



LUND INSTITUTE
OF TECHNOLOGY
Lund University



Master of Science Thesis

Real-Time Linux in an Embedded Environment

A Port and Evaluation of RTAI
on the CRIS Architecture

Martin P. Andersson
Jens-Henrik Lindskov

Submitted in Partial Fulfillment of
the Requirements for the Degree of
Master of Science in Computer Science
and Engineering.

Lund Institute of Technology
Lund University, Sweden

January 30, 2003

Abstract

Real-time, embedded systems and Linux are commonly used words in these days. This thesis looks deeper into the possibility of turning Linux into a real-time operating system. Particularly, it investigates available hard real-time solutions for Linux, but also looks into the soft variants for completeness. RTAI is selected as a suitable solution for Axis and is ported to ETRAX, the in-house developed CPU designed with networking and embedded systems in mind. A large number of performance tests are conducted during the evaluation to make sure that the implementation meets the demands of a real-time operating system. The evaluation shows that RTAI provides good real-time performance, especially when compared to standard Linux.

Preface

This master thesis project has been conducted at Axis Communications in Lund, Sweden, for the department of Computer Science at Lund Institute of Technology. The project was carried out over a period of 20 weeks.

We wish to thank our advisors at Axis Communications, Lars Viklund and Hendrik Ruijter and also Anders Nilsson, our advisor at the department of Computer Science.

The authors can be contacted by e-mail at `martin.andersson@linux.nu` and `jens-henrik@linux.nu`.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Description	1
1.3	Problem Analysis	1
1.4	Report Outline	2
2	Investigation	5
2.1	Real-Time Systems	5
2.2	Linux and Real-Time	6
2.3	Motivation for Axis	8
2.4	Evaluation	9
2.4.1	RTLinux	9
2.4.2	RTAI	11
2.5	Conclusion	16
3	Implementation	17
3.1	Hardware Abstraction Layer	17
3.1.1	Interrupt Handling on Linux/CRIS	18
3.1.2	Interrupt Handling on Linux/CRIS with RTAI	21
3.2	RTAI Kernel Module	24
3.3	Implementation Choices	25
3.3.1	Timers and Cycle Counter	25
3.3.2	Unaffected Interrupts	26
3.4	Limitations	26
3.5	Known Problems	27
4	Evaluation	29
4.1	Definitions and Measurement Approaches	29
4.1.1	Interrupt Latency	29
4.1.2	Interrupt Task Latency	30
4.1.3	Scheduling Latency	32
4.1.4	Scheduling Precision	32
4.1.5	Communication Overhead	33

4.2	Results and Discussion	34
4.2.1	Interrupt Latency	34
4.2.2	Interrupt Task Latency	39
4.2.3	Scheduling Latency	39
4.2.4	Scheduling Precision	40
4.2.5	Communication Overhead	43
5	Final Improvements	45
5.1	Test results	46
5.1.1	Interrupt Latency	46
5.1.2	Scheduling Precision	47
6	Conclusion	49
6.1	Discussion	49
6.2	Summary	50
	Bibliography	54
A	Test Environment	55
B	Test Programs	57
B.1	Interrupt Latency	57
B.1.1	RT-Handler	57
B.1.2	Linux-Handler	58
B.2	Interrupt Task Latency	59
B.2.1	RT-Task	59
B.2.2	Kernel-Thread	61
B.3	Scheduling Latency	62
B.4	Scheduling Precision	64
B.5	Communication Overhead	66
B.5.1	Between RT-Tasks	66
B.5.2	RT-Task to User-Space	67
B.5.3	User-Space to RT-Task	68
B.6	Load Program	69
B.7	Interrupt generator	69
C	Modified Kernel Files	73
D	Kernel Modules	77
E	Glossary	79

Chapter 1

Introduction

1.1 Motivation

Axis Communications and many other device manufacturers are turning to Linux for embedded systems. Using a full-featured UNIX-like operating system obviously has many advantages. However, the Linux kernel can not support hard real-time processing, which may be required by certain embedded applications. There exists a number of extensions to Linux that provide support for hard real-time tasks.

Axis uses the ETRAX processors in most of its products and has previously ported Linux and some real-time operating systems to ETRAX. However, Axis and its customers have an interest in being able to combine Linux with hard real-time, as this combination would provide both a full-featured free UNIX-like system with many available applications, hardware drivers, *etc.* and hard real-time capabilities.

1.2 Problem Description

The purpose of this thesis is to investigate available hard real-time extensions to the Linux kernel, select a suitable one for Axis, port it to the Axis ETRAX architecture and evaluate its real-time performance.

1.3 Problem Analysis

The investigation covers how real-time performance can be achieved in Linux and the availability of hard real-time extensions. It describes how the extensions work technically, the ease of porting, available documentation and licensing rules. Axis' need for real-time is examined in order to select a suit-

able extension.

The major challenge in porting the selected extension is to understand how the Linux kernel works on ETRAX in order to safely implement the architecture specific parts of the extension. It is important to keep the standard kernel intact as far as possible.

The goals of the evaluation are to verify the functionality of the system as well as measure its real-time performance on the ETRAX platform.

1.4 Report Outline

Chapter 2 - Investigation This chapter presents background theory and definitions. It also describes in which ways real-time can be achieved in Linux. An investigation of real-time extensions is made and finally a suitable extension is selected for porting.

Chapter 3 - Implementation The design of the selected extension and especially how it modifies the Linux kernel, is explained in this chapter. Our implementation choices and difficulties are also described.

Chapter 4 - Evaluation Defines the tests used to evaluate the functionality and performance of the system. The results are presented and discussed.

Chapter 5 - Conclusion This chapter contains our conclusions based on the evaluation. A discussion with respect to our personal reflections and suggestions of future enhancements is also presented.

Appendix A - Test Environment A detailed description of the test environment is presented here. That includes the hardware used, software versions, network configuration *etc.* It is possible to repeat the tests using this information and Appendix B.

Appendix B - Test Programs In this appendix the source code of the test programs is presented.

Appendix C - Modified Kernel Files Shows a list of files modified in the Linux kernel by the selected extension. Also, a brief description of the modifications is provided.

Appendix D - Kernel Modules Describes essential parts of the selected extension together with file sizes and memory usage.

Appendix E - Glossary Some acronyms and common concepts are explained shortly.

Chapter 2

Investigation

The purpose of this chapter is to explain in short the concept of real-time and what can be expected from a real-time operating system. The different approaches towards providing real-time in Linux are discussed. Then a comparison of two hard real-time extensions to Linux is made and finally a selection of one of them is motivated, based on the requirements of Axis. The selected extension is ported and evaluated during the following parts of the thesis.

2.1 Real-Time Systems

A real-time system is a system in which the correctness of the system depends not only on the logical results that the system produces, but also on the time at which the results are produced [1]. This is a formal definition of a real-time system. Before proceeding it is appropriate to define and explain some other related concepts.

- The *response time* of an application is the time interval from when the application receives a stimulus, usually provided via a hardware interrupt, to when the application has produced a result based on that stimulus [2].
- The *deadline* of a certain task in an application is the longest acceptable response time for the task.

For example, say we have a robot arm picking up components from a conveyor belt. An optical sensor informs the robot when a component is approaching the arm. After the robot has received the information there is a certain small amount of time available for the robot to react and for the arm to move down to the right position over the conveyor belt. This time is the deadline for the “move-arm”-task of the robot application. If the arm is not there in time, *i.e.*

if the deadline is missed, the component may be lost. If the arm reaches the position in time we say that the deadline is met.

A *hard* real-time system is a system in which all the deadlines of the system must be met at all times. It is the system designer's responsibility to make sure that the deadlines *can* be met, that is the system must not be overloaded. A *soft* real-time system, on the other hand, is a system in which the deadlines usually must be met, but it may be acceptable if a small number of deadlines occasionally are missed [2]. An example of a hard real-time system is an air-traffic controller, here it is critical that every deadline is met. An example of a soft real-time system is an audio sampling application where it may be acceptable if some samples are lost from time to time, as long as it does not happen too often.

2.2 Linux and Real-Time

A real-time operating system (RTOS) can be described as a system that meets timing requirements of the processes under its control. Linux is not designed to provide real-time performance. It provides a good average performance for applications. For a real-time application with relatively long deadlines it may be sufficient if the environment can be controlled properly (fixed number of processes, well-tested drivers *etc.*).

However, for applications that require very low response times, or hard real-time, the standard Linux kernel is not sufficient. It is clear that in order to have hard real-time, guaranties must exist that no deadline is missed. This type of guaranties require a deterministic environment. For an operating system this means that it must be possible to predict the maximum time it takes to perform different tasks, such as interrupt handling and scheduling. Also, the kernel must be *preemptible*, that is if a lower priority process is running a system call in the kernel, it must be possible to interrupt it if a higher priority process is ready to run. This is currently not the case with Linux (2.4).

Improving the Kernel

There are two different approaches towards providing real-time in Linux. In the first one, the standard Linux kernel is improved, either by attempting to make the kernel preemptible (by altering the kernel in different clever ways, see [3]) or by adding preemption points to the code (*i.e.* checking more often if a higher priority process is ready to run). This results in a kernel more responsive to applications without any need for alterations in

these applications [3]. This approach is sometimes used together with a new improved scheduler implementation. MontaVista [6] and TimeSys [7] provide preemptible kernels while REDSonic [5] has preemption points [3]. It should be noted that in the coming releases of the Linux kernel, the preemption patches originally supplied by MontaVista (now maintained by Robert Love¹) are included by default and the functionality is available as a configure option. Thus, the Linux kernel is expected to provide better real-time performance in the future.

Adding a Real-Time Kernel

Another approach is to make the Linux kernel fully preemptible by adding a hardware abstraction layer “between” the system hardware and Linux. Also a new separate real-time scheduler is used which runs Linux as its lowest priority thread. The abstraction layer takes control over the system interrupts and passes them on to Linux only if no real-time task is running. When Linux tries to disable interrupts it only sets a flag in the abstraction layer and cannot really turn off the interrupts. Thus, the real-time scheduler has full control over the system and Linux runs virtually unmodified. A small real-time kernel has been added to the system. The real-time tasks are written as kernel modules and executed within kernel-space. They have access to a special real-time API. There are two projects providing this technique, RTLinux [13] and RTAI [8]. RTLinux is the oldest project of the two, in fact, RTAI is based on the ideas behind RTLinux.

Discussion

The two approaches both have their pros and cons. In the preemption improvement approach the major disadvantage is the lack of guaranties it can provide. Unless every possible code path in the kernel is examined, it is impossible to provide a guarantee about the latency [4]. It is clear that analyzing all possible paths is very hard, and even if it was possible under some restrictions, the development of the kernel and addition of new drivers *etc.* would make the analysis hard to maintain. Also new code would have to meet the requirement not to introduce additional long non-preemptible kernel code paths. The motivation for the preemption improvement approach is the fact that it improves the Linux kernel without the users having to modify their applications.

In the approach based on a hardware abstraction layer the application is preferably split up into two parts:

¹Patches and information available at <http://www.tech9.net/rml/linux/>

- The part with timing requirements (e.g. data sampling). It executes as a real-time task in kernel-space.
- The other non time-critical part (e.g. user interface) executing in user-space.

The different parts communicate with each other using for example FIFO-queues or shared memory. The split often requires a new design of the application. Also, the real-time tasks are written as kernel modules using the special real-time API, not the standard Linux API. Writing Linux kernel modules requires different programming skills than it does writing a Linux application process [4]. However this is a small price to pay, since this approach can provide the deterministic environment required for hard real-time. This is possible mainly because the API-functions available to the real-time tasks are well tested and analyzed, as are the internal functions of the small real-time kernel. The amount of code which must be analyzed in this approach is relatively small and contained, compared to the entire Linux kernel. Another advantage with this approach is that Linux runs virtually unmodified. Thus, it is easy to maintain this solution when new kernels and drivers are released.

To summarize, in the preemption improvement approach, all applications can without modification benefit from an improved Linux kernel with low latency; soft real-time can be obtained. But only the hardware abstraction approach can provide the guarantees required to support hard real-time, with the extra programming effort as the only major downside.

2.3 Motivation for Axis

As a developer of network cameras, video servers, print servers *etc.*, Axis and its customers would benefit from guaranteed response times in their systems. Possible applications include:

- Sampling of data under time restrictions.
- Surveillance cameras that must respond quickly.
- Wireless baseband in software, for example Bluetooth implementations.
- Control loops.

As concluded in the previous section, guaranties require hard real-time and therefore the only two alternatives available are RTAI and RTLinux. It is also these extensions that provide the best response times. In the following sections, RTAI and RTLinux will be further examined.

2.4 Evaluation

2.4.1 RTLinux

Background

Real-Time Linux (RTLinux) began as a research project in 1995 at the Dept. of Computer Science at New Mexico Institute of Technology [13]. The immediate goal was to develop a Linux kernel that would support real-time control of scientific instruments [14]. The work by Victor Yodaiken and Michael Barabanov resulted in a small real-time executive running Linux as a completely preemptible task.

RTLinux was quickly adopted as the real-time processing software of choice in a variety of production projects. In 1998, the company called Finite State Machine Labs (or FSMLabs) was formed around RTLinux. FSMLabs now has a number of strategic partners, one of them is Red Hat, Inc., which has selected RTLinux as its standard approach to hard real-time Linux application requirements.

Design

The design goals of RTLinux are stated in the (incomplete) design whitepaper [17]. Some of them are: reliability, predictability, performance, transparency and modularity. Especially the two latter are worth mentioning. Transparency means there should be no surprises in black boxes or hidden components in the system. Modularity means that applications, which do not need some system features, should be able to remove these features. All of this applies to the basic idea that the real-time operating system should be small and as simple as possible.

The need to meet user requirements for development tools, graphical user interface and networking support in the real-time operating system, as well as the pure real-time support, is satisfied by letting standard Linux run on top of the small real-time system. This design originated from the understanding that it is not feasible to identify and eliminate all aspects of kernel operation that lead to unpredictability (such as standard Linux scheduling, device drivers and uninterruptible system calls). Instead the idea was to construct a small predictable kernel separate from the Linux kernel, and to make it simple enough that operations could be measured and shown to have predictable execution [16].

Figure 2.1 shows an RTLinux system. As shown in the figure, a hardware abstraction layer is added between the system hardware and the standard

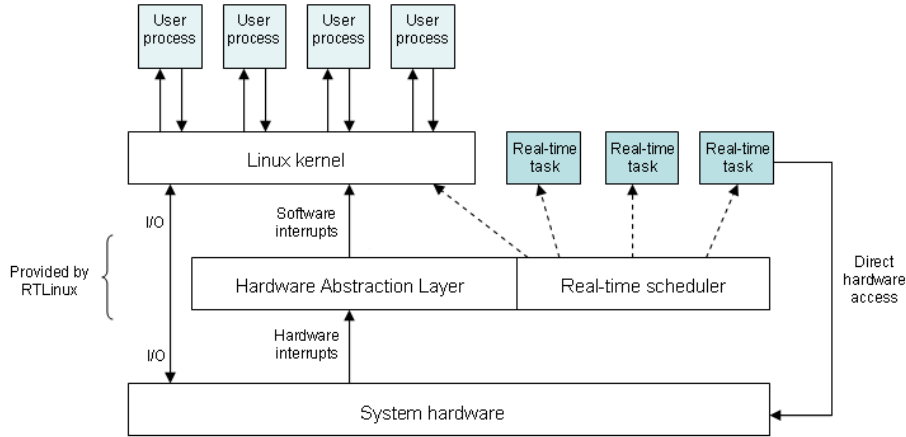


Figure 2.1: An RTLinux system

Linux kernel. A preemptive fixed-priority scheduler handles the real-time tasks as well as the Linux kernel, which runs as the lowest priority task. The abstraction layer intercepts all hardware interrupts from the underlying system. If an interrupt is not related to any real-time task, it is passed on to Linux, but only when no real-time task is running.

The RTLinux executive is in itself non-preemptible. Unpredictable delays within the RTLinux executive are eliminated by its small size and limited operations [16].

A real-time task is written as a kernel module and when loaded it informs the real-time scheduler about its deadline, period and release-time. A real-time task should not make Linux system calls as that would infer unpredictability. Also, it should be made as simple as possible and leave the non real-time parts of the code to the user-space application.

Features

Some of the key features of RTLinux are:

- Interprocess communication using FIFO-queues or shared memory.
- The API is compatible with POSIX 1003.13 “minimal real-time operating system”.
- Synchronization is achieved through POSIX thread mutex variables, POSIX conditional variables or POSIX semaphores.
- High precision timing (the programmable timer chip available on most platforms is used [15]).

Documentation and Licensing

The RTLinux API is well documented, especially since a big part of it is POSIX. Installation instructions, getting started documentation and a FAQ are available as well as a mailing list.

FSMLabs holds a patent on the basic process for real-time in RTLinux. The patent describes the technique of using a software emulation of the interrupt control hardware to prevent the non-real-time OS from causing delays in real-time operations [13]. It is a software patent only valid in the U.S. [19].

Currently, RTLinux is available in different forms from FSMLabs. “RTLinux/Pro” and “RTLinux/BSD” (which have the same code-base) are commercial distributions shipped with a development kit. Also, the remains of the open-source project that RTLinux once was, are available in “OpenRTLinux”. OpenRTLinux is released under GPL² and OpenRTLinux Patent License [18]. Any modification of the code covered by the Open RTLinux License must be released under GPL and it must be open and available on the web [19].

Summary

RTLinux is a widely used and tested solution. It has been around in some form since 1995 and is now used in many applications around the world. The technology is solid, and the use of POSIX means that an application programmer hopefully does not have to learn yet another API.

2.4.2 RTAI

Background

Development of the DIAPM³-RTLinux variant started immediately after RTLinux was released because the people at DIAPM were not satisfied with the performance offered by the first version. They had been using a self made DOS based real-time variant earlier and thought they could take some of the techniques developed there and put them into RTLinux. They maintained the 2.0.xx kernel-patch from RTLinux and the RTLinux scheduler base also remained mostly untouched, but a set of new features were added. DIAPM modified all that was related to the real-time timing, such as introducing periodic timing. They greatly improved the efficiency of the one-shot timing by using the CPU TSC⁴ instead of using the timer circuit as RTLinux originally did.

²GNU General Public License, see <http://www.gnu.org/copyleft/gpl.html>

³DIAPM = Dipartimento di Ingegneria Aerospaziale — Politecnico di Milano

⁴Time Stamp Clock

In the beginning of 1999, the 2.2.xx kernel was available and its hardware interaction was better organized with far less interrupt disabling/enabling. This made it possible to patch the kernel in the way preferred by DIAPM and in April the first version under the acronym RTAI (Real-Time Application Interface) was released by Paolo Mantegazza.

RTAI is now an open source project with an active development community. The only supported architecture in the first release was the x86, but now a wide variety of architectures are supported:

- x86
- PowerPC
- ARM
- MIPS

Design

The design of RTAI is clean. It consists of an abstraction layer called Real-Time Hardware Abstraction Layer (RTHAL) and a small real-time kernel that runs Linux as its idle task. Both the real-time kernel and the project itself are named RTAI. A sign of the modularity of this design is that one must not even use the supplied RTAI kernel but can in fact use any real-time kernel that interacts with the HAL interface. For a figure describing the design of RTAI, we refer to Figure 2.1, since the concepts are almost identical.

The RTAI project is (apart from the small patch that installs the HAL in the kernel) entirely built upon the use of modules. This makes it easy to maintain and extend the system at runtime.

Three primary functions are performed by the RTHAL [10]:

- Gathers pointers to the required internal data and to functions mainly related to hardware into a single structure, `rthal`. The functions can be dynamically switched to appropriate software emulation functions by RTAI when hard real-time is needed.
- Makes available the substitutes of the above selected functions and sets `rthal` pointers to point to them.
- Substitutes the affected function calls with calls through the `rthal-struct`.

The kernel patch changes very few lines of code in the standard Linux kernel. As of the effect this can have on Linux, we can not say it better than the RTAI people themselves:

At this point, it should be noted that Linux is almost unaffected by RTHAL, except for a slight (and negligible) loss of performance due to calling of cli and sti related functions in place of their corresponding hardware function calls, and due to the use of function pointers instead of directly linked functions.⁵ [10]

From the moment the main RTAI module is mounted and initialized, Linux is no longer in control of the hardware interrupts and therefore imposes no threat to the hard real-time tasks.

Real-time tasks are created as modules, which execute in kernel-space. This has some advantages:

- Tasks can not be swapped-out and the number of TLB⁶ misses are reduced.
- Tasks are executed in processor supervisor mode and have full access to the underlying hardware.
- The RTOS and the real-time task share execution space and the system call mechanism is therefore implemented as simple function calls instead of slower software interrupts.

Features

RTAI has a lot of features and some of them are:

- FIFO-queues
- Mailboxes
- Intertask Messaging
- Extended Intertask Messaging
- RPC⁷
- NET_RPC, which is a support for making RTAI a distributed OS (functions can be called remotely and operate on remote objects)

⁵`cli` and `sti` are used in Linux to disable and enable interrupts respectively.

⁶Translation Look-aside Buffer, see Appendix E

⁷Remote Procedure Call

- Shared memory
- RTAI semaphores
- RTAI pthreads, which implements Posix 1003.1c API (including mutex and conditional variables)
- RTAI pqueues, which implements the message queues section of the Posix 1003.1d API
- Rate Monotonic Scheduling (RMS)
- Earliest Deadline First (EDF) scheduling

Miscellaneous

- Dynamic memory (`rt_malloc`, `rt_free`) with good real-time behavior, but not hard real-time.
- Software watchdog module
- RTAI maintains compatibility with the V1 RTLinux API.
- kgdb: Source-level debugging from a host linked by a serial line.
- Linux Trace Toolkit (LTT). It is a full-featured tracing system for the Linux kernel. It includes both the kernel components required for tracing and the user-level tools required to view the traces.
- `/proc`-interface, showing for example the IRQs used by RTAI, information about the tasks in the scheduler and various other information depending on which modules are loaded.

User-space real-time

- LXRT is an API for RTAI which makes it possible to develop both hard and soft real-time applications entirely in user-space, without having to create kernel modules. This is useful because programming errors will not crash the entire system, and one can also use standard debuggers. We will not delve deeper into the inner workings of this technique but we note that it is well tested and used. As can be expected, the performance is not quite as good as in kernel space [20]. So far LXRT is only implemented on the x86 platform.

Documentation and Licensing

The RTAI documentation is well written, although not quite up-to-date. The mailing list has a lot of traffic and questions get answered quickly.

RTAI is an open-source project. It was earlier released under LGPL⁸ 2, but the core has recently changed to GPL 2, while the rest remains LGPL 2. The parts of RTAI released under GPL are the parts that potentially may be claimed to implement the teachings of the RTLinux patent. With this move from LGPL to GPL there should be no problem with the Version 2 of the Open RTLinux Patent License [18], which says: "The Patented Process may be used, without any payment of a royalty, with two types of software. The first type is software that operates under the terms of a GPL...". Eben Moglen, general counsel of the Free Software Foundation appears to support this position [12].

Summary

RTAI was derived from an early version of RTLinux, but has since followed its own track and evolved into a mature and feature-rich environment which is fully devoted to open source software. It is under active development in an open community, continuously contributing to the development process. RTAI seems to be mature enough to use with most applications and a lot of work is devoted to further improve it.

⁸GNU Lesser General Public License, see <http://www.gnu.org/copyleft/lesser.html>

2.5 Conclusion

This section presents our conclusions based on the investigation.

RTLinux and RTAI are based on the same idea and they work very much alike. The performance provided is essentially the same. There are however some more or less important differences outlined in Table 2.1.

<i>RTAI</i>	<i>RTLinux</i>
RTAI uses a small kernel patch.	RTLinux uses a large kernel patch.
RTAI is an open-source initiative and is likely to continue that way.	RTLinux started as open-source, but is commercial today. Any development seems to be in the commercial versions.
RTAI is actively developed.	RTLinux development seems to have stalled in the free versions.
RTAI has its own API, but has a POSIX module that supports some POSIX calls.	RTLinux API is fully compatible with POSIX 1003.13 “minimal real-time operating system”
RTAI has many features.	RTLinux has a minimalistic approach.

Table 2.1: The essential differences between RTAI and RTLinux

Using a small kernel patch, as RTAI does, makes it easy to maintain the system between different kernel versions. As to features it is not certain that more are necessarily better; it could be convenient with many features but it also makes the system more complex and harder to understand. The real-time part of an application should be made as simple as possible and leave the rest to the non real-time part.

It is mostly an advantage to use a standardized API such as POSIX because it may be easier for newcomers to migrate to the system. RTLinux has full support for POSIX 1003.13 while RTAI provide only some POSIX compatibility. However, the RTAI API is not difficult to understand.

We select *RTAI* as the extension of choice for the rest of this thesis. This is mainly because of its

- commitment to open-source,
- active development
- and small modifications of the Linux kernel.

Chapter 3

Implementation

As a part of this thesis, RTAI has been ported to CRIS¹ and this chapter explains the basics of the inner workings of the port. As always in these cases, the code itself is the most detailed documentation, but it could be helpful to read this chapter to get an overview before trying to read it. This chapter also gives an overview of the modifications made to the normal Linux kernel, especially how the interrupt paths have been modified in order to make hard real-time possible. These paths are essential and are therefore described both before and after the modifications made by RTAI.

RTAI consists of a patch that installs the necessary hooks into the kernel and a set of modules. These modules are primarily the `rtai.o` module which is architecture specific and others which are generic and add enhanced functionality such as scheduling and message handling.

3.1 Hardware Abstraction Layer

The kernel patch installs a hardware abstraction layer, called RTHAL, available as a kernel configure option. The layer acts as an interface between Linux and the hardware. All function calls related to interrupts are gathered in a struct named `rthal` (see Appendix C). For example, the `cli()`-macro, which is used by Linux to disable interrupts, now goes through the struct instead. When RTAI is not mounted the macro will work as before, disabling interrupts in the hardware, but when RTAI is mounted, only a flag will be set, which ensures that Linux does not receive any interrupts while real-time tasks and handlers do. In this way Linux, although not aware of it, is not allowed to perform actions that could threaten the real-time behavior of the system.

¹Code Reduced Instruction Set, the CPU architecture designed by Axis and used in its ETRAX processors.

Great care has been taken when patching the kernel as to not alter the default behavior. A small performance loss in RTAI can be traced to this approach. On the other hand, even when the hardware abstraction layer is configured in the kernel, the behavior is unchanged as long as RTAI is not mounted.

For a complete list of files that are modified by the kernel patch, please refer to Appendix C.

3.1.1 Interrupt Handling on Linux/CRIS

The major responsibility of the hardware abstraction layer is to catch interrupts from the hardware and if appropriate send them to Linux or to real-time handlers (such as scheduler functions). If Linux has disabled interrupts in the abstraction layer, it should not receive any interrupts until it re-enables them later.

This described mechanism of interrupt handling requires very careful modifications of the inner workings of the kernel. It is vital to understand how the interrupt handling is normally performed to be able to safely make the modifications. This section presents the results of a technical investigation, which was essentially made by reading code and trying to understand the functionality.

Interrupt Paths

In Figure 3.1 a schematic view of the interrupt paths in the Linux kernel for CRIS is shown. In most cases an interrupt will cause the processor to start executing at one of the `IRQxx_interrupt`-routines². The routine will start by saving registers and disable interrupts (as they are not disabled automatically). Then the interrupt is masked. If it had not been masked and interrupts were re-enabled before the specific interrupt had been acknowledged, an infinite loop would have been triggered.

The C-routine `do_IRQ` will be called with the interrupt number as a parameter. It will execute an interrupt handler, which should also acknowledge the interrupt. Normally interrupts are kept disabled during execution of the handler, but depending on how the handler was installed it may run with interrupts enabled. This approach is possible since the interrupt was masked previously. When the handler has finished, the `IRQxx_interrupt`-routine

²These routines are entered in the interrupt vector whenever a handler for the interrupt “xx” has been installed, e.g. by a hardware driver.

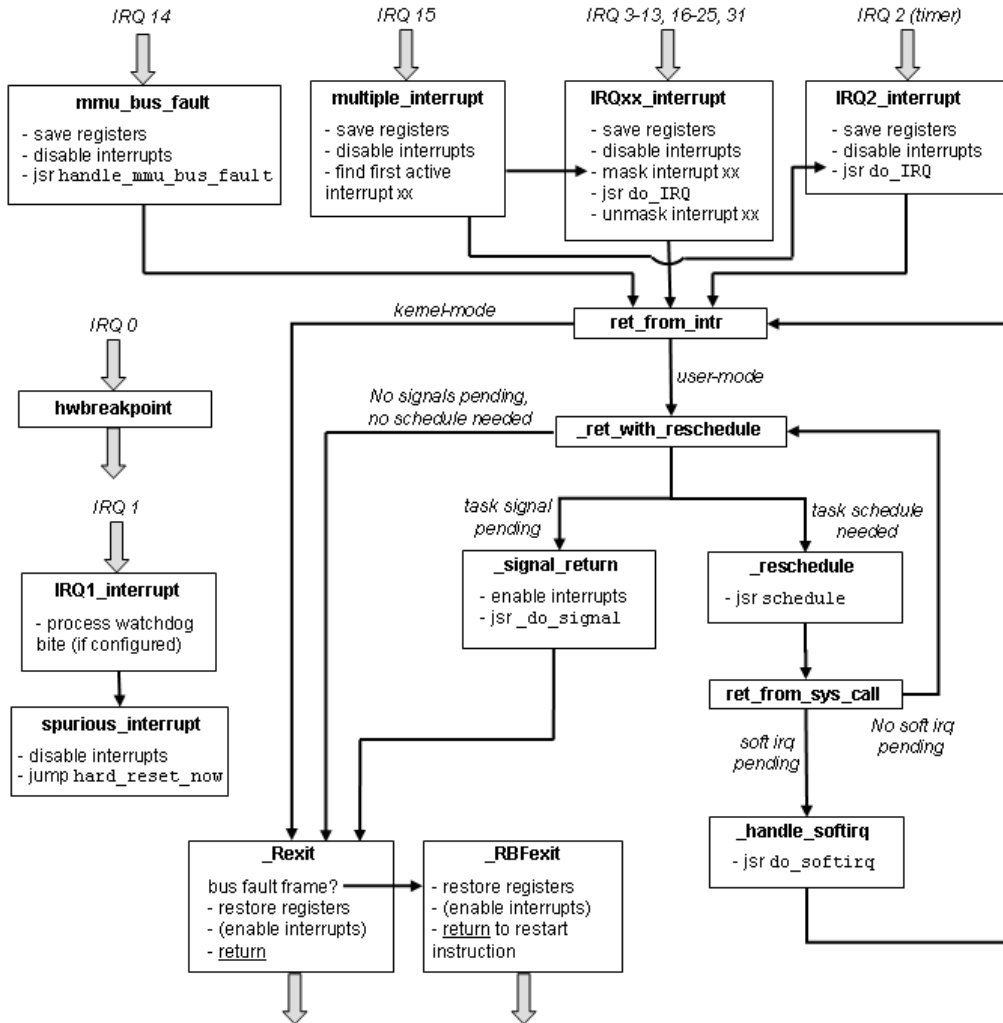


Figure 3.1: Interrupt handling paths in the Linux kernel on CRIS

continues and unmask the interrupt.

The path just described is the most important one to understand, but it is also important to see what happens afterwards. Some routines are worth an explanation.

- `ret_from_intr` checks whether the processor was in user or kernel (supervisor) mode when the interrupt occurred. User mode means e.g. that it was executing an application in user-space and kernel mode that it was running some kernel code, such as a driver. In user-mode a somewhat longer path is taken before returning.
- `_ret_with_reschedule` checks whether the current process has any signals pending or a schedule is needed. If so the appropriate routines are called.
- `_Rexit` is responsible for restoring registers previously saved and return to the code executed by the processor when the interrupt occurred. All interrupt handling ends with the return through this routine.

Special Cases

- IRQ 0: Hardware breakpoint (`hwbreakpoint`) is used only for debug.
- IRQ 1: Interrupt from the watchdog. If the watchdog is enabled in the kernel config the `IRQ1_interrupt`-routine will print out debug information and later the chip will be reset. The routine will also be called when resetting the chip from software as the watchdog is used to do this.
- IRQ 2: For timer interrupts, a special routine called `IRQ2_interrupt` is used. It works like the other `IRQxx_interrupt`-routines except that it does not mask (block) the irq. This is because `do_IRQ` also runs any pending soft-interrupts (by calling `do_softirq`) after the actual interrupt handler. `do_softirq` enables interrupts during execution of handlers and during this time a new timer interrupt could be processed but only if it is not masked. The timer interrupt is crucial and should always be processed as soon as possible since the watchdog must be reset and the system clock updated.
- IRQ 14: The MMU bus fault occurs for example when an instruction tries to access a memory address, for which the virtual-to-physical translation is not cached in the TLB. When a fault occurs the `mmu_bus_fault`-routine is called. It saves registers and disables interrupts in a way similar to a normal interrupt. Then it calls the C-routine

`handle_mmu_bus_fault`, which is supposed to fix the fault, whichever it might be. Upon return, `ret_from_intr` is called, but instead of `_Rexit`, a special function `_RBFexit` is used. It will restore registers and restart the instruction that caused the fault originally.

- IRQ 15: The multiple interrupt occurs when there are several interrupts active at the same time. The `multiple_interrupt`-routine will first save registers, disable interrupts and then process the first individual interrupt waiting by calling its shortcut into the `IRQxx_interrupt`-routine as shown in the figure. The shortcut is called since registers are already saved. The interrupt is then processed in a normal way and if there are still more than one interrupt active afterwards, `multiple_interrupt` will be called again to process the next one.

3.1.2 Interrupt Handling on Linux/CRIS with RTAI

So far, the normal interrupt handling paths in Linux/CRIS have been described. It is now time to introduce the hardware abstraction layer, RTHAL, in the kernel. When doing so, some factors have to be considered:

- The longer interrupts are disabled in the kernel, the longer worst-case latencies will show up in the system. It is important to have interrupts disabled only when absolutely required. Otherwise a real-time handler may be blocked longer than necessary.
- It is important that the code running when interrupts are disabled is time-predictable and not dependent on the state of the Linux kernel.
- As said earlier, when the kernel is configured to include RTHAL it should still (as far as possible) operate as usual when RTAI is not mounted. When RTAI is mounted the layer should provide the necessary functionality.

Interrupt Paths with RTHAL

In Figure 3.2, the interrupt handling paths with RTHAL present in the kernel are shown. The following changes and additions have been made, compared to the plain kernel:

- The call to `do_IRQ` has been replaced with a call through the `rthal`-struct, *i.e.* `rthal.do_IRQ`.
- For the timer interrupt `rthal.do_timer_IRQ` is called instead.
- In `IRQxx_interrupt` the “unmask interrupt” has been replaced with a call to `rthal.unmask_if_not_rtai`.

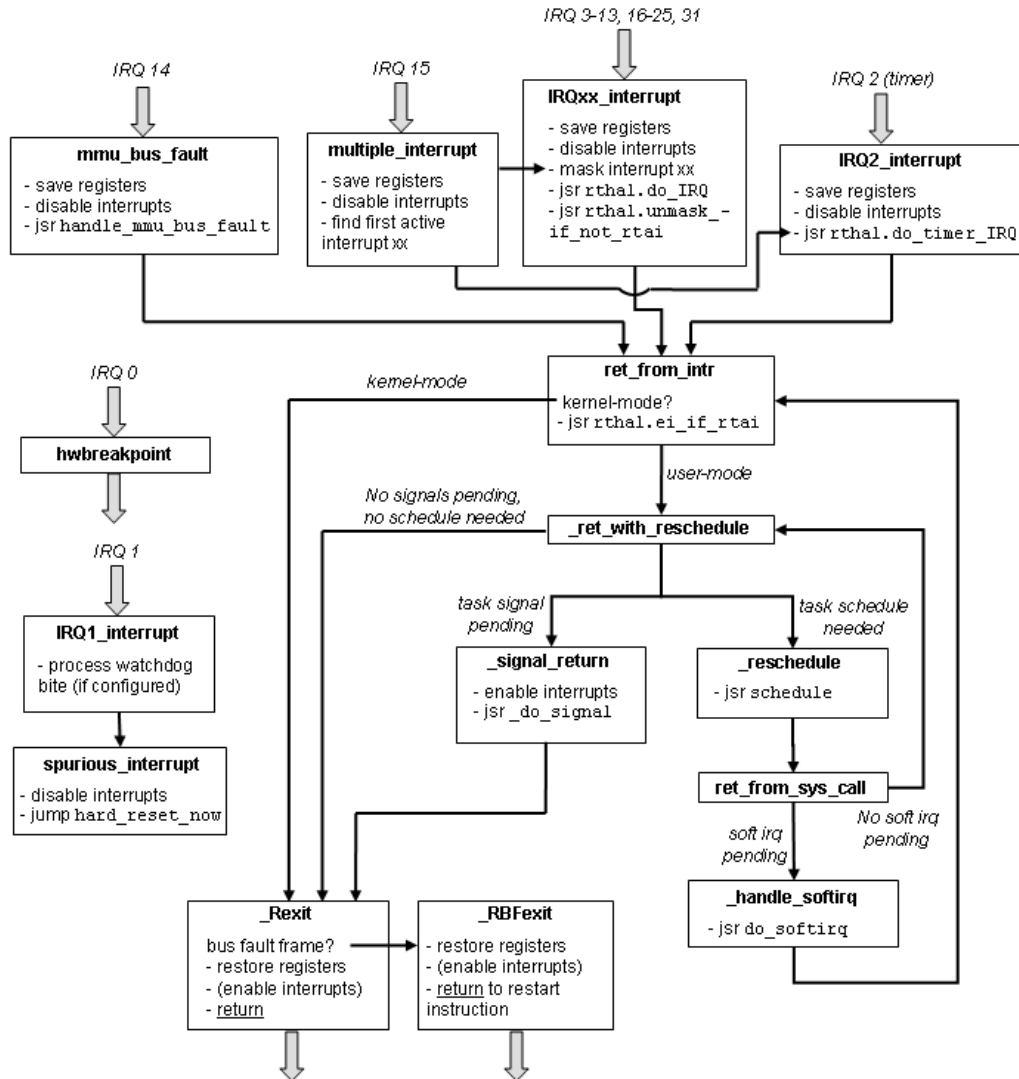


Figure 3.2: Interrupt handling paths with RTHAL present

- In `ret_from_intr` a call to `rthal.ei_if_rtai` has been added.

Table 3.1 shows what the added functions do with and without RTAI mounted.

rthal-function	RTAI not mounted	RTAI mounted
<code>do_IRQ</code>	A call to the standard <code>do_IRQ</code> -function.	A call to <code>dispatch_irq</code> in the RTAI module.
<code>do_timer_IRQ</code>	A call to the standard <code>do_IRQ</code> -function.	A call to <code>dispatch_timer_irq</code> in the RTAI module.
<code>unmask_if_not_rtai</code>	Unmasks the interrupt.	Does nothing.
<code>ei_if_rtai</code>	Does nothing.	Enables interrupts.

Table 3.1: Functions added by RTHAL to the interrupt handling paths

Motivation

By letting the calls to `do_IRQ` go through the `rthal`-struct, minimal changes have to be made to the low-level interrupt handling which saves registers *etc.* Then, when RTAI is mounted it is easy to redirect the calls to RTAI functions instead.

Since an interrupt may be pended by the `dispatch_irq`-function, the installed Linux interrupt handler that normally would acknowledge the interrupt is not always run directly. The interrupt can therefore not be unmasked as usual if RTAI is mounted. Thus the need for `unmask_if_not_rtai`. Pended interrupts are unmasked after they have been delivered to Linux. However, if a real-time handler is installed it is run directly and has the responsibility to acknowledge and then the interrupt is unmasked directly.

If the processor was in user-mode when an interrupt occurred and a schedule is needed then the path where interrupts are disabled is quite long and more importantly unpredictable³. Should an interrupt occur during this time, its processing would have to wait until interrupts are re-enabled (in either `_signal_return` or `_Rexit`). Without RTAI this is not the case since the schedule and soft-irq functions enable interrupts at certain points, but when RTAI is mounted these instructions will only be soft, *i.e.* setting a flag; the real interrupts would remain disabled. It is crucial that an installed real-time handler is not delayed. To prevent the unpredictable delay `ei_if_rtai` is called in `ret_from_intr`, thus hard-enabling interrupts at an early stage. It should

³The time for schedule and possible soft-irq processing.

be safe to do so since `schedule` and `do_softirq` protect themselves during some critical parts by disabling interrupts soft, during these parts RTAI will not send any interrupts to the Linux kernel. A nested interrupt would not take the path through `ret_with_reschedule` again, as all these routines are called from kernel mode.

3.2 RTAI Kernel Module

RTAI consists of a set of modules working together. Some of them add the functionality of a real-time scheduler and FIFO queues, while others add for example the POSIX pThreads and pQueues API. The most significant one though is the main, architecture specific, module called `rtai.o`. It contains for example the functions for mounting RTAI, trapping interrupts, installing real-time handlers *etc.* In this section the basic functionality of the module will be described.

Before RTAI has been mounted, all interrupt-related calls go through the `rthal`-struct, but they do the same thing as usual, for example hard-disabling and enabling interrupts. As RTAI is mounted the old `rthal`-struct is saved and a new one installed. The new `rthal` will direct the calls to various internal functions in the RTAI module instead, thus allowing RTAI to take control over the system.

Some of the internal key functions in the RTAI module are:

- `linux_cli`: The call `rthal.disint` is directed to this function. It only sets the flag, which says that Linux has disabled interrupts. After a call to this function, Linux will not receive any interrupts to its handlers. Before RTAI was mounted, interrupts would have been hard-disabled when `rthal.disint` was called.
- `linux_sti`: The call `rthal.enint` is directed to this function. It clears the flag, which says that Linux has disabled interrupts. Any pending interrupts will be delivered to Linux (by calling the standard `do_IRQ`-function for each one) and afterwards Linux will receive interrupts as soon as possible again. Before RTAI was mounted, interrupts would have been hard-enabled when `rthal.enint` was called.
- `dispatch_irq`: As mentioned before in section 3.1.2, a call to `rthal.do_IRQ` is directed to this function. So, when an interrupt occurs and RTAI is mounted, this function will be called. It checks whether there is a RT-handler installed on the specific interrupt and if so calls that handler. If instead there is no RT-handler installed, the interrupt is pended to

Linux. At the end of `dispatch_irq` there is a check to see if Linux has its interrupts enabled (indicated by the flag) and if so `linux_sti` is called, which as described previously will deliver the interrupt to Linux.

- `dispatch_timer_irq`: This is similar to `dispatch_irq` but is used for timer-interrupts. The timer interrupt must be acknowledged directly since the real-time scheduler is dependent upon this interrupt. The watchdog must also be reset if it is configured in the kernel. Finally, the interrupt can be pended for Linux as usual or a RT-handler called.
- `linux_save_flags`: In many cases Linux disables interrupts before entering a critical region. Sometimes it wants to be able to restore them afterwards (if they were on before the critical region they should be enabled again). Then the CPU flags must be saved. When RTAI is not mounted, a call to `rthal.getflags` will return the real CPU flags, but when RTAI is mounted, `linux_save_flags` is called instead. It will only return the interrupt disable/enable-flag set by `linux_cli/linux_sti`.
- `linux_restore_flags`: This function is called to restore the flags saved previously by `linux_save_flags`. If the flags indicate that interrupts should be enabled for Linux, `linux_sti` is called and otherwise the flag is set to disable interrupts for Linux.

Other vital functions in the RTAI module are of course the API functions, such as `rt_request_global_irq` for installing a real-time interrupt handler and `rt_pend_linux_irq` for pending an interrupt to Linux from a real-time handler.

3.3 Implementation Choices

This section is intended to highlight some implementation choices made.

3.3.1 Timers and Cycle Counter

The ETRAX 100LX processor provides two general timers in hardware, each one has an 8-bit counter, which is loaded with a value before the timer is started. The timer then counts down to one from the programmed value and generates a timer interrupt. The counter then “wraps”, *i.e.* it starts over again with the loaded value. The clock frequency of the timer, *i.e.* how often the counter is decremented, can also be adjusted within certain limitations.

Normally, in Linux/CRIS, only one timer is used and the other one is available to e.g. device drivers. The timer used is run at 25 kHz and the counter

is loaded with 250, this will generate a timer interrupt every 10 ms.

In the RTAI implementation on CRIS, both available timers are used in a *cascade mode* that results in a 16-bit counter instead of an 8-bit when using only one timer. When RTAI is loaded the cascaded timer is programmed to run at 6.25 MHz, which gives the real-time scheduler the possibility to program an interrupt with a theoretical accuracy of 0.16 μ s. Compared to the default configuration where the accuracy would be 40 μ s this is a great improvement. By default the counter is loaded with a value of 62500 and this setting will generate an interrupt every 10 ms. In this way the standard time interval is not affected when no real-time tasks require it.

The timer cascade-mode in RTAI can be disabled in the kernel config, for example if one of the timers is needed in a device driver. However, disabling this functionality will result in a significant loss of accuracy.

There is a potential problem related to the fact that ETRAX, as many other architectures, does not have a cycle counter⁴ implemented in hardware. A cycle counter is used to keep track of time. In the implementation of RTAI, the same counter used in the timer has been used to count cycles⁵. Although this approach is acceptable, it has the problem that the function updating the cycle counter has to be called at least once in every time window (*i.e.* between two timer-counter wraps) or else the cycle counter will become corrupt. The normal time window in Linux/CRIS is 10 ms, but when using periodic real-time tasks, this can be set to any arbitrary value. If the time window is very small, then it could happen that the interrupts are disabled over the whole time window and therefore corrupting the cycle counter. A solution to this problem could be to implement either a cycle counter or a larger timer counter in hardware in future products.

3.3.2 Unaffected Interrupts

As shown in Figure 3.2 the interrupts with numbers 0 and 1 are not caught and pended. This is because they are related to debug and watchdog respectively.

3.4 Limitations

Because of the limited time available for this project there are some architecture specific parts of RTAI that are still not ported to CRIS, they are:

⁴An internal counter which increments with the clock frequency of the CPU.

⁵Actually the cycles counted here are the clock cycles of the timer, not the CPU.

- Shared Memory (communication primitive)
- LXRT (hard real-time support in user-space)
- RTAI System Requests. This is a way to install a function that can later be called from user-space but will run in kernel-space. The call is similar to normal system requests, but arbitrary functions can be installed. The parts necessary in the main module is implemented but the calling mechanism is not.

There are also a number of examples shipped with RTAI which show how to use the API, most of these examples can be compiled although they are not always applicable to CRIS.

As of this writing the MMU bus fault is not pended, but is instead run directly. The `handle_mmu_bus_fault`-routine (which handles the fast TLB-fill) must be run directly, otherwise the execution would not be able to continue. Hence, the interrupt can not be caught and pended in the normal way. Some architectures handle the TLB-fill in hardware so that it may look as if the whole interrupt is pended. This is not the case on CRIS since the data structures are implemented in software. It could however be further investigated if it is possible to pend the slow `do_page_fault`-part of this interrupt in an easy way.

Normally a pointer to the `pt_regs`-struct is sent to `do_IRQ`, but as no interrupt handler called through this routine is using the struct, no effort has been made to save it when pending and then later send it along when dispatching the interrupt to Linux. Instead a “dummy” struct is used and this is also the case on at least the ARM and PPC architectures. This approach improves interrupt latency performance, as the whole struct would otherwise have to be copied, since it is saved on the kernel stack and would have been destroyed before being used. The MMU bus fault is using the struct, but this interrupt is not pended as mentioned above.

3.5 Known Problems

There is one known stability problem with the implementation of RTAI on CRIS. It usually makes Linux stop responding, although most times the real-time tasks are allowed to continue. The problem only shows up under heavy network load and can be triggered by running two flood pings (`ping -f`) simultaneously towards the developer board. Besides this problem, which has only shown up under flood pings, the port seems very stable.

It should be noted that as of this writing the `rtai.o` module is actually linked directly into the kernel on the CRIS port. This is not the customary way to do it, instead it should be loaded as a module. The approach has been taken because it minimized the stability problem mentioned above.

As to real-time scheduling, the period of a periodic task is sometimes not correct. This only occurs when cascaded timers are disabled in the kernel config and the scheduler is set to run in oneshot mode. The period has then been seen to be both one half and sometimes one third of the desired period.

Chapter 4

Evaluation

In order to evaluate the performance and verify the basic functionality of the RTAI port a number of tests were conducted. This chapter describes how these tests are defined and performed. When the tests are applicable also to standard Linux, a comparison is made. Finally, the results are presented and discussed.

Two standard PC workstations (referred to as PC1 and PC2), one developer board with an ETRAX processor and a logic analyzer have been used during the tests. For a full description of the test environment (hardware configuration, software versions, load definitions *etc.*) see Appendix A. In Appendix B the source code for the test programs can be found.

4.1 Definitions and Measurement Approaches

4.1.1 Interrupt Latency

Interrupt latency is the amount of time between when an interrupt is generated (internally or by an external device) and when an installed interrupt handler starts to execute. When the system is in an idle state this time is very short, but will be longer when for example other interrupts are processed.

Interrupt latency is a very important measurement in a real-time system. It affects many other performance aspects such as scheduling precision and interrupt task latency. A worst case interrupt latency yields for example a lower boundary for the worst case interrupt task latency. The most interesting tests are those with load as the worst latencies show up under load.

In order to measure interrupt latency, a signal from the parallel port of PC1 is changed from high to low. The signal is connected to a pin on a general I/O

port of the developer board. The general I/O port is configured to generate an interrupt when the signal is low. An installed interrupt handler will send a response on the parallel port of the developer board when it is run. PC1 will change the signal back to high again when it receives the response and wait some time before repeating the process. The timing sequence is shown in Figure 4.1.

The time interval between when the PC changes its signal and when the interrupt handler responds on the developer board is taken as a measurement of interrupt latency. The process is repeated many times and the time intervals are recorded by the logic analyzer.

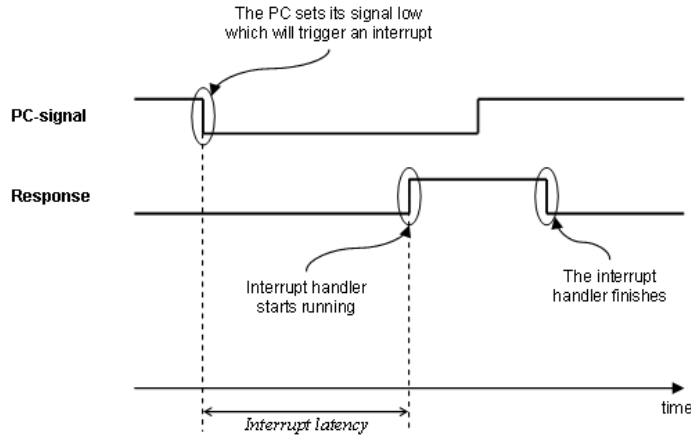


Figure 4.1: Interrupt latency timing

Measurements are performed using four different configurations, all with and without load (coming from PC2 in this case):

- RTAI mounted with a real-time interrupt handler.
- RTAI mounted with a standard Linux interrupt handler.
- Without RTAI but with the hardware abstraction layer present and compiled into the kernel.
- Without RTAI and a plain Linux kernel.

4.1.2 Interrupt Task Latency

Consider an example of a real-time system which uses an external device that generates data and signals to the system via an interrupt when this data is ready. A high-priority real-time task reads the data and processes it within

a certain amount of time. Other lower priority tasks may be executing in the system, such as user-interface tasks, but they are not time critical. The amount of time between when the external device is ready and signals with an interrupt and when the task is run and can start processing the data is of importance. This time interval is defined as *interrupt task latency*.

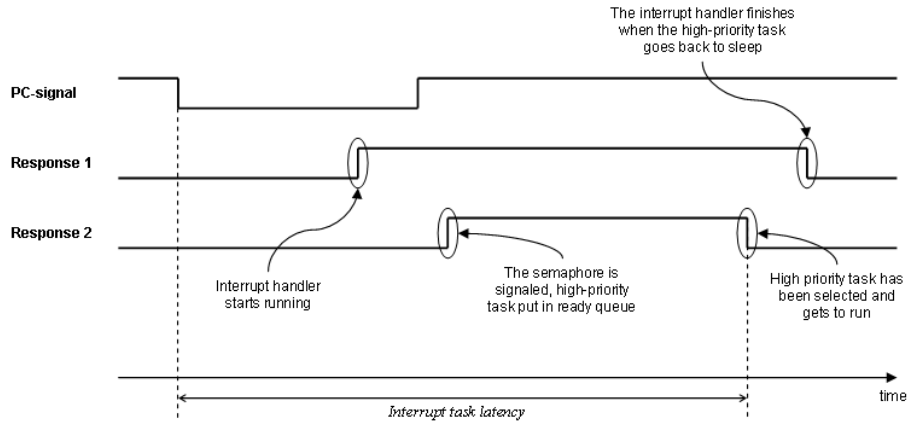


Figure 4.2: Interrupt task latency timing

In order to measure interrupt task latency a signal is generated on the parallel port of PC1. The signal triggers an interrupt on the developer board and an installed handler will respond with a high signal on a response pin on the parallel port. The handler will set another response pin high before signaling a semaphore on which a high priority task is waiting. The task should now start running as soon as possible, since it has the highest priority and its waiting condition has been fulfilled. When the task starts to run, it immediately clears the second response pin and then waits on the semaphore again, thus allowing the interrupt handler to finish and take down the first response pin. In Figure 4.2 the whole timing sequence is depicted. The time interval between when the PC signal is generated and when the second response pin is cleared by the task, is measured by the logic analyzer and taken as a measurement of the interrupt task latency.

The measurements are also performed on standard Linux, except that the task is a kernel thread and the semaphore is replaced with a wait queue.

Both tests are performed with and without load. The load will in this case come from PC2.

4.1.3 Scheduling Latency

A high priority real-time task ready to run should preempt any lower priority tasks that may be executing. The amount of time it takes for the scheduler to preempt a lower priority task, select the high priority task and start it is called *scheduling latency*, that is, the time between when the high priority task is ready to run and when it actually starts to run. There are different ways in which a task can be made ready to run. Two cases have been studied. In both cases, measurements are made with and without load.

Case 1: Using a semaphore

A low priority task signals a semaphore on which a high priority task is waiting. A pin is set high on the parallel port before the semaphore is signaled. When the high priority task starts to run, it immediately clears the pin and waits for the semaphore again.

Case 2: Using suspend/resume

In this case, a low priority task resumes a high priority task that previously suspended itself. A pin is set high on the parallel port before the high priority task is resumed. When the high priority task starts to run, it immediately clears the pin and then suspends itself again.

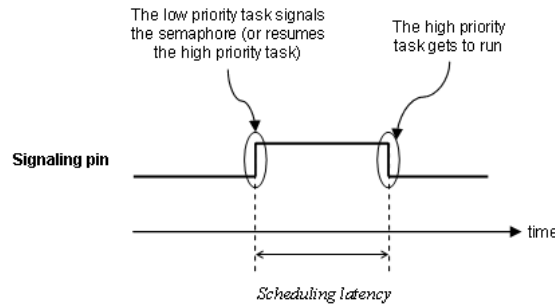


Figure 4.3: Scheduling latency timing

4.1.4 Scheduling Precision

Scheduling precision is a measurement of how well the scheduler is able to maintain a desired fixed time-period for a task.

For example, if the period of a periodic real-time task is set to be $T \mu\text{s}$ and the task is started at $t = 0 \mu\text{s}$, then ideally it should run at times $t = T, 2T, 3T, \dots \mu\text{s}$ regardless of any lower priority tasks (such as Linux) that

may be running in between. It is likely that it will not run exactly at these points in time. The first time it will execute will most likely be at $t = T + h$ μs where h is a small non-zero time interval. This deviation from the correct time is known as *scheduling jitter*. The scheduling jitter depends on how often the scheduler is run which in turn depends on the granularity of the timer, processing of interrupts *etc.*

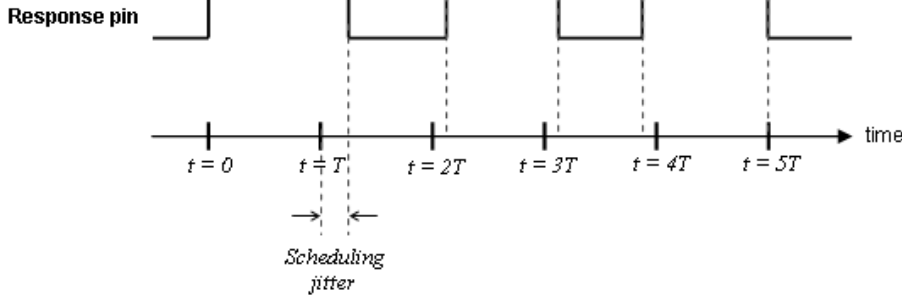


Figure 4.4: Scheduling precision timing

Scheduling jitter is measured by letting a periodic real-time task toggle a pin on the parallel port of the developer board and then wait the rest of its period. The time intervals between the toggles are measured by the logic analyzer. These intervals are compared to the desired period and the deviation is taken as a measurement of the scheduling jitter. Note that this differs slightly from what is shown in Figure 4.4. If the task is delayed, the measured period will be too long. Even if the next execution starts at the correct time, the next measured period will be too short.

Two tests are performed, one without any load and one when a number of lower priority real-time tasks are added to the load used in the other tests.

The scheduler is set to periodic mode and some different periods are tested.

4.1.5 Communication Overhead

When sending data from a real-time task or handler to a Linux process it is interesting to see how large the overhead is. That is how much processing time does it take to read from or write to a communication primitive such as a FIFO buffer. Besides FIFO buffers can for instance shared memory be used to communicate, but at the time of this writing it is not yet ported to CRIS. Therefore FIFO buffers will be used in the following tests. From within a real-time task a FIFO buffer is accessed using API function calls and from Linux it is accessed through an entry in the `/dev`-directory using normal file operations such as `open`, `close`, `read` and `write`.

This test is not really a test of the real-time performance of the system, but will present results that can be useful when designing such a system. The results can for example be used to make sure that the system is not overloaded when sending data from RT-tasks running in kernel-space to processes running in user-space.

To measure the communication overhead one small and one large message is sent and read a number of times by an RTAI task. A pin of the parallel port is used to signal when the reading and writing starts and stops. Note that it is the same message that is first written and then read by the task. The interrupts are hard-disabled during the reading and writing to obtain the correct times. The time intervals obtained in this test will be the actual times it takes to put or get something to or from the buffer.

In addition to these measurements, an approach to measure the whole communication process overhead between a user-space Linux process and a real-time task will be performed. Messages of varying size will be sent from the real-time task and read in the user-space process and then the message is sent back again to the real-time task. A pin on the parallel port is set high by the real-time task before sending a message and set low when the message has been read back by the task. The time interval is measured by the logic analyzer. This test will be performed without any other system load in order to obtain a good estimate of the time it takes to use FIFO buffers as a means of communication between user-space and real-time.

4.2 Results and Discussion

All values presented in this section are in μs if not stated otherwise.

4.2.1 Interrupt Latency

As mentioned earlier, the measurements are performed using four different configurations, all with and without load:

1. RTAI mounted with a real-time interrupt handler.
2. RTAI mounted with a standard Linux interrupt handler.
3. Without RTAI but with the hardware abstraction layer present and compiled into the kernel.
4. Without RTAI and a plain Linux kernel.

A comparison between Linux and RTAI is made in this section. The comparison is based upon the two normal configurations: RTAI with an RT-handler (1) and plain Linux (4). The other two configurations are motivated to see how RTAI affects the Linux kernel and are only present in the interrupt latency measurements.

Table 4.1: **Interrupt latency without load**

	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
1. RTAI RT-handler	5.3	25.9	5.8	5.6	0.4	225110
2. RTAI Linuxhandler	16.0	384.0	18.9	17.9	4.7	206995
3. Linux RTHAL	7.5	44.7	9.3	9.3	0.8	217201
4. Linux plain	3.5	32.5	4.3	3.9	0.9	254033

Table 4.2: **Interrupt latency with load**

	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
1. RTAI RT-handler	4.4	64.9	17.9	18.5	5.7	223559
2. RTAI Linuxhandler	15.8	1209.0	182.8	115.6	165.9	246465
3. Linux RTHAL	6.6	154.3	23.0	22.7	7.8	243723
4. Linux plain	5.2	162.9	14.3	11.3	6.4	476052

RTAI RT-handler (1) vs. Plain Linux (4)

The difference between RTAI and Linux is small without system load. RTAI has a slightly higher mean value ($5.8 \mu s$ compared to $4.3 \mu s$ for Linux), which is due to the hardware abstraction layer. It imposes a slight, predictable increase in latency. This increase is visible when looking at the distribution in Figure 4.5. It appears not much has been gained with RTAI when there is no system load.

As load is applied, the difference becomes more obvious. While Linux still has a better average, the worst case measured latency is 2.5 times higher for Linux compared to RTAI ($162.9 \mu s$ for Linux and $64.9 \mu s$ for RTAI). Figures 4.7 and 4.8 are plots of measured latencies for Linux and RTAI respectively. The samples appear more limited in RTAI, Linux has for example several samples over $70 \mu s$ while RTAI has none.

The distribution is shown in Figure 4.6. The larger average latency in RTAI can be explained by the interrupt dispatching function. It is mainly a loop

running with interrupts disabled until an active Linux interrupt is found. The routine can be optimized in future versions.

RTAI with a Linux Interrupt Handler (2)

RTAI is mounted in this test, but the measurements are performed on a Linux interrupt handler. By comparing the results with those of plain Linux, it can be seen how much a standard Linux interrupt handler is delayed by RTAI. As seen in Tables 4.1 and 4.2 there is a significant performance loss, especially when observing maximum values. The increased latencies are mainly the result of RTAI pending interrupts for Linux.

When RTAI is mounted, a normal Linux interrupt handler is not run directly, but the interrupt is still processed at a low-level and pended to Linux. Before the interrupt handler is run, interrupts are enabled in the hardware. Thus, a number of interrupts which all require low-level processing, could occur between when a specific interrupt is pended and when the handler is run. When load is applied, many interrupts will occur and the handler will be delayed even further. Note that it is not until the handler is run that the response pin is changed and the measurement value taken.

In plain Linux, interrupts are disabled from the moment the interrupt occurs to when the handler gets to run and therefore is not delayed in this way.

Linux with RTHAL Present (3)

The third test shows the slight overhead that the hardware abstraction layer imposes on the kernel by substituting some assembler macros for function calls. Note that RTAI is not mounted in these tests.

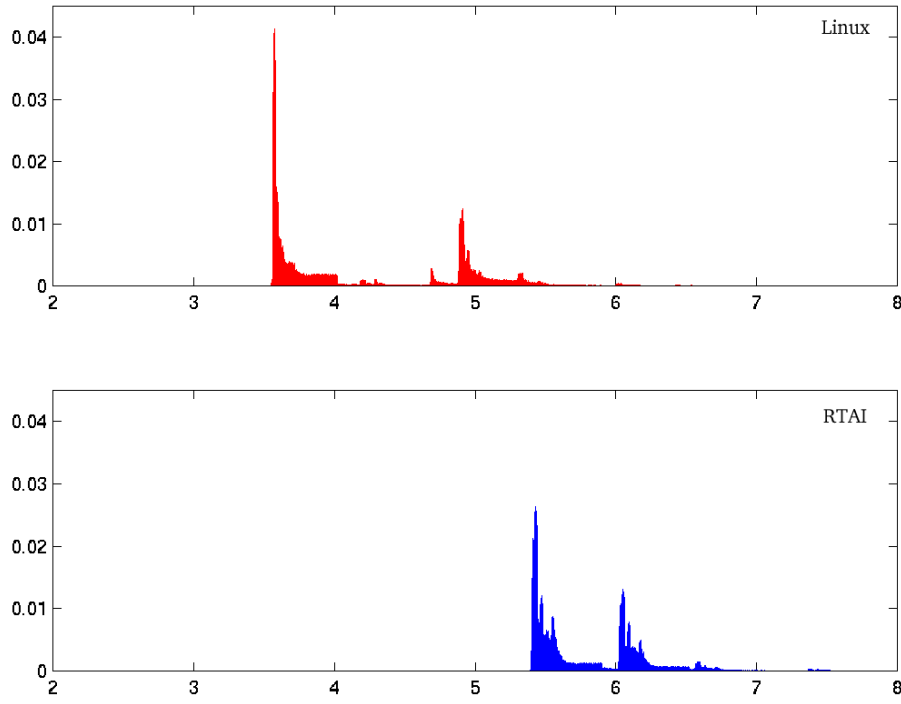


Figure 4.5: Interrupt latency distribution without load, Linux vs. RTAI

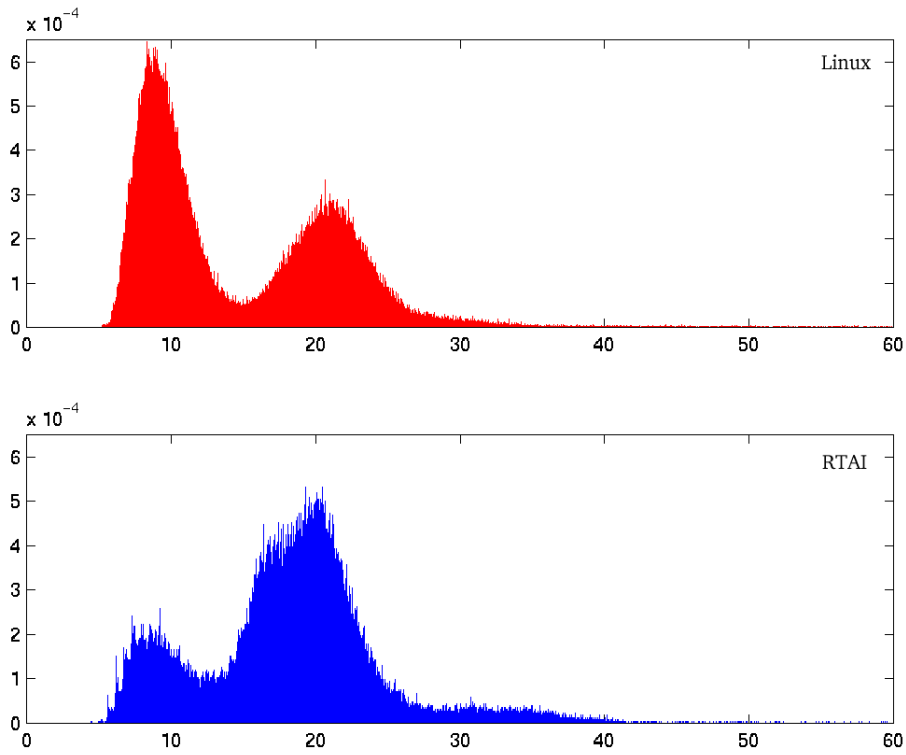


Figure 4.6: Interrupt latency distribution with load, Linux vs. RTAI

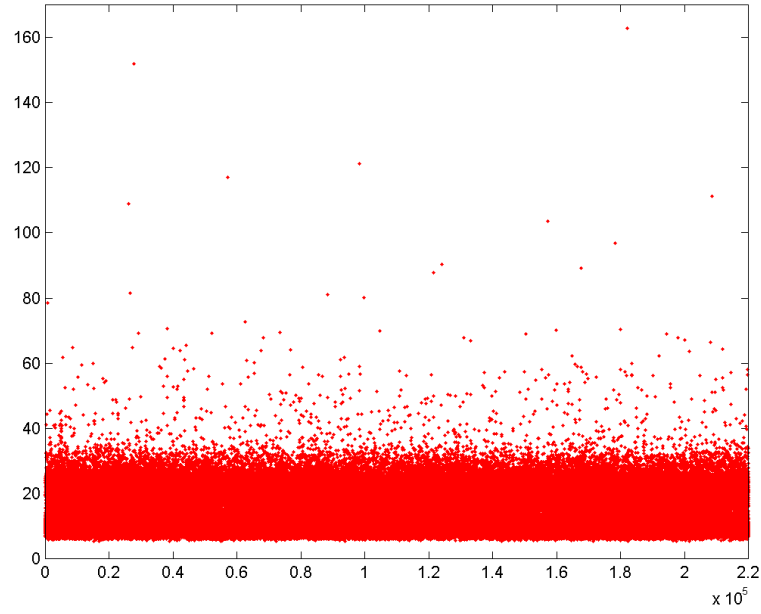


Figure 4.7: Interrupt latency, Linux with load (220 000 samples)

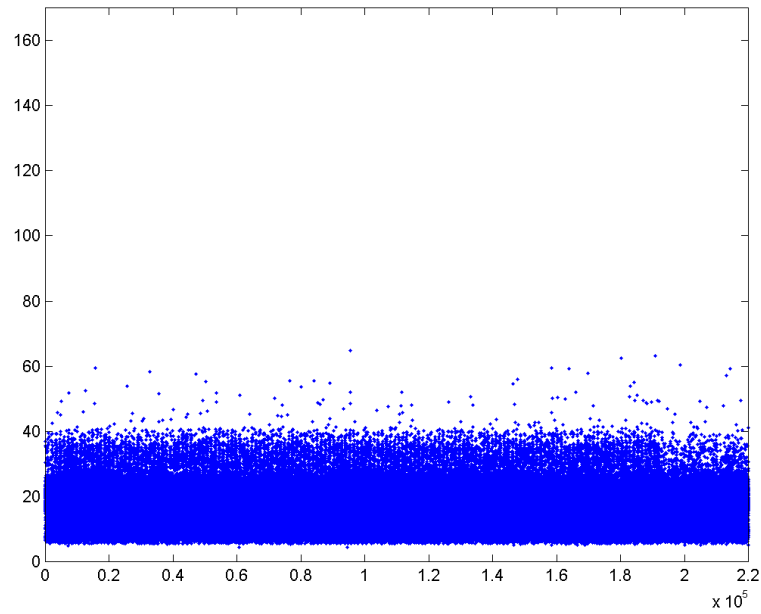


Figure 4.8: Interrupt latency, RTAI with load (220 000 samples)

4.2.2 Interrupt Task Latency

Table 4.3: Interrupt task latency without load

	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
RTAI	28.0	68.0	33.2	31.0	4.3	20496
Linux (plain)	44.8	332.3	49.7	49.4	5.2	10363

Table 4.4: Interrupt task latency with load

	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
RTAI	37.6	142.0	63.0	63.3	9.4	12883
Linux (plain)	57.0	84585.0	3147.5	471.0	6393.7	10226

When comparing RTAI to Linux, one immediately notes the big difference in the load tests. Interrupt task latency is composed of essentially two different time intervals as shown in Figure 4.2. The first is interrupt latency and the second is scheduling latency. While the difference in interrupt latency is relatively small with load (mean 17.9 μ s for RTAI and 14.3 μ s for Linux), the difference in interrupt task latency is large (mean 63.0 μ s for RTAI and 3147.5 μ s for Linux). This is due to how the Linux kernel thread is scheduled. When the semaphore is signaled in RTAI by the RT-handler, it immediately schedules while interrupts remain disabled. In Linux, the interrupt handler finishes and interrupts are reenabled before the kernel thread is scheduled. This means that during heavy load a lot of interrupts will be processed before the thread is actually started.

4.2.3 Scheduling Latency

Table 4.5: Scheduling latency without load

	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
Semaphore	18.5	29.2	20.9	19.3	2.7	23784
Suspend/Resume	10.5	18.4	11.8	10.6	1.9	17279

Tables 4.5 and 4.6 show that signaling a semaphore is slower than directly resuming the high priority task. However, using a semaphore is much more realistic in applications containing many threads.

Table 4.6: **Scheduling latency with load**

	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
Semaphore	24.7	57.7	30.5	29.8	3.3	15167
Suspend/Resume	17.6	46.1	21.4	20.8	2.7	18155

The min values of 18.5 μ s (semaphore) and 10.5 μ s (suspend/resume) in the tests without load are probably close to the actual times it takes to schedule. However, as interrupts are enabled during these tests, the time increases as shown in Table 4.6 when load is applied. Interrupts can occur after the low priority task has set the response pin high, but before the schedule function has had a chance to disable interrupts. Hence the increase in measured time intervals. Actually, the mean values increases by 9.6 μ s in both test cases. This increase is close to the increase in mean values observed when applying load to the interrupt latency tests (12,1 μ s).

4.2.4 Scheduling Precision

The values shown in the tables below are the jitter of the task periods and not the periods themselves. In order to obtain the jitter, the desired period was subtracted from the sampled values. In all but the plots, the absolute value of the deviation is used and shown. Note that, despite the period being written in ms in the tables, the measured values are in μ s. In the test where the task period equals 1 ms, the timer is also programmed to give interrupts with this period. In the other tests, the timer is programmed with Linux standard value of 10 ms.

Table 4.7: **Scheduling jitter without load**

<i>Period</i>	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
1 ms	0.00	35.09	1.03	0.27	1.80	135522
50 ms	0.00	34.86	2.55	2.37	1.53	21070
200 ms	0.16	23.87	9.56	9.56	2.42	15005

Table 4.8: **Scheduling jitter with load**

<i>Period</i>	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
1 ms	0.00	33.97	3.22	2.55	2.72	253415
50 ms	0.00	90.19	10.50	6.77	10.72	22723
200 ms	0.00	83.22	13.42	10.78	10.56	12012

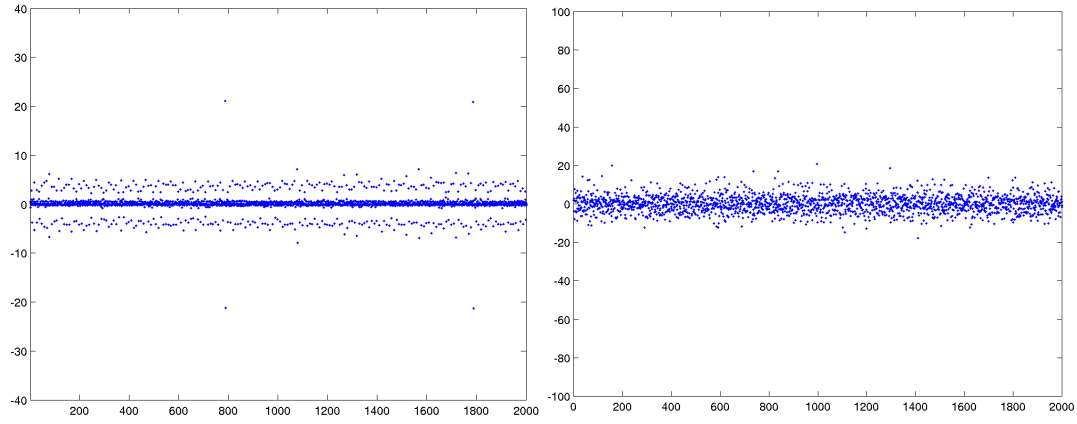


Figure 4.9: 1 ms without load and with load (see comments on next page)

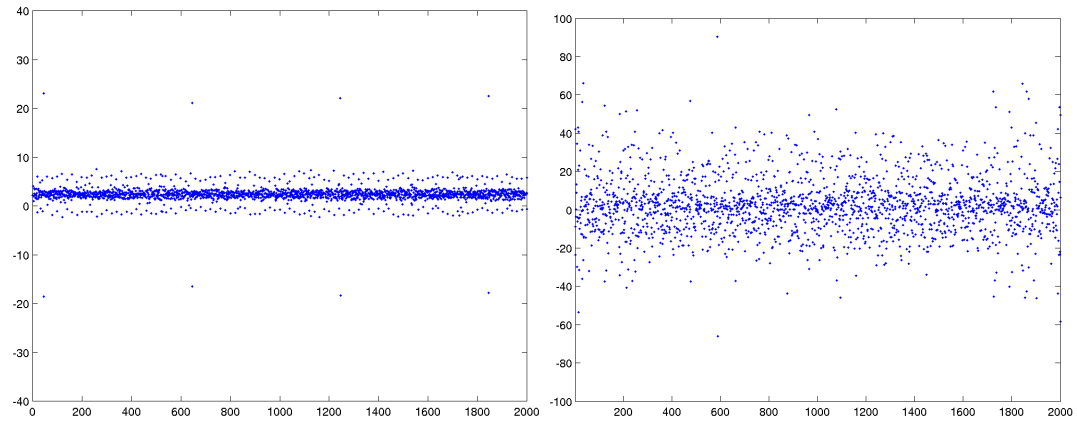


Figure 4.10: 50 ms without load and with load

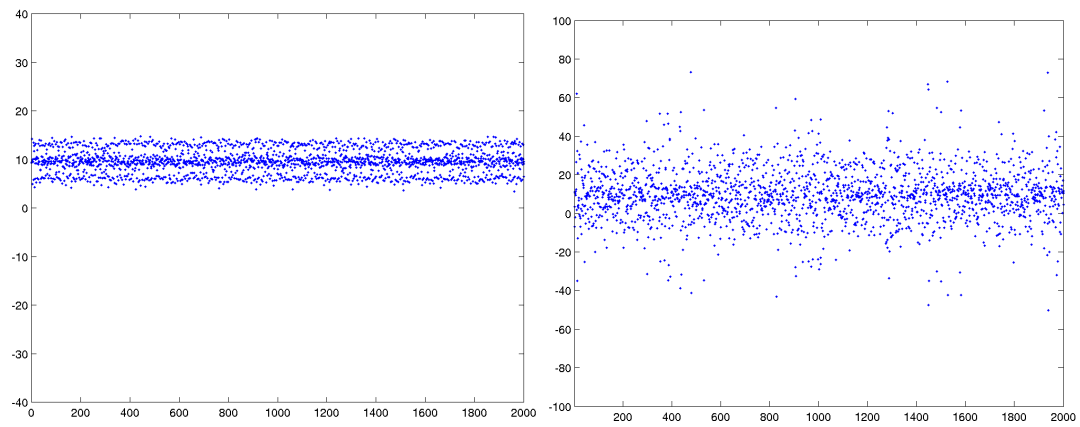


Figure 4.11: 200 ms without load and with load

Comment: The load script was not run on the developer board during the 1 ms load test because of stability problems.

As can be seen in Table 4.7 and Table 4.8, the jitter is much worse when the system is under heavy load. This comes from the fact that the system disables all interrupts a short time when receiving new interrupts. If a new interrupt is generated when the interrupts are disabled, it will not be handled by the system until the interrupts are reenabled again. Periodic tasks can be delayed this small fraction of time because they are started from the timer interrupt.

In Figure 4.9, Figure 4.10 and Figure 4.11 the first 2000 measurements with the desired period withdrawn is shown, both without and with load. It can be seen that the median drifts upwards when the period increases. There is a slight offset in Figure 4.9, a slightly larger one in Figure 4.10 and in Figure 4.11 it seems to be around $10\ \mu\text{s}$. In a perfect system this offset should be zero. There is however no drifting in the period, it is constant over time. In the 200 ms period test case, a period of 200.00956 ms appears to be used instead. It is a very small offset from the desired value, but nevertheless an offset. We believe this behaviour to be caused by an internal clock that does not oscillate with the desired frequency of exactly 25 MHz, but instead has a very slight offset. However, as long as this offset is constant it should not cause any real problems as it can be compensated for.

The maximum deviation from the correct period is $83.22\ \mu\text{s}$. While this may sound like a large value at first, it is actually less than 0.09 ms. Most of the measured jitter comes from the interrupt latency. The worst case in the scheduling precision test corresponds to the worst case in the interrupt latency test.

In this test, no comparison to Linux has been made because Linux can only schedule tasks with a granularity of 10 ms while RTAI has a granularity of one timer tick (approximately $0.16\ \mu\text{s}$ in RTAI/CRIS).

4.2.5 Communication Overhead

Table 4.9: Communication Overhead, Write

<i>Size</i>	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
4 bytes	15.80	28.36	16.32	16.00	0.76	17823
1024 bytes	53.96	71.56	55.40	54.34	1.87	14774

Table 4.10: Communication Overhead, Read

<i>Size</i>	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
4 bytes	7.72	10.72	8.01	7.80	0.56	17823
1024 bytes	33.52	35.22	34.97	34.96	0.15	14774

As can be seen in Table 4.9 and Table 4.10, writing takes more time than reading does. This is probably caused by some extra checks required in the write case. Both the write and read functions copy the message when called. The difference in time between writing and reading different message sizes is not linear in the size of the message. There is a small constant offset involved in calling the function, disabling the interrupts and some checks.

Table 4.11: Communication Overhead, rt-task to user-space and back again

<i>Size</i>	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
4 bytes	189.63	610.87	198.88	196.81	7.40	15040
1024 bytes	334.22	671.01	345.37	343.94	7.06	18271

In Table 4.11, it is the minimum values that show the most accurate time as the interrupts are enabled. The test tries to measure the time it takes to send a message from a rt-task to a user-space task which reads it and then sends it back again for the rt-task to read. Thus, it is the total time to and from user-space which is measured.

Chapter 5

Final Improvements

This chapter describes modifications made to the implementation after the tests in the evaluation were made and most of the report written. It contains some important results.

Motivation

When looking at the test results in Chapter 4, it was obvious to us that the implementation could be improved. For example, the interrupt latency distribution in Figure 4.6 shows a big difference between RTAI and Linux. RTAI has most of its samples in the second peak while Linux has most of them in the first.

Modification

As mentioned in section 4.2.1, there is a long loop running with interrupts disabled in the interrupt dispatch function. This loop has now been modified to enable interrupts once every cycle.

Improvements

As a result of the modification, the stability has improved significantly. It was previously possible to bring down the system almost immediately by running two ping floods towards the developer board. This is not possible any more. In fact, the system has been running the interrupt latency test for several days under three ping floods without any problem. However, some stability problem still exists. It is for example possible to trigger it by running the scheduling precision test for a few minutes with a period of 1 ms and a number of ping floods present.

Although no changes to the logic of the program was made during the improvement, the *timing* was altered. This further implies that the stability

problem is not due to a direct programming error in the RTAI code itself, but rather to the behaviour introduced by RTAI in the kernel.

5.1 Test results

Due to time constraints, we were only able to rerun a small subset of the tests in Chapter 4.

5.1.1 Interrupt Latency

Figure 5.1 shows the interrupt latency distribution before and after the modifications. It is a clear improvement. There are now many more samples in the first peak and the distribution is also more concentrated around the peaks.

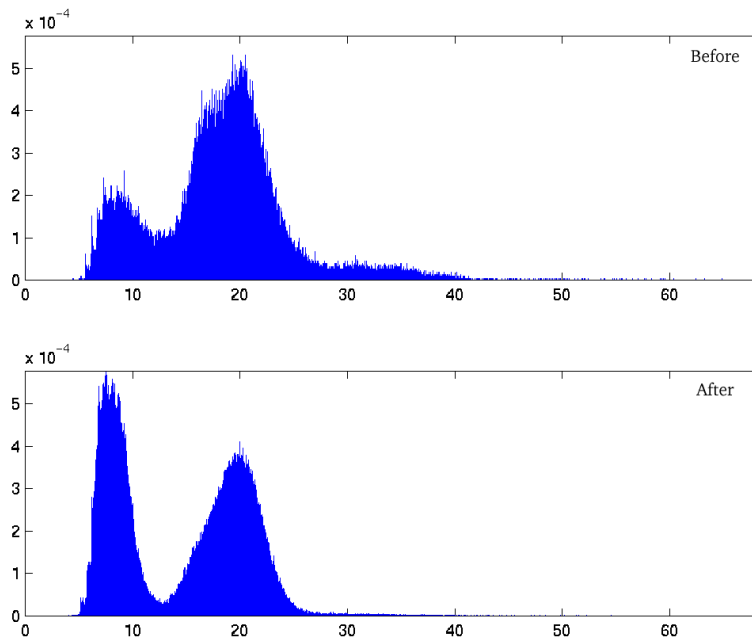


Figure 5.1: Interrupt latency distribution, before and after the modifications.

Table 5.1 shows a small improvement in mean and median values. The measured maximum value is slightly higher in the modified version, but still much smaller than for Linux. It should be noted that many more samples were taken in the last test and it was therefore more likely that a higher maximum value should be discovered.

Table 5.1: Interrupt latency with load

	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
Linux	5.2	162.9	14.3	11.3	6.4	476052
RTAI before changes	4.4	64.9	17.9	18.5	5.7	223559
RTAI after changes	4.1	68.5	14.3	15.6	6.0	918598

5.1.2 Scheduling Precision

The major difference between these results and those presented earlier in section 4.2.4 is that the load script could be applied without breaking the system. The values are otherwise very similar to those obtained earlier.

Table 5.2: Scheduling jitter

<i>Period</i>	<i>Min</i>	<i>Max</i>	<i>Mean</i>	<i>Median</i>	<i>Std. Dev.</i>	<i>#Values</i>
1 ms, no load	0.00	24.81	1.06	0.17	1.95	425893
1 ms, load	0.00	74.14	3.17	2.31	3.74	454880

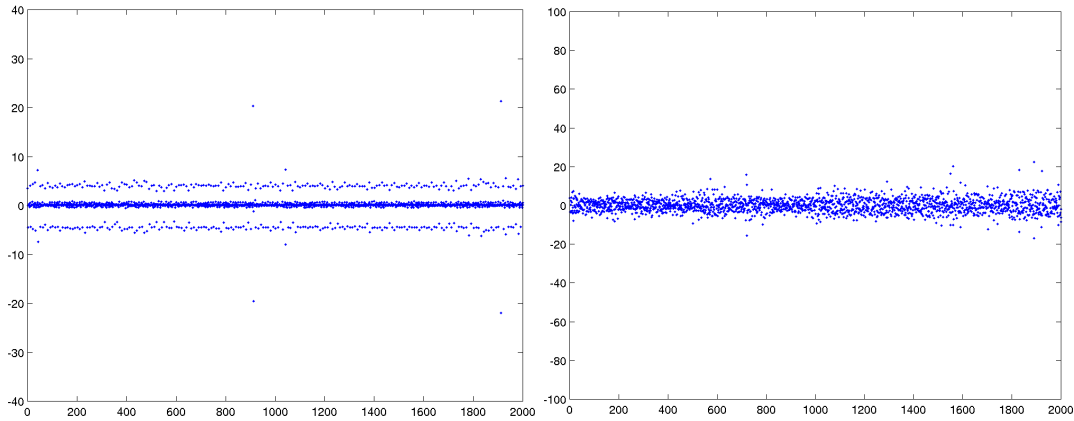


Figure 5.2: 1 ms without load and with load

Chapter 6

Conclusion

The evaluation leads to three primary conclusions:

- The basic functionality of RTAI is working on ETRAX.
- The interrupt latency in RTAI is much more limited than in Linux. The measurement shows a 2.5 times higher worst case interrupt latency in Linux compared to RTAI.
- All measurements indicate a good average real-time performance in RTAI.

To sum up, it can be said that RTAI is a solution that provides good real-time performance, while preserving the entire functionality of Linux.

6.1 Discussion

Porting a system such as RTAI is not easy. The documentation available at this level for both RTAI and Linux is very limited. In order to fully understand, one can not rely on written documentation, but must instead read the code itself. This is a time consuming task. The process of debugging the kernel in an embedded system is also very time consuming and the addition of RTAI does not make it any easier.

The functionality of the RTAI system was tested only through the performance tests. While basic functionality, such as installing handlers and running periodic tasks are tested, a lot of testing remains in order to verify the functionality of the entire system.

In order to make RTAI a commercially attractive solution for the ETRAX platform, some of the limitations mentioned in sections 3.4 and 3.5 could need further attention. The stability problem, although much smaller now

as described in Chapter 5, would for example have to be solved.

When one compares real-time performance, RTAI is much better than Linux as can be seen in Chapters 4 and 5. The improved performance is visible in all of the tests. For example, in the interrupt latency test where the plain Linux kernel had a maximum response time of 163 μs , RTAI never went over 68 μs .

Although many samples were taken during the evaluation, there is no guarantee that the measured worst cases correspond to the actual worst cases of the system. However, the measured values should not be far off. Guaranties, as required by hard real-time, require a full code-path analysis. Such an analysis is possible in RTAI because of its limited size and operations, but is beyond the scope of this master thesis.

6.2 Summary

This thesis started with a brief overview of real-time systems in Chapter 2. An investigation into how real-time can be achieved on Linux was made. It was found that two major approaches existed; One that improved the standard Linux kernel in some way and one that added a small real-time kernel in combination with a hardware abstraction layer. It was concluded that the first approach is good because it improves Linux without the users having to modify their applications. However, it can not offer any guaranties required for hard real-time. The second approach makes it necessary to split applications into one real-time part and one user-space part, but the approach can deliver hard real-time.

As said in the problem description for this thesis, hard real-time extensions should be investigated and therefore only projects based on the second approach were considered. It was found that two such projects existed, RTLinux and RTAI. They were both evaluated and it was found that, although they provide essentially the same performance, RTAI is more actively developed and is open-source. Hence, RTAI was chosen as the extension to port to ETRAX.

The implementation of RTAI on ETRAX was described in Chapter 3. It started with the hardware abstraction layer. In order to introduce the layer into the Linux kernel, the interrupt paths were studied in detail and an investigation of how the paths could be safely modified by a kernel patch was made. Some function calls were substituted and others added. The important factors of not disabling interrupts when it is not absolutely necessary and not running unpredictable code during this time, were considered.

For the real-time scheduler to be precise, it was necessary to configure the two hardware timers in cascaded mode. This yielded a granularity of 0.16 μ s, which compared to the default configuration with an accuracy of 40 μ s is a big improvement.

The MMU bus fault is not handled by RTAI on ETRAX and it was concluded that a further investigation can be made into whether it is possible to handle the MMU bus fault, or parts of it, in an easy way.

A thorough evaluation was made in Chapter 4. A number of tests were designed and test programs written. Real-time performance such as interrupt latency and scheduling jitter was measured and also a basic functionality verification was achieved through the tests. The results concluded that RTAI provides good real-time performance, especially compared to standard Linux.

Chapter 5, which was written after the evaluation was made, described changes to the implementation. Besides better performance, the stability of the implementation improved significantly.

Bibliography

- [1] John A. Stankovic, et al. *What is predictability for Real-Time Systems?* Editorial, Dept. of Computer and Information Science, University of Massachusetts, 1993.
- [2] Kevin Dankwardt. *Real-Time and Linux, Part 1*. Embedded Linux Journal, January 2002.
- [3] Kevin Dankwardt. *Real-Time and Linux, Part 2: the Preemptible Kernel*. Embedded Linux Journal, March 2002.
- [4] Tim Bird. *Comparing two approaches to real-time Linux*. Guest column at Linuxdevices.com, Dec 2000. Available at <http://www.linuxdevices.com/articles/AT7005360270.html>.
- [5] REDSonic, Inc. homepage. Available at <http://www.redsonic.com/>.
- [6] MontaVista Software, Inc. homepage. Available at <http://www.mvista.com/>.
- [7] TimeSys(TM) homepage. Available at <http://www.timesys.com/>.
- [8] The DIAPM RTAI project homepage. Available at <http://www.rtai.org/>.
- [9] Paolo Mantegazza. RTAI History. Available at <http://www.aero.polimi.it/~rtai/documentation/articles/history/>.
- [10] Paolo Mantegazza. *Dissecting DIAPM RTHAL-RTAI*. Available at <http://www.aero.polimi.it/~rtai/documentation/articles/paolo-dissecting.html>
- [11] Rick Lehrbaum and Kevin Dankwardt. *RTAI goes (partly) GPL*. Linuxdevices.com, March 2002. Available at <http://www.linuxdevices.com/articles/AT2899063844.html>.
- [12] Eben Moglen. About the RT-Linux patent and RTAI. Available at <http://www.aero.polimi.it/~rtai/documentation/articles/moglen.html>.

- [13] FSMLabs, Inc. homepage. Available at <http://www.fsmlabs.com/>.
- [14] Victor Yodaiken and Michael Barabanov. *A Real-Time Linux*. New Mexico Institute of Technology.
- [15] Michael Barabanov. *A Linux-based Real-Time Operating System*. Master's thesis, New Mexico Institute of Mining and Technology, June 1997.
- [16] FSMLabs, Inc. *Getting started with RTLinux*. April 2001. Available at <http://www.fsmlabs.com/developers/docs/pdf/GettingStarted.pdf>
- [17] FSMLabs, Inc. *The RTLinuxCompany Design Whitepaper*. April 2001. Available at http://www.fsmlabs.com/developers/white_papers/design.htm
- [18] The Open RTLinux Patent License, version 2.0. Available at <http://www.fsmlabs.com/about/patent/openpatentlicense.htm>.
- [19] Ismael Ripoll. *RTLinux versus RTAI*. 2002. Available at http://bernia.disca.upv.es/rtportal/comparative/rtl_vs_rtai.html
- [20] Arno Leucht. *Hard real-time capability under Linux – development within the micro-kernel real-time architecture*. February 21, 2002. Available at <http://www.electronicengineering.com/features/embedded/OEG20020220S0009>

All information from the web pages was collected in August 2002.

Appendix A

Test Environment

The RTAI version used is 24.1.9 and it is configured to use the uniprocessor scheduler with the default scheduling algorithm. The tests in Chapter 4 are performed on revision R1_0_0 of the implementation on CRIS. It can be obtained from the CVS repository at Axis upon request. The Linux kernel version used is revision R2_4_20_021202 and the compilation tools are those shipped in `cris-dist_1.24-1_i386.deb`, also available from Axis.

The following hardware has been used during the tests:

- PC1: Standard PC (Pentium III 600MHz, 256MB SDRAM, 100Mbit Ethernet adapter) running Debian GNU/Linux 3.0.
- PC2: Standard PC (Pentium III 650MHz, 256MB SDRAM, 100Mbit Ethernet adapter) running Debian GNU/Linux 3.0.
- An ETRAX 100LX-based developer board with 8MB RAM running Linux.
- Tektronix TLA 715 Logic Analyzer.

Some tests are conducted under load. By load we mean:

- A PC sends a flood ping to the developer board (`ping -f`).
- A script is running in a loop on the developer board, reading files in the `/proc`-directory.

The developer board is disconnected from the network when a test is run without load. When load is applied, all three computers (PC1, PC2 and the developer board) are connected to a 100 Mbps Ethernet switch (from PLANET Technology Corp, model SW-800). This little network is isolated during the tests.

Appendix B

Test Programs

B.1 Interrupt Latency

B.1.1 RT-Handler

```
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <asm/io.h>
#include <asm/sv_addr.agh>
#include <asm/rtai.h>
#include "test.h"

/* The real-time interrupt handler */
void handler(void)
{
    set_response_pin1_high();

    // Wait for PC to take "down" the pin
    while ((*R_PORT_PA_READ&0x80)==0x80);

    set_response_pin1_low();
}

static __init int init_testhandler(void)
{
    // Setup parallel port
    initialize_port();
    set_response_pin1_low();
    set_response_pin2_low();

    // Make sure the interrupt is masked at first
    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    // Set General Port PA to input direction, pin 7
    // This is a write-only register!!! Must use shadow-reg!
    REG_SHADOW_SET(R_PORT_PA_DIR, port_pa_dir_shadow, 7, 0);

    // Mount (init & mount) rtai
    rt_mount_rtai();

    // Install and enable irq-handling
    rt_startup_irq(PA_IRQ);
    rt_enable_irq(PA_IRQ);
}
```

```

        // Request rt-irq
        rt_request_global_irq(PA_IRQ, handler);

        // Enable the external interrupt on General Port PA
        *R_IRQ_MASK1_SET = IO_STATE(R_IRQ_MASK1_SET, pa7, set);

        printk("*** intlat installed on irq %d ***\n", PA_IRQ);
        return 0;
    }

static __exit void cleanup_testhandler(void)
{
    // Disable the external interrupt on General Port PA
    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    // Disable and remove irq-handling
    rt_disable_irq(PA_IRQ);
    rt_shutdown_irq(PA_IRQ);

    set_response_pin1_low();
    set_response_pin2_low();

    // Release the rt-irq
    rt_free_global_irq(PA_IRQ);

    rt_umount_rtai();

    printk("*** intlat unloaded ***\n");
}

module_init(init_testhandler);
module_exit(cleanup_testhandler);

```

B.1.2 Linux-Handler

```

#include <linux/module.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <asm/io.h>
#include <asm/sv_addr.agh>
#include <linux/ptrace.h>
#include "test.h"

/* The interrupt handler */
void handler(int irq, void* dev_id, struct pt_regs* regs)
{
    set_response_pin1_high();

    // Wait for PC to take "down" the pin
    while ((*R_PORT_PA_READ&0x80)==0x80);

    set_response_pin1_low();
}

static __init int init_testhandler(void)
{
    // Setup parallel port
    initialize_port();
    set_response_pin1_low();
    set_response_pin2_low();

    // Make sure the interrupt is masked at first

```

```

    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    // Set General Port PA to input direction, pin 7
    REG_SHADOW_SET(R_PORT_PA_DIR, port_pa_dir_shadow, 7, 0);

    // Install and enable irq-handling
    if (request_irq(PA_IRQ, handler, SA_INTERRUPT, "testhandler", NULL)) {
        printk("Error: intlnt_linux could not grab the irq!\n");
        return 1;
    }

    // Unmask the external interrupt on General Port PA
    *R_IRQ_MASK1_SET = IO_STATE(R_IRQ_MASK1_SET, pa7, set);

    // Mount RTAI (in some test cases)
    //rt_mount_rtai();

    printk("*** intlnt_linux installed on irq %d ***\n", PA_IRQ);
    return 0;
}

static __exit void cleanup_testhandler(void)
{
    // Umount RTAI
    //rt_umount_rtai();

    // Disable the external interrupt on General Port PA
    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    set_response_pin1_low();
    set_response_pin2_low();

    // Release irq
    free_irq(PA_IRQ, NULL);

    printk("*** intlnt_linux unloaded ***\n");
}

module_init(init_testhandler);
module_exit(cleanup_testhandler);

```

B.2 Interrupt Task Latency

B.2.1 RT-Task

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtai_types.h>
#include <rtai_sched.h>
#include "test.h"

// Set to 0 if suspend/resume is wanted instead of wait/signal
#define USE_SEMAPHORE 1
#define STACK_SIZE 500
#define HIGH_PRIO 0

#if USE_SEMAPHORE
static SEM sem;
#endif

static RT_TASK high_prio_task;

```

```

/* High prio task function */
static void high_prio_fun(int data)
{
    while(1) {
        set_response_pin2_low();
#ifdef USE_SEMAPHORE
        rt_sem_wait(&sem);
#else
        rt_task_suspend(&high_prio_task);
#endif
    }
}

/* Interrupt handler */
void handler_fun(void)
{
    set_response_pin1_high();
    set_response_pin2_high();
#ifdef USE_SEMAPHORE
    rt_sem_signal(&sem);
#else
    rt_task_resume(&high_prio_task);
#endif
    set_response_pin1_low();
}

int init_module(void)
{
#ifdef USE_SEMAPHORE
    RTIME start;
#endif
    // Setup parallel port
    initialize_port();
    set_response_pin1_low();
    set_response_pin2_low();

    // Make sure the interrupt is masked at first
    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    // Set General Port PA to input direction, pin 7
    REG_SHADOW_SET(R_PORT_PA_DIR, port_pa_dir_shadow, 7, 0);

    rt_mount_rtai();

    // Install and enable irq-handling
    rt_startup_irq(PA_IRQ);
    rt_enable_irq(PA_IRQ);

    // Request rt-irq
    rt_request_global_irq(PA_IRQ, handler_fun);

    // Initialize the task
    rt_task_init(&high_prio_task, high_prio_fun, 0, STACK_SIZE,
                HIGH_PRIO, 0, 0);
#ifdef USE_SEMAPHORE
    rt_sem_init(&sem, 0);
    start = rdtsc() + nano2count(1000000000); // 1s from now
    rt_task_make_periodic(&high_prio_task, start, nano2count(1000000000));
    start_rt_timer(0); // Start with default value
#endif
    // Enable the external interrupt on General Port PA
    *R_IRQ_MASK1_SET = IO_STATE(R_IRQ_MASK1_SET, pa7, set);
    return 0;
}

```

```

void cleanup_module(void)
{
    // Disable the external interrupt on General Port PA
    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    // Disable and remove irq-handling
    rt_disable_irq(PA_IRQ);
    rt_shutdown_irq(PA_IRQ);

    // Release the rt-irq
    rt_free_global_irq(PA_IRQ);

    set_response_pin1_low();
    set_response_pin2_low();

    rt_task_delete(&high_prio_task);

#ifdef USE_SEMAPHORE
    stop_rt_timer();
    rt_sem_delete(&sem);
#endif
    rt_umount_rtai();
}

```

B.2.2 Kernel-Thread

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/io.h>
#include <asm/smplock.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <asm/io.h>
#include <linux/ptrace.h>
#include "test.h"

static int kthread_shutdown = 0;
DECLARE_WAIT_QUEUE_HEAD(kthread_wait_queue_interrupt);
DECLARE_WAIT_QUEUE_HEAD(kthread_wait_queue_shutdown);

/* High prio task function */
static int high_prio_fun(void * whatever)
{
    // Detach from the original process
    sprintf(current->comm, "High prio fun");
    lock_kernel();
    exit_mm(current);

    // While the module is not unloaded
    while(!kthread_shutdown) {
        set_response_pin2_low();
        interruptible_sleep_on(&kthread_wait_queue_interrupt);
    }

    return 0;
}

/* Interrupt handler */
static void handler_fun(int irq, void* dev_id, struct pt_regs* regs)
{
    set_response_pin1_high();

    set_response_pin2_high();
}

```

```

        wake_up_interruptible(&kthread_wait_queue_interrupt);

        set_response_pin1_low();
    }

int init_module(void)
{
    // Setup parallel port
    initialize_port();
    set_response_pin1_low();
    set_response_pin2_low();

    // Make sure the interrupt is masked at first
    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    // Set General Port PA to input direction, pin 7
    REG_SHADOW_SET(R_PORT_PA_DIR, port_pa_dir_shadow, 7, 0);

    // Install and enable irq-handling
    if (request_irq(PA_IRQ, handler_fun, SA_INTERRUPT, "handler", NULL)) {
        printk("Error: inttasklat_linux could not grab the irq!\n");
        return 1;
    }

    // Enable the external interrupt on General Port PA
    *R_IRQ_MASK1_SET = IO_STATE(R_IRQ_MASK1_SET, pa7, set);

    // Fork the main thread
    kernel_thread(high_prio_fun, NULL, 0);

    return 0;
}

void cleanup_module(void)
{
    // Tell the kernel thread to stop
    kthread_shutdown = 1;

    // Simulate an interrupt to awaken the kernel thread.
    wake_up_interruptible(&kthread_wait_queue_interrupt);

    // Disable the external interrupt on General Port PA
    *R_IRQ_MASK1_CLR = IO_STATE(R_IRQ_MASK1_CLR, pa7, clr);

    set_response_pin1_low();
    set_response_pin2_low();

    // Release irq
    free_irq(PA_IRQ, NULL);
}

```

B.3 Scheduling Latency

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <rtai.h>
#include <rtai_types.h>
#include <rtai_sched.h>
#include "test.h"

#define STACK_SIZE 500 // Task stack size
#define HIGH_PRIO 0
#define LOW_PRIO 1

```

```

#define HIGH_PRIO_PERIOD 1000000000 // 1s, not interesting in this case.
#define LOW_PRIO_PERIOD 200000000 // 0.2s, time between reschedules.

// Define if interrupts should be disabled during tests.
// #define NO_INTERRUPTS

// Set to 0 if suspend/resume is wanted instead of wait/signal
#define USE_SEMAPHORE 1

static RT_TASK high_prio_task;
static RT_TASK low_prio_task;

#if USE_SEMAPHORE
static SEM sem;
#endif

/* High prio task function */
static void high_prio_fun(int data)
{
    while(1) {
        set_response_pin1_low();
#if USE_SEMAPHORE
        rt_sem_wait(&sem);
#else
        rt_task_suspend(0);
#endif
    }
}

/* Low prio task function */
static void low_prio_fun(int data)
{
    while(1) {
        set_response_pin1_high();
#if USE_SEMAPHORE
        rt_sem_signal(&sem);
#else
        rt_task_resume(&high_prio_task);
#endif
        rt_task_wait_period();
    }
}

int init_module(void)
{
    RTIME now, high_prio_start, low_prio_start;

    // Setup parallel port
    initialize_port();
    set_response_pin1_low();

    rt_mount_rtai();

    // Initialize tasks
    rt_task_init(&high_prio_task, high_prio_fun, 0, STACK_SIZE,
                HIGH_PRIO, 0, 0);
    rt_task_init(&low_prio_task, low_prio_fun, 0, STACK_SIZE,
                LOW_PRIO, 0, 0);

#if USE_SEMAPHORE
    rt_sem_init(&sem, 0);
#endif

    // Start timer with default value
    start_rt_timer(0);
}

```

```

        now = rdtsc();
        high_prio_start = now + nano2count(1000000000); // 1s from now.
        low_prio_start = now + nano2count(2000000000); // 2s from now.
        rt_task_make_periodic(&high_prio_task, high_prio_start,
                               nano2count(HIGH_PRIO_PERIOD));
        rt_task_make_periodic(&low_prio_task, low_prio_start,
                               nano2count(LOW_PRIO_PERIOD));

        return 0;
}

void cleanup_module(void)
{
    stop_rt_timer();

    rt_task_delete(&high_prio_task);
    rt_task_delete(&low_prio_task);

#ifdef USE_SEMAPHORE
    rt_sem_delete(&sem);
#endif

    rt_umount_rtai();
}

```

B.4 Scheduling Precision

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <rtai.h>
#include <rtai_types.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include "test.h"

#define STACK_SIZE 2000 // Task stack size
#define HIGH_PRIO 0
#define LOW_PRIO 1
// #define TIMER_PERIOD 1000000 // 1.0 ms
// #define TASK_PERIOD 1000000 // 1.0 ms
#define TIMER_PERIOD 500000 // 500us
#define TASK_PERIOD 500000 // 500us

// Define if lower priority tasks should run concurrently
#define LOAD

static RT_TASK high_prio_task;
#ifdef LOAD
static RT_TASK low_prio_task_1;
static RT_TASK low_prio_task_2;
#endif

/* High prio task function */
static void high_prio_fun(int whatever)
{
    while (1) {
        toggle_response_pin1();
        rt_task_wait_period();
    }
}

#ifdef LOAD
/* Low prio task function */

```



```

static void low_prio_fun(int whatever)
{
    while (1) {
        rt_task_wait_period();
    }
}
#endif

int init_module(void)
{
    RTIME now, start_1, start_2;

    // Setup parallel port
    initialize_port();
    set_response_pin1_low();
    set_response_pin2_low();

    rt_mount_rtai();

    // Initialize tasks
    rt_task_init(&high_prio_task, high_prio_fun, 0, STACK_SIZE,
                HIGH_PRIO, 0, 0);
#ifdef LOAD
    rt_task_init(&low_prio_task_1, low_prio_fun, 0, STACK_SIZE,
                LOW_PRIO, 0, 0);
    rt_task_init(&low_prio_task_2, low_prio_fun, 0, STACK_SIZE,
                LOW_PRIO, 0, 0);
    // Let the tasks be preemptive even in oneshot mode
    rt_preempt_always(1);
#endif

    start_rt_timer(nano2count(TIMER_PERIOD));

    // Start the task
    now = rdtsc();
    start_1 = now + nano2count(1000000000);
    start_2 = now + nano2count(1000000000) - nano2count(200000);

    rt_task_make_periodic(&high_prio_task, start_1,
                          nano2count(TASK_PERIOD));
#ifdef LOAD
    rt_task_make_periodic(&low_prio_task_1, start_2,
                          nano2count(TASK_PERIOD));
    rt_task_make_periodic(&low_prio_task_2, start_1,
                          nano2count(TASK_PERIOD));
#endif
    return 0;
}

void cleanup_module(void)
{
    stop_rt_timer();

    rt_task_delete(&high_prio_task);
#ifdef LOAD
    rt_task_delete(&low_prio_task_1);
    rt_task_delete(&low_prio_task_2);
#endif
    rt_umount_rtai();
}

```

B.5 Communication Overhead

B.5.1 Between RT-Tasks

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <rtai.h>
#include <rtai_types.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include "test.h"

// Message size of 4 or 1024 bytes.
// #define FIFO_SIZE 4
// #define MESSAGE_SIZE 4
#define FIFO_SIZE 1024
#define MESSAGE_SIZE 1024

#define STACK_SIZE 500 // Task stack size
#define HIGH_PRIO 0
#define FIFO_NBR 0
#define HIGH_PRIO_PERIOD 100000000 // 0.1s, time between messages

static RT_TASK high_prio_task;
static int data[FIFO_SIZE];

/* High prio task function */
static void high_prio_fun(int whatever)
{
    while (1) {
        unsigned long flags;
        flags = rt_global_save_flags_and_cli();

        // Write data
        set_response_pin1_high();
        rtf_put (FIFO_NBR, data, MESSAGE_SIZE);
        set_response_pin1_low();

        // Read data
        set_response_pin2_high();
        rtf_get (FIFO_NBR, data, MESSAGE_SIZE);
        set_response_pin2_low();

        rt_global_restore_flags(flags);

        rt_task_wait_period();
    }
}

int init_module(void)
{
    RTIME now, high_prio_start;

    // Setup parallel port
    initialize_port();
    set_response_pin1_low();
    set_response_pin2_low();

    rt_mount_rtai();

    rtf_create(FIFO_NBR, FIFO_SIZE);

    // Initialize task
    rt_task_init(&high_prio_task, high_prio_fun, 0, STACK_SIZE,
```

```

        HIGH_PRIO, 0, 0);

    rt_set_oneshot_mode();
    start_rt_timer(0);

    now = rdtsc();
    high_prio_start = now + nano2count(1000000000); // 1s from now.

    rt_task_make_periodic(&high_prio_task, high_prio_start,
                        nano2count(HIGH_PRIO_PERIOD));
    return 0;
}

void cleanup_module(void)
{
    stop_rt_timer();

    rtf_destroy (FIFO_NBR);

    rt_task_delete(&high_prio_task);

    rt_umount_rtai();
}

```

B.5.2 RT-Task to User-Space

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <rtai.h>
#include <rtai_types.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include "test.h"
#include "test_fifo_complete.h"

#define STACK_SIZE 500           // Task stack size
#define HIGH_PRIO 0             // Task priority
#define HIGH_PRIO_PERIOD 200000000 // 0.2s, time between messages

static RT_TASK high_prio_task;

static void high_prio_fun(int whatever)
{
    while (1) {
        // Write data
        set_response_pin1_high();
        rtf_put (FIFO_OUT, data, MESSAGE_SIZE);
        rt_task_wait_period();
    }
}

int fifo_read_handler(unsigned int fifo)
{
    // Read data
    rtf_get (FIFO_IN, data, MESSAGE_SIZE);
    set_response_pin1_low();
    return 0;
}

int init_module(void)
{
    RTIME now, high_prio_start;

    // Setup parallel port

```

```

        initialize_port();
        set_response_pin1_low();
        set_response_pin2_low();

        // Create FIFOs
        rtf_create (FIFO_OUT, FIFO_SIZE);
        rtf_create (FIFO_IN, FIFO_SIZE);

        // Start and mount rtai
        rt_mount_rtai();

        rtf_create_handler(FIFO_IN, fifo_read_handler);

        // Initialize task
        rt_task_init(&high_prio_task, high_prio_fun, 0, STACK_SIZE, HIGH_PRIO, 0, 0);

        // Start timer
        rt_set_oneshot_mode();
        start_rt_timer(0);

        // Start the task
        now = rdtsc();
        high_prio_start = now + nano2count(1000000000); // Start in 1s.
        rt_task_make_periodic(&high_prio_task, high_prio_start,
                             nano2count(HIGH_PRIO_PERIOD));

        return 0;
}

void cleanup_module(void)
{
    rtf_destroy (FIFO_OUT);
    rtf_destroy (FIFO_IN);

    stop_rt_timer();

    rt_task_delete(&high_prio_task);

    rt_umount_rtai();
}

```

B.5.3 User-Space to RT-Task

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <rtai_fifos.h>
#include "test_fifo_complete.h"

static int end;
static void endme(int dummy) { end = 1;}

int main(int argc, char *argv[])
{
    int fd0;
    int fd1;

```

```

    signal(SIGINT, endme);

    // Input
    if ((fd0 = open("/dev/rtf0", O_RDONLY)) < 0) {
        fprintf(stderr, "Error opening /dev/rtf0\n");
        exit(1);
    }

    // Output
    if ((fd1 = open("/dev/rtf1", O_WRONLY)) < 0) {
        fprintf(stderr, "Error opening /dev/rtf1\n");
        exit(1);
    }

    while (!end) {
        read(fd0, &data, MESSAGE_SIZE);
        write(fd1, data, MESSAGE_SIZE);
    }

    close(fd0);
    close(fd1);

    return 0;
}

```

B.6 Load Program

```

#!/bin/sh
cd /proc
while true
do
    cat cmdline
    cat cpuinfo
    cat devices
    cat dma
    cat execdomains
    cat fasttimer
    cat filesystems
    cat iomem
    cat ioports
    cat ksyms
    cat loadavg
    cat locks
    cat meminfo
    cat misc
    cat modules
    cat mtd
    cat partitions
    cat slabinfo
    cat stat
    cat swaps
    cat uptime
    cat version
done

```

B.7 Interrupt generator

/* This program is executed on PC1 in the interrupt latency and interrupt task latency tests. It generates a signal on the parallel port within a random time interval. */

```

#include <module.h> // for versions etc.
#include <asm/io.h> // for inb and outb

```

```

#define DEV_MAJOR 126
#define BASE_OUT 0x378
#define BASE_IN 0x379

/* The longest loop count a spike may take. If
   this limit is reached the generator stops */
#define SAFETY_SPIKE_LIMIT 100000UL

/* The time interval (in seconds) of the delay between spikes */
#define DELAY_LOW 0.2
#define DELAY_HIGH 0.5

/* Global flag indicating run-condition */
volatile unsigned int running = 0;

static void start_spikes(void);
static void stop_spikes(void);
static void do_spikes(void);

/*
 * NOTE: The below implementation of pseudo-random number
 * generator is not good, but it is fast and simple and
 * should suffice in our application.
 */
// -----
#define RAND_MAX 32767

unsigned long int next = 1;

int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
// -----

/* Handles writes on the device registered with this module */
static ssize_t handle_dev_write(struct file *file, const char *buf,
                                size_t count, loff_t *off)
{
    if (count>1 && buf[0]=='G' && buf[1]=='0') {
        start_spikes();
    } else if (count>1 && buf[0]=='N' && buf[1]=='0') {
        stop_spikes();
    } else {
        printk("spike: Unknown command received.\n");
    }
    return count;
}

/* Device operations */
static struct file_operations dev_fops = {
    write:    handle_dev_write,
};

/* Generates spikes periodically as long as "running" is true */
static void do_spikes(void)
{
    unsigned long j;
    unsigned long flags;
    unsigned long safety;
    unsigned int counter = 0;
    char curr;

```

```

float delay_interval = DELAY_HIGH - DELAY_LOW;

while (running) {
    // Read current state of cris response pin
    curr = inb(BASE_IN)&0x40;

    // Disable interrupts
    __save_flags(flags);
    __cli();

    // Set data low
    outb(0, BASE_OUT);

    // Wait until dev.board changes the response pin
    safety = 0;
    while ((inb(BASE_IN)&0x40)==curr && ++safety<SAFETY_SPIKE_LIMIT);

    // Safety check in case of RTAI failure
    if (safety>=SAFETY_SPIKE_LIMIT) {
        printk("spike: *** WARNING: SAFETY LIMIT REACHED ***\n");
        stop_spikes();
    }

    // Set data high
    outb(255, BASE_OUT);

    // Restore interrupts
    __restore_flags(flags);

    // Increase counter
    ++counter;
    if (counter%10==0) {
        printk("spike: %d spikes has been generated.\n", counter);
    }

    // Wait delay between spikes
    j = jiffies + ((rand()/(RAND_MAX+1.0))*delay_interval + DELAY_LOW)*HZ;
    while (jiffies < j)
        schedule();
}

/* Starts generating spikes if they are not already started */
static void start_spikes(void) {
    if (!running) {
        running = 1;
        printk("spike: +++ Starting spikes +++\n");
        do_spikes();
    }
}

/* Stops generating spikes */
static void stop_spikes(void) {
    running = 0;
    printk("spike: --- Stopping spikes ---\n");
}

/* Initializes the module */
static __init int init_parapc(void)
{
    // Register device
    if (register_chrdev(DEV_MAJOR, "spike", &dev_fops)<0) {
        printk("Unable to get major %d for spike\n", DEV_MAJOR);
        return 1;
    }
}

```

```
    }

    // Set data high
    outb(255, BASE_OUT);

    // Not running yet
    running = 0;
    printk("SPIKE GENERATOR module inserted\n");
    return 0;
}

/* Clean-up of the module */
static __exit void cleanup_parapc(void)
{
    // Unregister device
    unregister_chrdev(DEV_MAJOR, "spike");

    // Set data high
    outb(255, BASE_OUT);
    printk("SPIKE GENERATOR module removed\n");
}

module_init(init_parapc);
module_exit(cleanup_parapc);
```


Appendix C

Modified Kernel Files

This appendix describes files modified by the kernel patch and briefly how they are modified. This is not intended to be an in-depth description explaining all details; for that kind of information we direct the interested reader to the patch itself.

linux/Makefile

Only changed when linking the `rtai.o`-module directly into the kernel. This is currently the case as described in 3.5.

linux/arch/cris/config.in

linux/Documentation/Configure.help

linux/arch/cris/defconfig

Adds a few configuration alternatives for enabling or disabling RTHAL in the kernel. Also an option for turning off the cascaded timer mode used in RTAI is added.

linux/include/asm-cris/irq.h

Adds RTHAL to the low-level `IRQxx_interrupt`-routines as described in section 3.1.2. This is done by modifying the macros that create the routines.

linux/arch/cris/kernel/entry.S

Adds RTHAL to the interrupt paths as described in section 3.1.2.

linux/arch/cris/kernel/ksyms.c

Adds some exported symbols that the RTAI modules need.

linux/arch/cris/kernel/irq.c

Initializes the `rthal`-struct. The interrupt related calls will go through the struct after this initialization, but the functions that are installed will perform as usual until RTAI is mounted.

Also, some functions for handling the interrupt vector are made available.

linux/arch/cris/kernel/time.c

Adds a timer interrupt handler that is used when pending the timer interrupt. This is to speed things up because some of the things done in the normal timer handler is now done in RTAI before the interrupt is pending to Linux.

linux/arch/cris/mm/fault.c

Changes a `sti()`-call in the `do_page_fault`-routine into a `hard_sti()` that really enables the interrupts on the hardware level.

linux/include/asm-cris/system.h

The `rthal`-struct is defined in this file and it looks like this:

```
struct rt_hal {
    void (*do_IRQ) (int, struct pt_regs*);      /* 0 */
    void (*do_timer_IRQ)(int, struct pt_regs*); /* 4 */
    long long (*do_SRQ) (int, unsigned long);   /* 8 */
    void (*disint)(void);                       /* 12 */
    void (*enint)(void);                       /* 16 */
    unsigned long (*getflags)(void);            /* 20 */
    void (*setflags)(unsigned long);            /* 24 */
    unsigned long (*getflags_and_cli)(void);    /* 28 */
    void (*ei_if_rtai)(void);                   /* 32 */
    void (*unmask_if_not_rtai)(unsigned int);   /* 36 */
};
```

Also all macros that are related to interrupts are redefined so that they go through the `rthal`-struct.

linux/arch/cris/mm/ioremap.c**linux/mm/vmalloc.c****linux/include/asm-cris/pgalloc.h**

The modifications to these files are only needed when the MMU bus fault is handled by RTAI and when `rtai.o` is used as a module. Since neither of this

is the case presently, the files are not patched.

The patch propagates the modifications of the kernel memory map to all existing processes in the system. Kernel mappings, as created by `vmalloc()` and the like, modify the kernel memory map only. Usually, existing processes don't receive this mapping. Should they ever need it, it is added to their map later upon a page-fault.

Should RTAI install a new interrupt dispatcher even for the `do_page_fault` part of the MMU bus fault, as mentioned in section 3.4, it is necessary to propagate the kernel memory map modifications to all processes that existed before RTAI was loaded. Otherwise the system would run in a double page fault: Some page is not there -> page fault -> page fault and so on.

Appendix D

Kernel Modules

Table D.1 describes essential kernel modules in RTAI and their respective file sizes and memory footprints. The figures are in bytes and the memory footprint is as reported by `lsmod`.

Note that the main module is currently not used as a module, but instead linked directly into the kernel as said in 3.5.

Module	File Size	Memory Footprint	Comments
<code>rtai.o</code>	13933	5528	The main module
<code>rtai.o</code>	15276	14664	The main module linked with <code>rt_printk</code> ¹
<code>rtai_sched_up.o</code>	56779	41596	The real-time uniprocessor scheduler linked with <code>rt_mem_mgr</code> ²
<code>rtai_fifos.o</code>	29328	17464	Adds FIFO queue support for communication between Linux and RTAI
<code>rtai_pqueue.o</code>	22186	17216	Adds support for POSIX pQueues
<code>rtai_pthread.o</code>	51607	47808	Adds support for POSIX pThreads
<code>rtai_utils.o</code>	1451	452	Some utilities needed by <code>rtai_pthread.o</code>

Table D.1: Kernel Modules

¹This is a service for doing console printouts without affecting real-time performance.

²Real-Time Memory Manager. This service tries to implement a somewhat real-time safe memory allocation. It can be excluded and then the standard `kmalloc` is used instead.

Appendix E

Glossary

ARM	A processor architecture. RTAI has been ported to this architecture.
CRIS	Code Reduced Instruction Set, the CPU architecture designed by Axis and used in its ETRAX processors.
Deadline	The longest acceptable response-time for a task in an application.
ETRAX	A family of processors developed by Axis and based on CRIS. The processor is designed with networking and embedded systems in mind.
Cycle Counter	An internal counter that increments with the clock frequency of the CPU.
DIAPM	Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano. The place of origin for RTAI.
FPU	Floating Point Unit
GPL	GNU General Public License, see http://www.gnu.org/copyleft/gpl.html .
Hard Real-time	In a hard real-time system, all the deadlines of the system must be met at all times.
Kernel-Space	Most device drivers and the kernel itself run in kernel-space. Code running in kernel-space does not have memory protection, access restrictions <i>etc.</i> In short it has access to whatever it wants. See also <i>user-space</i> .
LGPL	GNU Lesser General Public License, see http://www.gnu.org/copyleft/lesser.html .

LXRT	RTAI service that provides support for hard real-time from Linux user-space.
MIPS	A processor architecture. RTAI has been ported to this architecture.
MMU	Memory Management Unit. A vital part of the processor providing memory protection, address translation <i>etc.</i>
PPC	PowerPC, a processor architecture. RTAI has been ported to this architecture.
POSIX	Portable Operating System Interface for UNIX.
Response Time	The time interval from when an application receives a stimulus to when the application has produced a result based on that stimulus.
RPC	Remote Procedure Call.
RTAI	Real-Time Application Interface.
RTHAL	Real-Time Hardware Abstraction Layer.
RTOS	Real-Time Operating System.
Soft Real-time	In a soft real-time system, the deadlines of the system must usually be met, but it is acceptable if a small number of deadlines are missed occasionally.
TLB	Translation Look-aside Buffer. Provides a cache of address translations. On CRIS, if a translation that is needed is not cached, an MMU bus fault occurs and the TLB must be filled by software.
TSC	Time Stamp Clock.
User-Space	This is where standard applications such as web browsers, command shells, <i>etc.</i> run. Each application process has its own protected memory area where it can not interfere with other processes. See also <i>kernel-space</i> .