack designed to be ported to embedded systems easily. The current version of RTLinux provides support of a small number of network card drivers.

This component is an **implementation of the UDP/IP stack from scratch** (not derived from BSD code as many other TCP/UDP/IP implementations). The main design criteria is efficiency, while features and compatibility are secondary. The stack will be prepared (interfaced) to work with the network device drivers developed in the project **EtherBoot**, which has a large base of supported drivers; and also be connected with the Linux networking stack (via a virtual network driver), providing a high level communication mechanism between RTLinux and Linux on the same machine.

Next figure outlines the internal structure of the proposed component: the core of the component is the UDP/IP stack, which provides the socket interface to RTLinux threads; the component will implements the required API to use all the Etherboot project network device drivers; and finally, a standard Linux network driver will be provided so that Linux user application can communicate via UDP/IP with RTLinux threads.

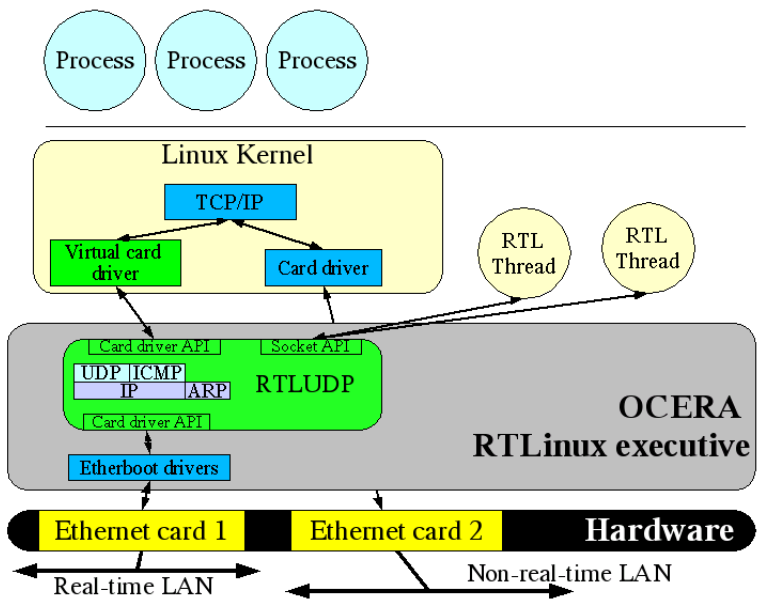


Figure 12-1. RTLUDP component

12.2. Layer

Low Level, it also interacts with the Linux Kernel since it allows to send ARP/IP packets via the kernel TCP/IP stack.

12.3. API / Compatibility

Standard BSD socket API: `send()` and `sendto()` for sending messages; `recv()` and `receivefrom()` to receive messages; and `connect()` to establish the connection.
The Stack will be compatible with all the Etherboot network drivers.

12.4. Dependencies

RTLUDP do not have dependencies with external components.

12.5. Status

Alpha status

12.6. Implementation issues

RTLUDP will implement a light UDP stack with the following functionality:

- ARP protocol: Complete.
- IP protocol: Complete. But packet split and recombine are not supported.
- UDP protocol: Complete.
- ICMP protocol: Only some parts will be implemented like ECHO, DESTINATION UNREACHABLE, etc. Packages related with host or router information and management will not be implemented.
- TCP protocol: TCP protocol will not be implemeted since it is not a real-time protocol.

Although the first version of RTLUDP stack will be implemented using a one copy method, we will study the way to provide a real zero copy network (may be using the STREAMS interface).

12.7. Validation criteria

The only validation criterias are:

- A Real Time behaviour, when the real time protocol is used, must be achieved, this will be the main goal of the RTLUDP.
- RTLUDP stack must be compatible with UDP/IP protocol.

12.8. Tests

There are several test that can be implemented, but the most important is to check the REAL TIME behaviour of the RTLUDP stack. All the others features of the stack can be checked with these tests. By these reasons, all the test must be oriented to real time applications. Several samples can be:

- Tests to check ARP protocol: The firsts tests must be oriented to check the ARP protocol since this protocol is the base of the IP communications on a ethernet-based LAN.
- Tests to check ICMP protocol: ICMP protocol is a non important feature of RTLUDP but it must be checked too. These tests should check if when an error is provocated, a correct ICMP is sended.
- Tests to check UDP protocol: UDP protocol will be the protocol used by RTLUDP, by this reason it must be completly probed, the tests could be, applications sending data, applications receiving video data, etc...

Bibliography

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

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

[PTS] *Posix Test Suite*.

[BPA] *Bigphysarea*.

[Lepreau02] L. Lepreau and M. Flatt, University of Utah, 2002, *The Oskit Project*.

[Gonzalez] Michael González-Harbour, Mario Aldea-Rivas, J.J. Gutierrez, and J.C. Palencia, 1998, Lecture Notes in Computer Science. Springer Verlag., 1411, 91-101, *Implementing and Using Execution Time Clocks In Ada Hard Real-Time Applications*.

[Gallmeister] Bill O. Gallmeister, 1995, *Programming for the Real World -POSIX.4*.

[OpenGroup] The Open Group, 1997, *The Single UNIX® Specification*.

[Unix] Kay A. Robbins and Steven Robbins, Prentice Hall, *Unix Programacion Practica*.

[TestSuite] *POSIX 1003.1d Test Suite*.