UPTEC IT08 016

Examensarbete 30 hp
December 2008

Performance Evaluation of GNU/Linux for Real-Time Applications

Tobias Knutsson

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Performance Evaluation of GNU/Linux for Real-Time Applications

Tobias Knutsson

GNU/Linux systems have become strong competitors in the embedded real-time systems segment. Many companies are beginning to see the advantages with using free software. As a result, the demand to provide systems based on the Linux kernel has soared. The problem is that there are many ways of achieving real-time performance in GNU/Linux. This report evaluates some of the currently available alternatives. Using Xenomai, the PREEMPT_RT patch and the mainline Linux kernel, different approaches to real-time GNU/Linux are compared by measuring their interrupt and scheduling latency. The measurements are performed with the self-developed Tennis Test Tool on an Intel XScale based Computer-On-Module with 128MB of RAM, running at 520MHz. The test results show that Xenomai maintains short response times of 58µs and 76µs with regard to interrupt and scheduling latencies respectively, even during heavy load of the Linux domain. When the Xenomai domain is loaded as well, responsiveness drops to 247µs for interrupt latency and 271µs for scheduling latency, making it a dead race between Xenomai and the PREEMPT_RT patched kernel. The mainline kernel performs very well when not subjected to any workload. In the tests with more load applied, performance deteriorates fast with resulting latencies of over 12ms.

Handledare: David Kiland
Ämnesgranskare: Wang Yi
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT08 016
Sponsor: Combitech AB

Tryckt av: Reprocentralen ITC

Sammanfattning (Swedish)

Det finns många exempel på maskiner som man normalt inte tänker på som datorer, men som innehåller de viktiga komponenterna som en dator har, en beräkningsenhet samt någon form av minne. Exempel på sådana system är DvD-spelare, bilar och mobiltelefoner. Sådana system kallas för *inbyggda system*. Många utav dessa har dessutom krav på att de måste utföra uppgifter inom vissa bestämda tidsintervall. DvD-spelaren måste till exempel rita upp 25 bildrutor per sekund för att bilden inte skall upplevas som instabil. Man säger att systemet är ett inbyggt *realtidssystem*. Traditionellt har inbyggda system haft väldigt begränsade resurser, vilket medfört att det inte varit möjligt att använda dem i kombination med exempelvis Linux som operativsystem.

I takt med att de inbyggda systemens resurser har ökat så har det blivit mer intressant att använda Linux i dessa system. Att Linux blivit populärt inom detta segment beror bland annat på att Linux bygger på *öppen källkod*, vilket betyder att vem som helst kan se exakt hur operativsystemet är uppbyggt. Många vill dessutom använda Linux i inbyggda realtidssystem på grund av denna orsak. Problemet är att det finns många olika sätt att anpassa Linux för att fungera bra i dessa system, vilket gör att det kan vara svårt att veta vilket man bör välja.

För att försöka ta reda hur de olika lösningarna skiljer sig åt har tre kandidater, som använder olika tekniker för att lösa problemet, valts ut och testats:

- **Xenomai**. Är ett eget litet operativsystem som kan köra Linux som en process samtidigt som den kan köra andra program. Genom att ge andra program högre prioritet än Linux kan man reagera snabbt när intressanta händelser inträffar.

- **PREEMPT_RT**. En modifikation av Linux som gör det lättare att bestämma vilket program som ska ha rätt att köra först, i situationer då flera program vill köra.

- **Linux**. För att testa vilka resultat som är möjliga endast genom att ändra Linux egna inställningar.

För att testa detta konstruerades ett testsystem som kallas Tennis. Tanken bakom namnet är en analogi till situationen då en människa försöker studsa en tennisboll mot en tegelvägg så fort som möjligt. Meningen är att mäta tiden mellan studsarna och på så sätt avgöra hur lång tid det tar för det system man vill undersöka, *målsystemet*, att reagera på intressanta händelser.

Praktiskt sett går det till så att man har en mätstation som skickar elektriska pulser till målsystemet. När signalerna når fram finns det ett program på målsystemet vars uppgift det är att svara på denna puls med en egen puls så snabbt som möjligt. Genom att mäta tiden mellan dessa pulser kan man ta reda på hur lång tid det tar för systemet att reagera på den första pulsen. När en sådan uträkning är gjord så skickas uppgifterna till en vanlig PC där de lagras permanent, varpå en ny iteration påbörjas. Totalt består varje mätning av tio miljoner iterationer.

Resultatet av mätningarna visar att både Xenomai och PREEMPT_RT klarar av att svara på händelser på omkring 250μs, trots att systemen är under kraftig belastning. Linux fungerar också bra så länge som systemet inte belastas. Men under belastning kan det ibland ta över 17ms innan Linux reagerar på att en puls har tagits emot, alltså 68 gånger så lång tid som för de andra två. Man bör dock tillägga att alla kandidaterna i de allra flesta fall svarar betydligt snabbare. På sidorna 67 till 76 kan man se hur svarstiderna är fördelade.

Innehåll

Kapitel 1

Introduction

In the area of real-time systems with high demands on security and stability, GNU/Linux has soared up as a strong competitor. It has a large user-base, a proven security model, the broadest device support and it is free in the full sense of the word. All of these features makes it interesting for anyone in need of an operating system that can be modified to fit their needs. This ability is especially important in the segment of embedded systems, where embedded refers to a system with a specific purpose that is integrated in a device of some sort, in contrast to a general purpose computer which is built to perform a wide range of tasks. The nature of embedded systems requires the ability to communicate with many different types of devices. Doing so may only be possible by low level surgery in the operating system's source code. If your system is not free, you have to rely on the manufacturer to provide this support. Additionally, many embedded systems have requirements regarding time, for example a DvD player must be able to produce 25 frames per second to make for a comfortable viewing experience. Such systems are called embedded real-time systems.

Many companies in the business of making embedded real-time systems have come to realize the benefits of using open source software. A well specified Application Program Interface (API) and a good documentation will get you far. But the additional possibility of open source software that allows you to look into the operating system's code and modify it gives a developer another dimension of control. Therefore, there is a steady stream of developers moving towards GNU/Linux implementations of their software.

1.1 Problem

Traditionally, the marriage between embedded real-time systems and GNU/ Linux has been less than ideal. The Linux kernel was never intended for use in neither embedded nor real-time systems[30]. But as the hardware resources have grown in the embedded systems along with the introduction Symmetric Multi Processing (SMP) to the mass-market, the landscape has changed. A modern embedded system often has more computing power then the systems for which the Linux kernel was originally designed, as a result it is no longer seen as too ungainly for the embedded world. Furthermore, many of the issues that arise when dealing with SMP systems are shared with real-time systems which have led to a more real-time friendly development philosophy among the Linux kernel developers.

As a result, consultant companies in the embedded field are eager to offer solutions based on GNU/Linux to their clients. One of those companies is Combitech, on whose behalf this report is compiled. At their Jönköping office, many of their clients are involved in making embedded real-time systems. The problem is that there are many different ways of achieving a Linux based system with real-time capabilities. This makes it hard to determine which way that best suits the case in question. Soft real-time systems might benefit from one type of approach while a system with hard deadline meeting requirements might be in need of another method.

1.2 Previous Work

Much research have been made in the area of how to apply real-time features to the Linux kernel. There were projects in existence ten years ago that adapted the Linux kernel to provide real-time services. In 1998, using an Intel Pentium processor running at 120MHz on system with 32MB of RAM, the RTLinux project were able to schedule tasks with a maximum latency of 15μs. This was done using a modified version 1.3.32 of the Linux kernel to which an interrupt dispatcher was added[31]. Today, RTLinux lives on in two forms. RTLinux Free, an open source version of the software and RTLinux Pro, a software component sold by Wind River Systems[25].

Another project under way in 1998 was the Kansas University Real-Time project (KURT)[13]. Their solution to the problem was to implement a new time keeping framework called UTIME, which provided more fine grained time keeping capabilities. In combination with UTIME they developed a new scheduling algorithm with strict priority assignments, and one where an application developer could define a static schedule. The result was a system achieving soft real-time performance without taking over vital functions from the standard kernel. KURT saw active development through the 2.4.x versions of the Linux kernel, but has not been updated since the release for kernel version 2.4.18[24].

A year later, in March 1999, the first official release was made from a project called Linux Real-Time Application Interface (RTAI)[18]. Using a microkernel approach, they presented a relatively rich real-time API. The RTAI framework provided services such as First In First Out queues (FIFOs), semaphores, shared memory, real-time schedulers to name a few. These were implemented as kernel modules, dynamically loadable at runtime. The architecture support was limited to the x86. RTAI is still a living project that has seen real-life implementations in robotics[5]. Support for additional architectures have been supported at times, but the project focuses on the x86.

Originating from RTAI, another microkernel based solution called Xenomai was declared an independent project from RTAI in 2005[10]. Xenomai focused on simplifying the migration process from traditional real-time operating systems to GNU/Linux by offering so called skins that emulated the API of an existing RTOS. Xenomai also aims to provide support for multiple processor architectures, x86, PowerPC and ARM among others[28].

In 2004, Sven-Thorsten Dietrich and Daniel Walker of Monta Vista Software Inc. released a modified version of the Linux kernel with preemptable locks. The modification led to ä dramatic improvement in the Real-Time response of the Linux kernel"[8]. Using this as a stepping stone, Ingo Molnar of Red Hat Inc. implemented his own version of a preemptable kernel based on an existing project of his called the Voluntary Preemption project. The result was the PREEMPT_RT patch that exists today. The goal is to achieve hard real-time performance without the use of a microkernel by means of using preemptable locks, threaded IRQs, high-resolution timers among other techniques[21].

Some of these projects are covered in more detail later on. For more information on these solutions, see Chapter 3.

The Real-Time Conference, 2007 IEEE-NPSS saw the presentation of a performance comparison between VxWorks, Linux, RTAI and Xenomai. Using a PowerPC based computer running at 1GHz with 1GB of SDRAM[19], the paper focused on measuring the interrupt latency and the scheduling latency. The conclusion was that VxWorks and RTAI generally achieved the best results with interrupt response times around 70μs and scheduling latencies of approximately 2μs. Xenomai, using a slightly different architecture, came in on a close third place. Additionally, the authors were surprised of how well the standard Linux kernel was able to keep up, trailing just a few microseconds after Xenomai. Notably, these tests were performed on unloaded systems, applying load quickly led to deteriorating results for the Linux kernel, while the other three saw only minor latency changes[2].

1.3 Goal

Real-time systems are complex and there are many features required to make them complete. The purpose of this report is to compare different ways of creating an operating system based on GNU/Linux with real-time capabilities, to weigh the pros and cons of different approaches. Rather than comparing all of the factors though, it will try to identify some key features that preferably should be measurable. It will try to determine which approach is best suited for different types of real-time systems based on these key elements.

Many of the previously existing reports have focused on evaluating a single approach to building a real-time system based on the Linux kernel. The most probable explanation is that choices have been more limited in the past. Furthermore, the state of some distributions have changed since the papers were written. Some have been commercialized and therefore have been made less accessible, others have been discontinued and thereby have become outdated.

There is a lack of coverage on the performance of current real-time options for Linux based systems running on the ARM architecture. Being a widely used architecture in the embedded world, home to a substantial number of hard real-time systems, this is a void that this report will try to shed some light on.

The quick pace of development within the Linux kernel and the PREEMPT_RT patch makes it interesting to present an up-to-date report on how these two projects measures up to more invasive ways of achieving real-time performing Linux systems.

Kapitel 2

Real-time Systems

A common misconception regarding real-time systems is that they are built to be fast. That statement is not necessarily false, but it depends on what is meant by speed. In computing, speed generally refers to the throughput of the system, that is to say the number of instructions that can be carried out within some time unit. In that respect, real-time systems are typically slower than their non-real-time counterpart. But what they lack in throughput they make up for in *responsiveness*, the time from an event to a response. A real-time operating system (RTOS) aims to offer as low response time, or latency, as possible. Even more important then a low latency however, is a *deterministic latency*, i.e. the ability to give a bounded interval for the latency of the system.

Deterministic latency is especially important in *hard real-time systems*. Hard, refers to the deadlines of the system which must be met, at all times. Typical examples of such systems are anti-locking brakes, chemical factory controllers, airplane landing gear controllers , etc. As you would expect there are also *soft real-time systems*. These are systems which benefits from real-time characteristics, but unmet deadlines does not endanger human lives or risk damage to property. Media applications are one example of soft real-time systems. For example a music player must update its output with a frequency of at least 1 kHz, since the human ear can detect latencies over 1 ms[11]. Failing to meet a deadline in this case will cause the listener to notice a skip in the song, whereas if a deadline was missed in an anti-locking brake system someone could die.

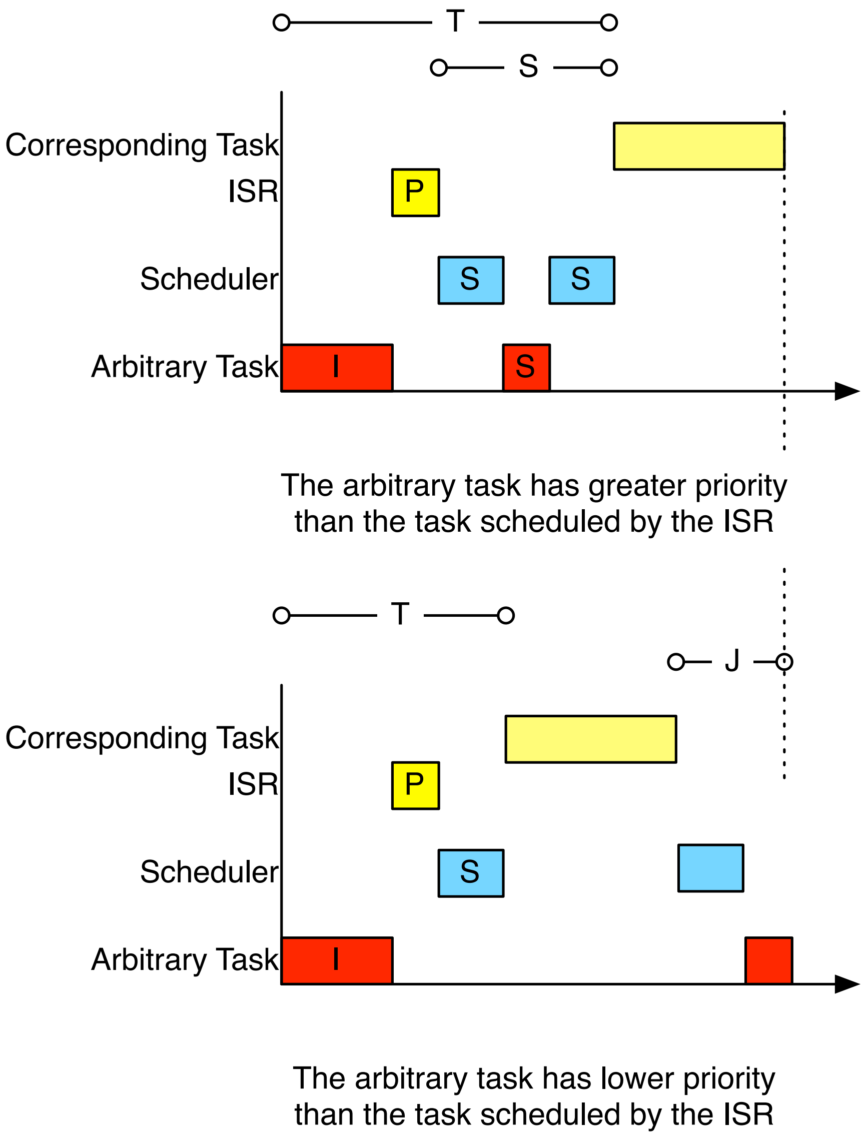
2.1 Latencies

Response times, or latencies, can be measured at many different points throughout a system. Since embedded real-time systems often have to react to input from an external device, the most commonly used is the *interrupt-to-process latency.* This is the sum of three main latency sources. First the *interrupt latency*, which is the time from when a central processing unit's interrupt line goes active until the operating system handles control over to the associated interrupt service routine. This latency is introduced by the fact that some parts of the OS's code must run with interrupts disabled for concurrency purposes. Secondly there is the *interrupt processing latency.* During this time the most time critical tasks are performed, for example reading of I/O registers and sending interrupt acknowledgements back to the device. Finally the OS's scheduler must be called. The scheduler must then decide which process to run. If there are other processes waiting to run with higher priority than the one related to the received interrupt they will be scheduled first. The time until the scheduler selects the process that will process the received data is called the *scheduling latency.* At this time the data can be interpreted and the appropriate action can be taken[11].

Figure 2.1 shows an abstraction of an arbitrary process running on a CPU when an interrupt is received, and then tries to illustrate the different latency sources. The following list describes the figure in detail:

- I is the time from when the interrupt is received until the interrupt service routine (ISR) is run, in other words the interrupt latency.
- The length of the ISR, P, corresponds to the interrupt processing latency.
- From the time when the ISR has finished until the time when the corresponding process is run, denoted as S, is the scheduling latency.
- T is the total interrupt-to-process latency, i.e. $T = I + P + S$.

A closer inspection shows that T varies between the two examples. That is, the total interrupt-to-process response time is not constant. Since deterministic latency is the most important feature of a good real-time operating system, this variance is an important measure of a system's performance. This property is called *jitter*. Jitter is defined as the maximum variation in a


T
S
Corresponding Task
ISR
P
Scheduler
S
S
Arbitrary Task
I
S
The arbitrary task has greater priority than the task scheduled by the ISR
T
J
Corresponding Task
ISR
P
Scheduler
S
Arbitrary Task
I
The arbitrary task has lower priority than the task scheduled by the ISR

Figur 2.1: *Illustration of latency sources.*

process's release"[4]. Thus, the jitter J of a system is the difference between the longest and shortest possible T.

2.2 Scheduling

As seen in Figure 2.1, the latency of a system is largely dependent on the priority of the concurrently running processes and the policy used to decide in which order those processes are to be executed. This is the job of an operating system's scheduling policy. The ultimate scheduler would have to implement many contradictive objectives simultaneously. It would ensure a high data throughput while maintaining low and deterministic response times. On top of this, it would have to make sure that no process suffered from starvation. As you might have guessed, there is no such thing as an optimal scheduling policy. There are many different ways of designing a scheduling algorithm, each has its own benefits and drawbacks. One of the fundamental differences between real-time operating systems and general purpose operating systems is what features that should be prioritized.

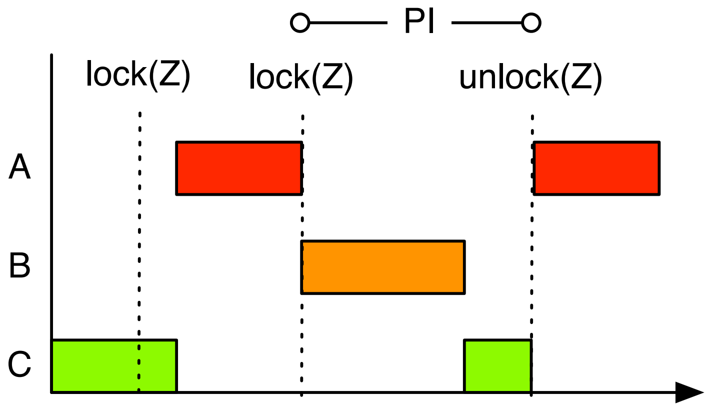
With origins in the UNIX world, the Linux kernel tries to be as *fair* as possible. That means that the available time to execute code on the CPU is divided among the existing processes. Additionally, processes can have different priorities, where those in the higher range will get a bigger time-slice and possibly get to run before processes with lower priority. A situation could occur however, where the lower process is allowed to run first, if it has a larger chunk of its time-slice left[3]. Thus, no process will ever be completely starved, even if there are many processes with higher priorities running at the same time.

In a general purpose operating system, this is a desired feature. Even though a big important job is being run on a particular machine, it is nice to be able to run a lower priority job side-by-side and have it finish within a reasonable time. In a real-time system however, fairness could lead to catastrophic results. Imagine running a fair scheduler in the computer of a car. At some point, one of the positioning lights stop working at which point a process is scheduled to display a message, relaying the information to the driver. At that time however, the computer is adjusting the fuel injection parameters, a high priority job, so the message process will have to wait. When the fuel injection adjustment is done, the car skids off the road and hits the railing. The computer schedules the job off launching the air-bags. But since the scheduler aims to be fair and the message process have been waiting for quite some time, it is allowed to run first. The result is that the message of the broken tail light is printed onto a screen before the air-bags are released, later than they could have been. This is why real-time scheduling policies are explicitly *unfair*, the process with the highest priority will always run first.

Only deciding which process to run after another one has finished will most often not be enough. Had the car hit the railing while adjusting the fuel injection settings, the natural action would be to halt the process currently running and invoke the scheduler again which would then decide that the launch-air-bags-process should be run first. This mechanism is called *preemption*[4]. In a real-time operating system, it is very important that the currently running process can be blocked and one with a higher priority scheduled as fast as possible. Any delay in this process will directly increase the system's response time and jitter. In a general purpose operating system it might be more appropriate to have fewer preemption points in the system. Thus, fewer context switches will take place and the total throughput will be higher.

Fine grained preemption will help keep the latencies low and predictable, but there are still pitfalls that must be overcome by a real-time scheduler. One of them has to do with the fact that there are certain operations that must be carried out in an atomic fashion. That is to say, there is some resource that can only be accessed by one task at a time. Such sections are often referred to as *critical sections*. The usual solution is to use some kind of lock that the task must own in order to access the resource that it must release when it does no longer need the resource in question. The problem arises when a high priority process needs a resource locked by a low priority one. Without extra thought to the scheduling policy, the high priority task would have to wait for all processes with higher priority than the one holding the lock to complete their execution before the low priority process would get a chance to release the lock. Imagine for example three tasks, A, B and C with the priorities high, medium and low respectively. These processes can all acquire the shared resource Z, but only one process can use the resource at once. Assume that C acquires Z. The high priority process A then becomes eligible to run. After executing for some time, it needs to use Z, but it is locked by

C. *A* will therefore be blocked. At this point *B* is the highest priority process which is not blocked, so therefore it is allowed to run, and can continue to run for an undetermined amount of time. Only after it is done is *C* scheduled to run, when it releases *Z*, *A* is unblocked and can continue its execution. The fact that the medium priority process *B* gets to run before the high priority process *A* is called *priority inversion*, which Figure 2.2 tries to illustrate.



Figur 2.2: *Classic case of priority inversion.*

A common solution to this problem is called *priority inheritance*. The basic idea is that a process holding some resource has its priority temporarily elevated to the highest priority of the processes waiting for that resource[4]. In this case it would mean that when *A* wants to acquire *Z*, *C* temporarily becomes a high priority process. Therefore, *C* will run before *B* and once it releases *Z*, its priority will be restores. At that point, *A* will be able to continue its execution, once it is done *B* will be scheduled to run.

Additionally, there are situations in real-time systems when it is required to perform some task at a periodic interval. There might be hardware devices that need attention at regular intervals for example. In order to support as small and as regular intervals as possible, it is important that the scheduler has an accurate and high resolution sense of time.

Kapitel 3

Approaches to Real-Time GNU/Linux

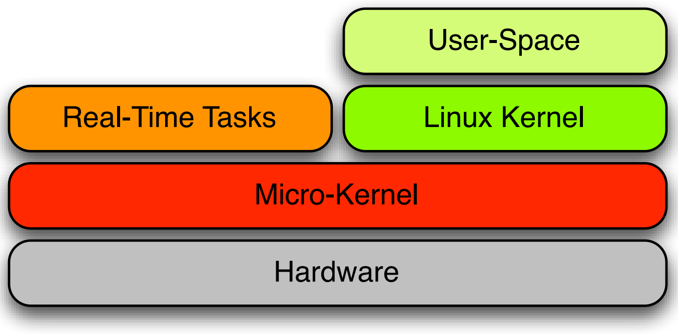
So how did GNU/Linux come to be a player in the embedded and real-time world? The first public announcement of the Linux OS came on October 5 1991, by founder Linus Torvalds. The purpose of his project was to make a free UNIX-clone for him and his fellow hackers. An OS without the limitations and high prices of the commercial UNIX distributions offered at that time. The developer base grew and the device support and stability grew along with it[17]. However, Linux by itself it not a complete OS. Linux is a kernel, thus it handles the core services such as device I/O, file systems, time-keeping , etc. But a complete OS is expected to provide more functionality; some sort of interactive shell and the ability to perform file operations to state a few. This is where GNU comes into the picture.

GNU's Not UNIX (GNU) is a project that started in 1983. Its goal was to create a complete and free OS. Unlike Linus Torvalds they decided to start with the tools and utilities needed to make a complete OS. By 1990 they had everything they needed except for the kernel. Therefore, when Linux became free software in 1992, the Linux kernel was combined with the tools and utilities from the GNU project to create a complete OS consisting of free software only[22].

With respect to its background its easy to see that GNU/Linux was not developed with embedded or real-time systems in mind. Thanks to its free nature though, developers have been able to tailor it to their needs which have allowed it spread from its origins as a desktop OS both into the server market as well as the embedded.

There are of course an infinite number of ways in which to create a GNU/-Linux system with real-time properties. The existing solutions are also built on more or less different architectures. In order to make for an easier overview however, the approaches have been grouped into three distinct categories. Solutions are grouped by the level of deviation from the standard Linux kernel tree. It should be noted however, that this is by no means the only possible way to group these approaches.

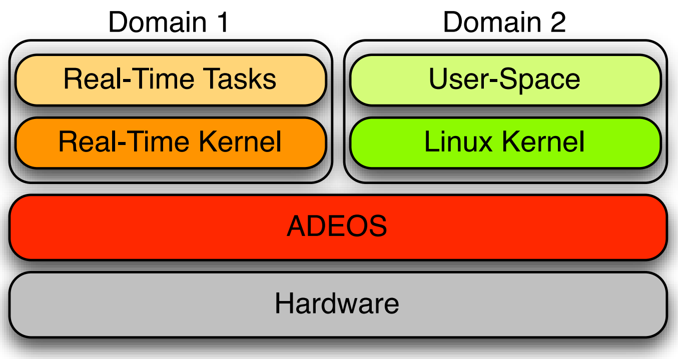
3.1 Micro-Kernel



Figur 3.1: *Layout of a micro-kernel system.*

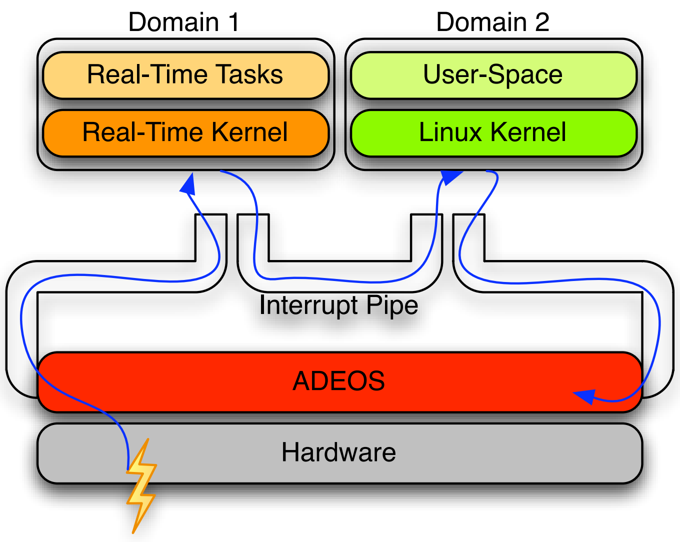
Microkernels where used in the first attempts at creating hard real-time systems based on the Linux kernel. The idea is to create a small operating system that has hard real-time capabilities. This operating system has to have the capability to handle incoming interrupts, scheduling tasks, handle synchronization services and other core functions[15]. So in a way, one could argue that using a microkernel architecture is not really a Linux system at all. The Linux kernel is just one task in the smaller operating system, the microkernel. Other tasks can serve hard real-time jobs by being assigned a higher priority than the Linux kernel task. The advantage of this approach is that hard real-time performance can be achieved even though the Linux kernel itself does not provide any such features. Essentially there is one real-time system and one non-real-time system running concurrently where the real-time system always has precedence over the non-real-time one. Figure 3.1 shows the idea behind this approach. There are two open source projects in development today that uses a micro-kernel architecture in combination with the Linux kernel, RTAI and Xenomai. They stem from the same project originally, so naturally they are somewhat similar. For example they both utilize ADEOS.

3.1.1 ADEOS



Figur 3.2: *ADEOS architecture.*

In the same sense that GNU/Linux should not be referred to as Linux, RTAI and Xenomai should in all fairness be called ADEOS/RTAI and ADEOS/Xenomai. The Adaptive Domain Environment for Operating Systems (ADEOS) is a *nano-kernel*. Just as the name implies, a nano-kernel is an even thinner kernel model. ADEOS intends to provide the possibility to share hardware between several operating systems[29]. This is accomplished by installing ADEOS as a hardware abstraction layer between the hardware and the actual operating systems as seen in Figure 3.2. ADEOS then allows the creation of multiple domains, which are basically sandboxes in which operating systems can be made to believe that they are communicating directly with the computers hardware. The code running in a domain does not have to be ignorant of ADEOS however, there are ways for communication. For example, the domains are arranged in a line, referred to as the interrupt pipe, which can be determined either by ADEOS itself or explicitly by a registered domain that is aware of ADEOS's presence, i.e. RTAI or Xenomai. When an interrupt is triggered, ADEOS will redirect it to the first domain in line. After that domain has processed that interrupt, it will be propagated to the next domain in line until all domains have been notified of the event. However, a domain with knowledge of ADEOS may choose to signal that the interrupt should not be relayed to any other domains down the line. Thus, by placing a real-time operating system at the head of the line, that system can call dibs on all interrupts before the other domains are aware of their existence. Figure 3.3 shows an example of an interrupt's route through a system.



Figur 3.3: *Abtraction of the Interrupt Pipe.*

Operating systems are supposed to directly control the hardware on which they run, this is after all their primary purpose. When using a nano-kernel however, the operating system must be enslaved by the nano-kernel in order to prevent it from ignoring the restrictions imposed by ADEOS. For example, the operating systems running under ADEOS can no longer be allowed to disable and enable interrupts at will. Disabling interrupts would eliminate any chances of ADEOS regaining control after handing it over to some domain.

One way of achieving this goal would be to change all of these operations in the operating system's code, before compiling it, to some calls that could be intercepted by ADEOS. Instead of using this rather invasive tactic however, ADEOS makes use of the fact that many architectures have hardware support of different privilege levels, or rings. The idea is to protect the system's integrity by restricting access to certain processor instructions in the outer rings, typically for running user applications, while code executing in the inner rings, typically the operating system kernel, have more privileges. When an attempt is made to execute an instruction from an incompatible level, a fault will be generated in the processor. The idea is for ADEOS to run at a privileged level and move the Linux kernel to a level where the dangerous instructions are considered illegal. When the Linux kernel tries to execute one of these instructions, ADEOS will intercept them and take the appropriate action. In order to maintain good throughput, only the most critical instructions are intercepted, Linux is still allowed to have direct access to hardware registers and maintain memory.

Using a loadable module implementation of ADEOS in combination with a special ADEOS domain positioned in front of the Linux domain, ADEOS is able to take control of the hardware without the Linux kernel's knowledge. This can be done on a standard kernel without any modifications. The first step is to load the ADEOS core module, the nano-kernel, which will be set up into an idle state. Then the Linux Handling Domain (LHD) module is loaded, which is the ADEOS domain to be positioned in front of the Linux domain in the interrupt pipe. Upon loading this module, the Linux kernel will be moved to an outer ring, and ADEOS will remain in the inner ring, calling all the shots. When the Linux kernel wants to disable interrupts, a fault will occur which will be caught by the LHD which then will stop all interrupt propagation to the Linux domain. However, domains in front of the LHD in the pipe, a hard real-time operating system for example, will still receive the interrupts when they occur. At some point, the Linux kernel will try to enable interrupts again, generate a fault caught by the LHD which will then start to relay interrupts to the Linux kernel again.

3.1.2 Xenomai

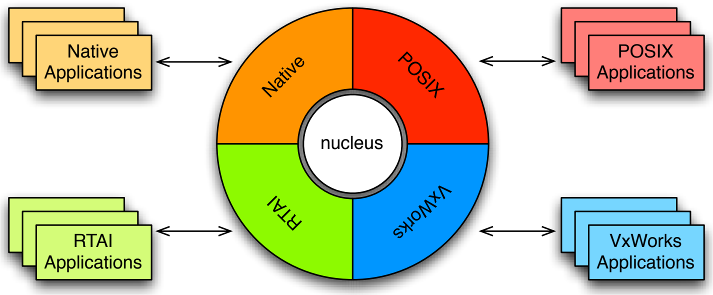
As is customary in Linux kernel development, Xenomai is implemented as a loadable kernel module. First, it uses the ADEOS nano-kernel to move the

Linux kernel to an ADEOS domain as described in Section 3.1.1. It then installs itself in a separate domain. Thanks to the fact that Xenomai is aware of ADEOS's presence, it can communicate with it. Therefore, Xenomai makes sure that its domain is placed at the head of the interrupt pipe. The resulting system has a layout which allows all interrupts to be processed by Xenomai before they reach the Linux domain.

Once Xenomai has been loaded, real-time applications can be implemented in different ways. If the objective is to write a device driver, the most natural might be to implement it via Xenomai's native interface and the Real-Time Driver Model (RTDM) interface available to Linux kernel modules. The module will use the standard way of loading modules in Linux as its entry point. From inside, it will then make calls to the Xenomai domain to set up interrupt handlers, tasks and so on. Being part of the kernel address space allows Xenomai modules to invoke functions available in the kernel. These might be requests to read hardware registers which are usually implemented in assembly language and are real-time safe. More intricate functions, such as printing functions, may also be invoked. Such methods are highly discouraged however, since such functions could cause unbounded latencies, thereby destroying the real-time behavior of the mirror.

If for some reason, it is desirable to implement some part of the real-time system in user-space, Xenomai provides the means to accomplish that. Separate from the Xenomai kernel module is the possibility to compile a set of user-space libraries which can be used by standard linux applications. There are advantages to this approach, for one developers might have more experience in writing standard programs then kernel modules, also, being in user-space provides floating point arithmetic operations and the ability to debug programs using the GNU Debugger (GDB).

Another advantage to writing user-space applications has to do with migration from other operating systems. Xenomai uses an API architecture in which a core API, called the nucleus can be interfaced from different compatibility layers. Figure 3.4 shows how applications written for different operating systems can utilize Xenomai's real-time core services by converting their native APIs to the generalized Xenomai nucleus. In practice, the skins are built up of different libraries which can be used to communicate with Xenomai.



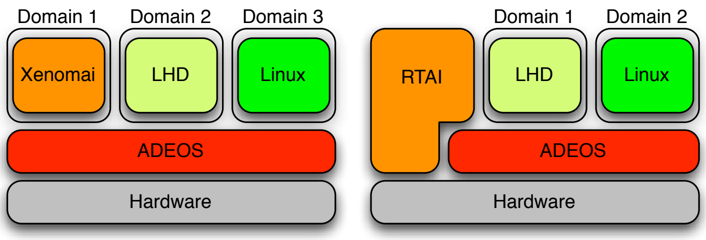
Figur 3.4: *The Xenomai API structure.*

Xenomai continues to be a very active project. In 2008, the project has published five releases so far. It currently supports six different compatibility skins in addition to its own API. The project aims to be as portable as possible and currently runs on a number of architectures. Both the 32 and 64-bit versions of both the x86 and the PowerPC are supported together with nine different types of ARM processors, Blackfin processors and Intel's server architecture IA64.

3.1.3 Real-Time Application Interface

As mentioned in Chapter 1, Xenomai is a project that is branched of the RTAI project, thus they are similar in some ways. For instance, they both utilize the ADOES nano-kernel to remove some of the Linux kernel's privileges and install their own domains at the head of the interrupt pipe[9]. A major difference is that RTAI attempts to cut the response times even further. To do this, they put the RTAI core in control of receiving all interrupts instead of ADEOS. Incoming interrupts that have not been registered with RTAI, are sent along to the Linux kernel by forwarding them to ADEOS and letting its nano-kernel do the usual work. In the event of receiving an interrupt that is associated with an RTAI application, the interrupt service routine associated with that interrupt will be awoken immediately[2].

The resulting architecture is somewhat more complex than that of Xenomai as seen in Figure 3.5. The major advantage is that the latency introduced by the nano-kernel is avoided for RTAI applications since ADEOS will never see these interrupts.



Figur 3.5: *The Xenomai and RTAI architecture compared.*

Comparing the API available to RTAI and Xenomai shows yet again that these two projects are still quite closely linked. Both have support for the features you would expect of a real-time operating system. These include basic task and time management services with dynamic priority assignments. Other capabilities are inter-process communication via message queues and shared memory that can span across kernel- and user-space, different types of synchronization services such as semaphores and conditional variables. They also share the capability to implement drivers in user-space with the support for user-space interrupt service routines[27][9].

Just like Xenomai, RTAI is still under active development. However, there is a difference in the objectives of the projects, which might explain why they forked. RTAI is focused on achieving the best performance possible. Xenomai is more interested in having a portable solution that is easy to maintain and extend[28].

3.2 Enhanced Kernel

The enhanced kernel approach is quite different from the micro-kernel approach in their ways of attempting to achieve real-time performance in a Linux system. Using a micro-kernel is in many ways the easy way out. It is not a simple matter of course, but it still allows the original Linux kernel to exhibit non-real-time behavior since it is run in a sandbox with lower priority than the real-time applications. While that solution certainly has its advantages, it does not deal with the root of the problem; the fact that the Linux kernel does not provide real-time characteristics. The enhanced kernel approach instead tries to fix the root problem by modifying important sub-systems and possibly add some new frameworks.

3.2.1 Kansas University Real-Time

One of the first projects with the ambition to produce a hard real-time system based on Linux, without the use of a micro-kernel, was Kansas University Real-Time (KURT). They identified timer resolution and scheduling abilities as two of the major obstacles for achieving real-time characteristics in Linux at that time. Timer resolution in a standard configuration Linux kernel is usually set to 100Hz. In other words, the minimum difference in time that the kernel is aware of when it comes to scheduling tasks for example, is 10ms. While suitable for a general purpose operating system, this is to long for many real-time applications. It is possible to simply increase this value to get a higher resolution. The problem with that solution has to do with the implementation of timers in Linux. Due to the fact that a timer interrupt is triggered every 10ms, even if no events are to be scheduled at that time, the overhead of running the interrupt service routine becomes large when increasing the interrupt rate towards the resolution required by some real-time applications.

Their solution to this problem was a new timer framework with microsecond resolution dubbed UTIME. The fundamental between the standard timer framework and UTIME is that the processor hardware timer is reprogrammed to fire interrupts when the next Linux timer expires instead of firing on a periodic basis[13]. Thus, one can produce a high resolution without the hazzle of linearly increasing overhead in the interrupt processing. In order to maintain compatibility the standard Linux components that requires the regular time tick, they also implemented a compatibility layer emulating the API of the old time framework.

Having a high resolution timer interface is of course useless if it is not used by some system. The second issue that the KURT project identified with Linux at that time was the implementation and available policies of the scheduler. Linux does not provide the possibility for a developer to provide an explicit schedule in which to run processes. Even if it would provide such a policy, the scheduler lacked the temporal resolution for it to be meaningful. Therefore, they implemented their own scheduling policy called

SCHED_KURT. This scheduler required a schedule being supplied, stating at what intervals each task would be run and for how long. KURT could be run in two modes, focused or mixed mode. In the focused mode, only KURT real-time tasks where allowed to run, all standard Linux tasks where blocked. Running in this mode eliminated some indeterministic behavior of Linux components not written with real-time constraints in mind. Mixed mode allowed tasks that where not KURT tasks to run in the gaps of the specified real-time schedule[13].

The real-time tasks in KURT where implemented as loadable kernel modules which can register themselves with the KURT core. This is done by passing the core a pointer to a structure containing pointers to an initialization function, a cleanup function, the name of the module, the function to call when the task is scheduled and the argument to call that function with. A system call can then be issued from user-space to specify when the real-time scheduling should begin and which schedule should be used at that time. The schedule is passed as a regular file following a special syntax known by the core[13].

Even though the KURT project is not in active development today, the project identified some of the key obstacles that had to be overcome if the Linux kernel ever was to be able to perform real-time tasks without the use of a micro-kernel. They also did the actual implementation and showed that it was possible.

3.2.2 The PREEMPT_RT Patch

Today, there is one major player in the enhanced kernel segment of real-time GNU/Linux. That is the so called PREEMPT_RT patch, or the rt-patch, named after the entry it registers in the Linux kernel configuration tool. There is a fundamental design difference between KURT and PREEMPT_RT. While KURT did not make use of a micro-kernel it still used its own scheduler and a custom way to register real-time tasks. Applying the rt-patch to a standard Linux kernel source changes none of the APIs. In essence, the rt-patch attacks the same weaknesses as KURT did back in the day, timer resolution and scheduling, but they do it in quite different ways.

Perhaps the most important feature that the rt-patch brings to the kernel is the introduction of *sleeping locks* to protect shared resources, called *rt_mutexes*. In Linux, the normal way to protect a resource against unwanted concurrent accesses is to use *spin locks*. If a spin lock fails in acquiring the lock, it will spin in a busy loop trying to grab the lock until it succeeds at which point execution will continue. This is another example of the kernel being optimized for throughput, since a spinning lock requires no scheduling and context switch while waiting for the lock, they are quite fast. In a real-time operating system this is devastating, no re-scheduling means no preemption which means that a high priority task would have to wait for a lower priority task to finish its critical section. As a solution, the rt-patch converts most of the spin locks to sleeping locks, some spin locks are kept however since there are critical sections which have extreme time requirements and cannot be interrupted. In practice this is done using the latest GCC extensions and some rather complex macros[21]. Using sleeping locks introduces the possibility to preempt a lower priority task, even if that task is executing in a critical section. Together with advances in the mainline Linux kernel, which makes an option available to make all code outside of critical sections preemptable, a kernel with the rt-patch is almost fully preemptable[21].

Using sleeping locks does not, by itself, solve the problem of preemption however. Settling for the solution that was just described will make the kernel essentially fully preemptable. But without attacking the priority inversion problem, indeterministic behavior still lingers. Even if a low priority task can be preempted when a high priority needs attention, if the low priority task holds some resource needed by the high priority task, priority inversion could occur if there are other processes running at the same time. Thankfully, the PREEMPT_RT patch has solved this problem as well by implementing priority inheritance in the rt_mutexes[21].

In order for it to be possible to preempt a task, that task must be running in a threaded context, i.e. the scheduler must be aware of its existence in order to be able to tell it what to do. In the mainline kernel however, there are some subsystems that make use of spin locks that are not running in a threaded context. The most prominent example of such a subsystem are the interrupt handlers. With the rt-patch, a new thread is created for every registered interrupt handler in which they run. Linux also makes use of software interrupts, signals that can be sent by software to trigger events in the kernel. The handlers for these signals are also placed in separate threads by the rt-patch. As a result interrupt handlers can be preempted by other tasks, both from kernel-space and by user-space processes, since they are all handled by the same scheduler. Moreover, it brings more customization possibilities to the system as a whole. It is possible to prioritize the execution of a user-space process over the handling of interrupts and also to perform differential rankings between interrupt handlers. An interrupt handler can still request to not run in a threaded context by passing a flag to the kernel when registering the interrupt. One example of such a handler is the one responding to the timer interrupt which needs to be run instantly. If the timer interrupt service routine was not allowed to run, there would be no re-scheduling, as the scheduler relies on the timer tick. The price of running in true interrupt context is that no lock can be taken since this would lead to a deadlock if that lock was taken by a thread, since that thread will not be able to run until the interrupt handler has finished processing[21].

In order to address the issue of timer resolution in the Linux kernel, the rt-patch makes use of a closely related project called *ktimers*. Ktimers is an extension to the Linux timer system. The project recognizes a problem with the standard Linux timer system; the timer system is highly optimized for managing timers that will not expire in most cases, while timers that do expire are not handled in an optimal manner. Previous attempts, such as UTIME, has tried to re-write the timer system to perform both tasks well. Ktimers instead leaves the current implementation in place for use with timers that are not expected to expire, for which it performs well. Examples of these timers are typically device drivers using a timer to signal a timeout during I/O from some device, which normally works, and therefore the timer is removed before it expires. For timers that almost always expires, typically sleeping operations and cyclic scheduling, ktimers provides a new timer interface with nanosecond precision, providing that the underlying hardware supports it[8][6].

Development in the PREEMPT_RT project is very active, often producing several releases for each release of the Linux kernel. Also, some features of the patch have been merged into the mainstream kernel. This is an important difference from previous projects such as KURT, the goal is to merge the whole rt-patch into the kernel and present it as an option in the standard Linux kernel configuration.

3.3 Mainline Kernel

Kernel versions prior to the 2.5-series of Linux would not have been possible to use for real-time applications. The reason is that the 2.4-series and earlier where to a large extent non-preemptable. Once a job in the kernel was launched, it was impossible to stop it unless it invoked the scheduler itself. The 2.6-series of today however are a different story. The standard kernel configuration offers an option to enable in-kernel instant preemption except in critical sections. Though this means that the kernel is not fully preemptable, some of the longest code paths are possible to interrupt if some higher priority task needs to run. This transition has been made possible in large parts thanks to the introduction of Symmetric Multi Processing (SMP). When then Linux kernel was adapted to run on SMP systems, all critical sections that needed protection from concurrent access was identified. The kernel developers then realized that the same sections marked boundaries, outside of which it would be safe to enable preemption in the kernel. With this in mind they presented an option that took advantage of this fact and enabled preemption in safe parts of the kernel[21].

There is also an option to replace the standard scheduler with a more deterministic replacement. Scheduling a process with the standard scheduler takes linearly longer time with respect to the number of processes and tasks available for scheduling. Thus, on a system with 100 processes, scheduling will take at worst ten times as long to schedule than a system with 10 processes. On a GNU/Linux system, it can be difficult to estimate the number of processes that will be running at a particular time, which means that the scheduler will perform in an indeterministic way. The replacement scheduler always schedules in a constant time, independent of how many processes are in the queue[14]. This scheduler is called the O(1) Scheduler, and is one example of a component that was originally developed for the rt-patch that has since been merged into the mainline kernel.

Finally, there is also a way to enable high precision timers in the form of so called *hrtimers*. Both hrtimers and ktimers builds on top of the existing timer system and leaves the current implementation intact. The difference is that hrtimers relies on the standard timer system for some parts of its functionality, while ktimers are built on its own foundation[8].

Kapitel 4

Evaluating Real-time Performance

When performing a comparative test between different solutions to the same problem there are many factors to consider. For one, it must be decided which features of the system that should undergo testing. Secondly, there might be more candidates available for testing then there is time available to carry out the tests. In that case, one must decide on which candidates that should be selected so as to give a correct image of the alternatives available. Finally, the method of testing must be specified in a way that does not favor any of the candidates. Favoritism can never be completely avoided, but it can be minimized.

4.1 Evaluation Features

As discussed in Chapter 2, there are many parameters involved in deciding the health of a real-time operating system. The kernel has to provide means for preemption as often as possible, ideally at an arbitrary location in the code. In addition, the kernel should protect the system from priority inversion and provide a robust real-time scheduler. In order to be able to perform scheduling and sleeping with a high accuracy, with respect to time, the operating system should provide a timer interface that offers a high resolution. Of course the most important feature is to provide deterministic response times, in addition the response times should be as low as possible.

It is out of the scope of this report to evaluate all of these factors separately. However, the resulting worst-case response time and jitter of a system can be seen as the aggregate of the performance of all other components performance. The reason is that every one of these components have a direct negative effect on the resulting response time, thus no sub-system can misbehave without it manifesting on the latency. In the process of optimizing and debugging a real-time system with the intent of lowering the latency, such a performance test would be less useful. It would only show that some component is generating long delays, but would not be as helpful in pin-pointing the actual component. In this case however, the objective is to compare different solutions as they are working today. Thus, measuring the response time will produce relevant results.

The maximum response time of a system can certainly be important to measure from an application point of view. However, it is often in the interaction with other devices that low response times are most important. There are tools available that makes use of the parallel port available on most desktop computers to measure these latencies. They usually work by utilizing the interrupt line available on the standard parallel port. A wire is connected from one of the output pins of the port to the interrupt line. Then a kernel module is loaded that registers the parallel port interrupt line to an interrupt handler. By storing the current time and changing the level of the output pin, the interrupt is triggered. Comparing the time at which the interrupt handler is run to the one previously stored gives the interrupt latency.

Such a tool has the advantage of only requiring a wire to function. However, there are several drawbacks with this kind of tool in our case. For starters, in the embedded world, parallel ports are few and far between. There are of course other ways of performing I/O, so such a tool could be ported to operate one some other pins. The real problem lies in the architecture of such tools. Since the measurements are made by the same system that is being measured, the tool will color the results. The precision in time of the measurements, depend on the system which is being audited. If there is a clock drift or some other unknown error in the time keeping system, this would go unnoticed by the test. Even though the most important part of the test lies in determining the worst-case latency, it is often desirable to log the result of all iterations in order to allow constructions of histograms and other graphs that show the distribution of the response times and thus the jitter.

But to log that much data usually requires it to be stored in some permanent memory. Storing data in permanent memory means doing potentially lengthy disk I/O which will also color the results.

This report is also interested in exposing any differences not only in the interrupt response time, but also in the scheduling latency. Therefore, a tool was needed that had the capability to measure the relevant latencies on an embedded platform. Additionally there was a need for a tool that would measure these features while presenting minimal changes to the system. As a solution this report suggests the Tennis Test Tool, presented in more detail in Chapter 6. Tennis uses external hardware to perform the measurements, thereby eliminating some of the problems discussed earlier. It is able to measure both the interrupt latency and the scheduling latency. Since all data is logged to permanent storage, it is also possible to calculate the jitter of the measured system.

4.2 Candidates for Testing

In order to give a representative study of the real-time GNU/Linux alternatives that are available today, the same tests should be performed on the same hardware using different Linux alternatives. Ideally, the same basic kernel version should be used in all tests, to which different modifications can be made. Additionally, candidates should be selected in such a way that the results of different types of modifications to the standard kernel can be studied.

Since there has been much development within the Linux kernel recently that could be beneficial from a real-time perspective, this report will select the mainline Linux kernel as one of its test candidates. This will determine how a simple customization of the standard kernel by means of changing a few default values in the kernel configuration tool can move Linux towards the real-time operating system arena.

As the second candidate, the Linux kernel with the PREEMPT_RT patch is chosen. Being the only alternative in active development today in the segment of kernel enhancements it would be interesting to see how far on the way to a real-time operating system Linux can get. That is to say, Linux without the use of a micro-kernel or some other underlying system that provides the actual real-time capabilities.

Finally, the line up would not be complete unless the two candidates selected so far were challenged by a micro-kernel solution. In this segment there are two projects that provide similar solutions, RTAI and Xenomai. This report will chose to examine the performance of Xenomai. Even though RTAI has been known to outperform Xenomai in some cases[2], Xenomai's more structured layout together with broader architecture support makes it more suitable and practical in an embedded environment.

Table 4.1 shows the exact versions of the three candidates that where chosen. The Linux kernel version 2.6.25.8 was chosen as the base merely because it was the most recent kernel available at the start of this project in June 2008.

	Mainline	PREEMPT_RT	Xenomai
Linux kernel version	2.6.25.8	2.6.25.8	2.6.25.8
Modification version	-	-rt7	2.4.4

Tabell 4.1: *Software versions of the test candidates.*

4.3 Test Method

In a way, the world of real-time is quite a pessimistic one. Results are only considered valid if they are extracted from the worst possible scenario. It does not matter if a system can produce a constant response time of 12μs except for one case that comes along every 403rd year, in which its response time is 2.3ms. Such a system would still be said to have a response time of 2.3ms with a jitter of 2.288ms. In real-time, you are only as good as your worst-case. The ideal way to determine the worst-case response time is to mathematically prove it. Using that approach requires the evaluation of every possible path that the execution might take in the code. For light weight systems that does not use an operating system, such an investigation could be feasible.

The Linux kernel however contains over 6 million lines of code[20]. To that one must add user-space libraries and applications. With such a large code base, tracing down every code path becomes a work worthy of comparison with Sisyphus's. Therefore, it is important that the measurements presented in this report are carried out in conditions that are likely to expose the worst-case performance of the system in question. The most recent performance measurement have not fully taken this in to consideration however, since the test was carried out on systems without applying any load[2].

This report will also carry out tests on the systems in an unloaded state. These tests will not however claim to measure the worst-case response times of the systems. Instead they are going to be used for comparison against other tests in order to estimate how much deterioration the system suffers with an increase in system load.

There will also be a set of tests carried out in which the systems are exposed to heavy load. These tests can not guarantee that the worst latency recorded is in fact the worst-case latency of the system. But recording ten million iterations of interrupt and scheduling latency measurements in each test, should provide a rather accurate approximation. Several different tools will be used to create the system load. The main tool will be Stress, which will create a variety of work that involves as many of the components in the Linux kernel as possible. Another series of tests are planned in which an extra load is to be applied consisting of extra network load and disk I/O, by continually sending files to the target system over the network. The reason is that Stress does not provide the possibility to put stress on the network natively. Finally, the Xenomai system will have to suffer the load of a third factor in the form of the Xload application in an attempt to introduce some load at the micro-kernel level.

4.3.1 Stress

Tracing through all possible code paths of the kernel is not an option to determine the worst-case latency. So in order to get as good of an estimation as possible, a tool is required that can traverse as many of these as possible. Stress offers a way to induce several different kinds of system load, therefore it interacts with many components of the kernel and should be able to expose some of the longer code paths through the kernel.

Stress is a user-space application that can spawn any number of worker threads that performs useless tasks with the only purpose of submitting the system on which it runs to heavy load. Worker threads can be of four different types:

- **CPU Worker**. Imposes general processor load by calculating the square root of randomly selected numbers.
- **VM Worker**. Stresses the dynamic memory functionality of Linux by continuously allocating and deallocating memory by using the system calls `malloc` and `free`, and touching all pages of the allocated memory.
- **I/O Worker**. Spins on the `sync` system call which forces the kernel to flush all cached data that is to be written to a file system, thereby increasing the I/O intensity.
- **HDD Worker**. Writes random data to a file in permanent storage using the `write` system call and then deleting that same file using the `unlink` system call, over and over again.

Additional options include the possibility to specify the amount of memory to allocate in each cycle and how large files to create before removing them and starting over.

4.3.2 Xload

This report aims to present a fair view of the real-time performance of different Linux based alternatives. While Stress will introduce the Linux kernel to a heavy load, thus providing an equal playing field for the mainline kernel and the rt-patched kernel, Xenomai will not suffer as much from this load. Since Xenomai has precedence over the Linux kernel, the load introduced by Stress is on a separate system. One could argue that this after all is the whole idea behind using a micro-kernel approach. However, the micro-kernel is intended to remove the indeterministic element of the general purpose operating system, it should still be able to handle the load of many concurrent real-time tasks.

Attempting to level the playing field, this report proposes a new application called Xload. The purpose of this application is to introduce load to the Xenomai system, by creating a number of real-time tasks that will stress the scheduling and preemption performance of Xenomai. Its implementation is strongly inspired by Stress. The difference is that Xload only offers one type of worker thread, namely the CPU worker. When the application is launched, it will register a configureable number of workers with Xenomai that will calculate the square roots of random numbers.

Kapitel 5

Preparing the Target System

Before the tests can be carried out, the system on which to perform the measurements has to be prepared with the appropriate software. This system is known as the *target system*, or simply the target. The system from which development is carried out is referred to as the *host*. This process involves setting up an environment capable of building the required software and acquiring the correct versions of all software components. Before compilation, the software must be configured in a proper manner and finally the software must be installed on the target system.

5.1 Hardware

Embedded systems come in many different shapes, but there are a few architectures that are more prominent. Those are the PowerPC and the ARM architecture. Since the ARM architecture is the most relevant to Combitech at this point in time, the tests presented in this report will be executed on an ARM-based system.

More precisely on a product called CM-X270, manufactured by a company called CompuLab. The CM-X270 is a *Computer on Module* (CoM). What that means is, that it combines some processing unit with volatile and permanent storage and external communication interfaces on one circuit board. Specifications vary, but typically the external communications interfaces includes ethernet, USB (host and device), LCD controller, RS232 serial ports and other serial interfaces found in embedded applications such as Inter Integrated Circuit (I^2C) and Serial Peripheral Interface (SPI) buses. All of these features are available on the CM-X270, moreover, by connecting the CM-X270 to an expansion board called SB-X270 another ethernet interface is available along with connectors to standard PC peripherals such as USB keyboards, flat screens, RJ45 ethernet cables and RS232 Sub-D 9 cables. Using the extension board also provides easier access to the general purpose I/O pins available on the CM-X270. The target system in this case, consists of a CM-X270 CoM connected to a SB-X270 base board. Table 5.1 lists the relevant specifications on the hardware:

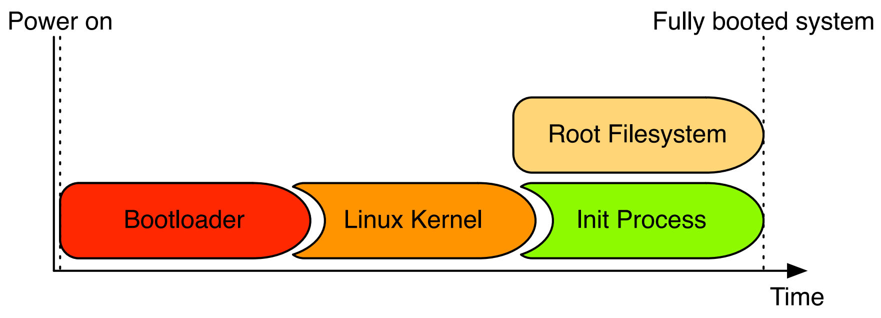
Component	**Specification**
Micro Controller	Intel PXA270
Processor	ARM5TE Compatible Intel XScale running at 520MHz*
Volatile Memory	128MB of SDRAM running at 208MHz*
Permanent Memory	4MB of NOR Flash 512MB of NAND Flash
Video Capabilities	PXA270 LCD Controller* Intel 2000BG Graphics Accelerator
Serial Interfaces	3 UARTs* USB Host and Device* I^2C* SPI*

* Marks features provided by the PXA270.

Tabell 5.1: *Relevant specifications of the CM-X270.*

As shown by Table 5.1, much of the required features are provided by the PXA270 chip except for storage capabilities. The NOR flash is connected directly to the processors local bus and can be accessed just like any RAM memory. This feature makes it perfect for storing the bootloader since code can be executed directly from within the memory. The NAND flash requires a bit more work to set up, but once that has been accomplished, it is faster than its NOR cousin. The CM-X270 is very well supported when it comes to Linux, as we will see later on. This in combination with the fact that the hardware was already available at Combitech made the choice of test platform an easy one.

5.2 Building GNU/Linux



Figur 5.1: *Components that make up a GNU/Linux system.*

In order to build a complete GNU/Linux system there are many components that must be built and put together in a working harmony. Linux is said to have a *monolithic kernel*, which means that the kernel is one piece of static binary code. It is possible to extend the kernel through *kernel modules* which can be compiled after the original kernel is compiled. However, a module is always compiled against a specific kernel version. This restriction is imposed to prevent the module from calling functions that might have been removed or changed in the kernel since the module was compiled.

Before the kernel can start working, it requires that some core hardware have been setup properly. This is the job of the *bootloader*. When the power button is pushed, a system defaults into a fragile state. Little is known about the surrounding environment. One thing that is known is a specific address from where the first instruction should be fetched. The code found at this address is usually what is known as the bootloader, or possibly a small piece of code instructing the CPU where to find the bootloader. For a bootloader to be compatible with Linux, it must ensure that the system's volatile memory has been properly configured as Linux assumes it to be done. Bootloaders on embedded platforms might also provide additional services such as downloading an operating system kernel to permanent storage.

When control is handed over from the bootloader to the kernel, Linux will continue to enable the rest of the system's hardware and load the available drivers. At this point, only drivers that have been compiled into the kernel can be loaded. After all initial configuration have been carried out, the kernel will try to mount a *root filesystem*, or *rootfs*. The rootfs is the first filesystem mounted, from which the kernel expects to find system critical resources. Most importantly, the kernel will try to launch the *init process* from this filesystem. The init process is the parent of all other processes and will usually initialize the user-space into an environment from which other processes can be started, in other words a fully initialized GNU/Linux system. Figure 5.1 shows how all these components work together during the startup of a system.

5.2.1 Acquiring the Source Code

In the open source world of Linux, the standard way of acquiring software is to download the source code and then compile that source code on the local machine. Working on source code as opposed to pre-compiled software has several advantages. It allows one the opportunity to remove components that are not needed, thus creating a smaller end result. When compiling the Linux kernel for example, there is a lot of code that is architecture specific, having the source code allows for all the unrelated code to be purged. Another advantage is the ability to modify the code, either by hand or by applying so called *patches*. In UNIX systems, a patch is a file containing information about changes that have been made to some file or set of files. Using the `patch` application, a patch can be applied to a file or set of files, automatically performing the same modifications that where made by the author of the patch.

The Linux kernel used to be stored in different locations depending on which architecture you wanted to build it for. Nowadays however, all architectures have been consolidated into one kernel tree available from `kernel.org`. This used to be the home of the x86 branch of the kernel. This archive holds all versions of the Linux kernel dating back to 1.0, and it is mirrored on many servers across the globe. The source can be downloaded over http, ftp or maintained via the git revision control application. In this case version 2.6.25.8 was downloaded using http. Three copies was made of this source code, one for each system to be tested.

Development of the PREEMPT_RT patch can be monitored from the project's homepage at `http://rt.wiki.kernel.org`, and the source code can be downloaded from `http://www.kernel.org/pub/linux/kernel/projects/rt/`. At the start of this project, the most recent patch available was patch-2.6.25.8-rt7, which has now been moved to the `older` directory. This patch was then applied to one of the Linux kernel directories.

Xenomai has its home at `http://www.xenomai.org`, and the source code is available at `http://download.gna.org/xenomai/stable/`. Version 2.4.4 was the most recent version available at the time and was therefore chosen. Installing Xenomai requires more configuration than the rt-patch, but the installation instructions in the downloaded archive makes the process go smoothly.

5.2.2 The GNU Toolchain

Before compiling the code, one need to have a compiler and the surrounding tools, assembler, linker , etc. All of these tools put together are known as a *toolchain*. In theory, it should be possible to build the Linux kernel using any toolchain that has support for the C language and the relevant assembly language. By far the most common toolchain used to build Linux is the GNU Toolchain. This toolchain is built from free software, just like Linux. It was chosen to compile the test candidates for several reasons:

- The Linux kernel developers use it, therefore it is known to build the Linux kernel successfully.
- Most people use it, thus it would be easier to get help if problems arose.
- The GNU Toolchain can be downloaded free of charge.
- The PREEMPT_RT patch uses features that requires the GNU C Compiler (GCC).

Working with embedded systems usually brings another dimension of difficulty in compiling software. In a non-embedded situation, you usually compile code to run on the same architecture as the one your compiler is running on. In embedded development however, it is often the case that you cannot run the compiler on the target's architecture. In this case, there is a need for a *cross-compiler*, a compiler running on one architecture, building code compatible with another one. A better word would be a *cross-toolchain* however, since it is not only the compiler, but all the tools that need to support this feature.

The GNU Toolchain is distributed as three separate components called GCC, Binutils and Glibc. GCC is the compiler itself and Binutils provides the other important tools such as the linker and the assembler. In order to build a Linux kernel, only these two components are needed. Glibc is the GNU C Library, this is the library used by all Linux applications written in C. Since the kernel does not use Glibc, it is not required to build it. The motivation for separating these components is to maintain flexibility. On embedded systems for example, Glibc is often replaced by other libraries that leave a smaller memory and size foot-print. In this case however, there was enough memory and space available to use Glibc.

One way to get a hold of a cross-toolchain is to download these components and build it yourself. As this process can be quite hard to perfect, some companies and organizations offer pre-compiled cross-toolchains, some are free and some are commercial products. A third way is to use OpenEmbedded, which will use a script to build a cross-toolchain suited for your host and target combination. Using OpenEmbedded brings other advantages with it, which is the reason it was used in this project. For more information see Section 5.2.5. The executable files of the cross-toolchain are usually prefixed with a name relaying their function. Often the architecture and operating system for which they build are used. In this case for example, the cross-compiler would be called `arm-linux-gcc`.

5.2.3 Configuring the Kernel

The Linux kernel provides several interfaces to configure the kernel, ranging from a question-based command line utility to a graphical selection system. From the chosen configuration tool, you have the ability to streamline the kernel to only contain the desired features and modules. In order to ease the process of configuring, default configurations are available for different types of systems. The CM-X270 board has good support in the Linux kernel and a default configuration is also provided which was used as a base configuration for all three kernels. To select a default configuration, one simply issues the following command from the root of the kernel source code:

```
$ make ARCH=arm cm_x270_defconfig
```

Make is a tool that automates the building of source code by means of so called *makefiles*, which are scripts telling `make` how to use a toolchain to build a piece of software. Makefiles can have multiple *targets*, different recipes of what to build. In this case we want to use the `cm_x270_defconfig` target. Default configurations are stored in architecture specific catalogues, thus we set the environment variable `ARCH` to arm, since we are building for an ARM target. To enter the graphical kernel configurator to make additional changes we use the target `xconfig`:

```
$ make ARCH=arm xconfig
```

Depending on which kernel we are configuring, we select to build the Xenomai or PREEMPT_RT related components into the kernel. But apart from that, all three kernels will incorporate the same drivers and other modules. To build the kernel, we use the make target zImage:

```
$ make ARCH=arm CROSS_COMPILE=arm-angstrom-linux-gnueabi- zImage
```

Previously used make targets did not actually build any code. The zImage target will do that however, therefore we need to instruct `make` not to use the standard toolchain, but to use our cross-toolchain. That is why the environment variable `CROSS_COMPILE` is set to the prefix given to the cross-toolchain. This prefix will be added to all commands in the makefile so that the output will be ARM-compatible code.

5.2.4 Root File Systems

The root filesystem is the first filesystem mounted by the Linux kernel. On a GNU/Linux system, this is where Linux will attempt to start the first user-space process from, the init process. Libraries used by applications that run on this system will be expected to exist on the rootfs as well. Normally, this is also where many of the applications that make up a GNU/Linux operating system would live. These include programs for manipulating files, configuring the system and so on. The rootfs is usually also the home of a system's device nodes. A device node is a file that does not point to some data stored on disk, but to a physical or virtual device. More exactly, it points to some kernel module that, in turn, communicates with the device.

A root filesystem can have many forms. In production environments, the rootfs is often available from some permanent storage connected to the system, a hard disk drive or a flash memory for example. During development however, it might be cumbersome to use this approach since the filesystem might be updated very often, or you may not have a working driver for the storage device. In this case, Linux offers the capability of mounting the root filesystem from a remote machine using the Network FileSystem (NFS). This allows a developer to copy files from and to the target's filesystem on his or her local machine and it also increases the available amount of space on the target filesystem since this could be quite limited on the on-board storage. The intention was for this project to use the NFS approach initially and then transition to a NAND Flash based rootfs. Attempts where made with mounting an NFS share as the rootfs, but they never succeeded. Since the kernel offered a working NAND Flash driver for the CM-X270, the flash disk was used from the start.

5.2.5 OpenEmbedded

There is a wide variety of applications available for GNU/Linux systems. Many of them are developed by people who do not prioritize the ability to cross-compile their software. If this feature is not built into the programs makefile from the start, cross-compiling larger projects might be difficult and could involve quite a lot of manual editing of the makefile. OpenEmbedded (OE) attempts to ease this hassle by providing a set of scripts that will automatically download and build applications for a selected system type.

OpenEmbedded has its home on the web at `http://www.openembedded.org`. Just like the Linux kernel developers, OE uses the git versioning system, and anyone can checkout their own version of OE from `git://git.openembedded.net/openembedded`. In order for OE to build working software for your target you need to setup a configuration file telling OE which processor and machine you are running. Compulab have already added support for their board in OE, so in this case, OE basically just needs to know that we are building for a CM-X270 board.

Building software is then as easy as invoking `bitbake` and giving it the name of the build script as an argument. Bitbake is the application used by OE to interpret the build scripts. These scripts include instructions on how to download source code, apply patches, how to construct the software and so on. In way it is similar to make, but working on a higher level, using make to perform the automation related to the actual compilation of the source code. To build the cross-toolchain, one simply navigates to the OE root and invokes `bitbake` in the following manner:

```
$ bitbake gcc-cross
```

5.3 Running GNU/Linux

Once all the kernels where constructed, it was time to get them running on the target. The CM-X270 uses a bootloader called ARMmon. It is developed by Compulab as closed source software and it is compatible with both Windows CE and GNU/Linux. ARMmon allows kernel images to be downloaded over the Trivial File Transfer Protocol (TFTP). In order for that to work, the machine used for development and compilation was setup as a TFTP server, serving a catalogue to the network containing the kernel images. To download a kernel to the target the following command was sent to ARMmon:

```
ARMmon > download kernel tftp kernel-image server-ip
```

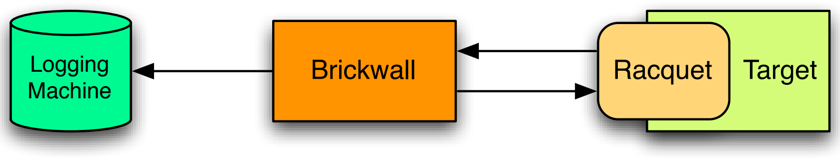
Communication with ARMmon took place through a Universal Asynchronous Receiver/Transmitter (UART). In order to format the NAND Flash and install a working rootfs, a USB memory was prepared with a working filesystem and tools to format the NAND Flash. At first the USB stick was mounted as the rootfs. After the NAND Flash had been formatted and a Journaling Flash FileSystem 2 (JFFS2) image of the rootfs copied to it, the NAND Flash partition containing the JFFS2 image was mounted as the target's root filesystem. The same rootfs was used in combination with all three kernels. To switch kernel, the target was reset after which another kernel was downloaded by altering the `kernel-image` in the line above. To boot the kernel, ARMmon was given the following command:

```
ARMmon > bootos "root=/dev/mtdblock1 rootfstype=jffs2 console=tty1"
```

`bootos` tells ARMmon to boot the recently downloaded kernel. The string within the quotes are boot commands to the Linux kernel. The `root` parameter tells the kernel from which device it should try to find a rootfs. In this case it points to the first Media Technology Device (MTD) partition. MTD is the name of the Linux subsystem in charge of handling flash based storage devices. `rootfstype` specifies the type of filesystem to expect. `console` tells the kernel where to display messages from the kernel, setting this to tty1 will cause these messages to appear on the screen connected to the CM-X270.

Kapitel 6

The Tennis Test Tool

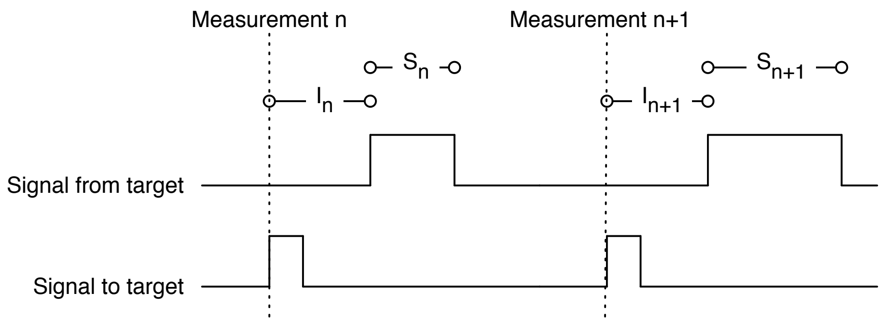


Figur 6.1: *Components that make up the Tennis Test Tool.*

The name Tennis comes from the abstraction of this system to a person using a tennis racquet to bounce a tennis ball against a brick wall as fast as he or she possibly can. The person with the racquet in this abstraction corresponds to a loadable kernel module whose job is to respond to an incoming signal as quickly as possible. The brick wall is represented by some hardware outside of the system that can measure the time between two bounces, i.e. how long it takes for the target system to respond to the signal. Finally, though there is no analogy to this in the abstraction, a machine is needed to permanently store measurements produced by the brick wall. An overview of the layout is presented in Figure 6.1.

Imagine that the brick wall component was to send a signal to the target system, and at the same time store the current time. The kernel module in the target system would react to that signal and send an answer as quickly as possible. Afterwards it would schedule a task to be run and then return. This would mean that the interrupt-processing latency would essentially be zero. Then, as soon as the scheduled task was run it would send another signal to the brick wall component. Additionally, the brick wall component was configured in such a way that it had the possibility to store the time when the two signals sent by the kernel module were received.

With a system like this it would be possible to measure both the interrupt latency and the scheduling latency. The interrupt latency, I, would be equivalent to subtracting the time stored when sending the signal from the brick wall component, from the time when the first signal is received. Following the same reasoning, the scheduling latency, S, is equal to the difference between the two returned signals. Figure 6.2 illustrates the intended signals. The time from when the outgoing signal is sent to when the second incoming signal is received corresponds to the interrupt-to-process latency, but this time always depends on how much interrupt processing that has to be done. But the interrupt processing time will vary depending on the device driver's implementation, not on the kernel in general. Therefore the interrupt-to-process latency for this particular driver says nothing about the kernel on which it runs and will not be analyzed. Since jitter is just the difference between the slowest and fastest response time that can also be calculated from the collected data.



Figur 6.2: *Signals sent and received by the Tennis Test Tool.*

6.1 The Brickwall Module

Brickwall is the collective name for the micro controller responsible for measuring the response times of the target system, together with the software running on that micro controller. It works by first storing the current value of a hardware timer in the micro controller. Then it generates a short pulse on an output I/O-pin. The response line from the target is connected to two different input I/O-pins on the brickwall micro controller. One of them is configured to store the current value of a timer on low-to-high transition. The second pin will store the value of a timer and generate an interrupt on a high-to-low transition. When the interrupt service routine is entered, one measurement cycle is completed. The resulting response times are calculated and sent to a remote system via a UART.

6.1.1 Hardware

From the layout of the Tennis system it was clear a suitable micro controller needed to be found. Four distinct requirements of the hardware was set to ensure a smooth development process. For one, there had to be an on-chip UART built present for communication with the logging computer. Secondly there had to be a hardware timer available to perform precise time measurements. Additionally, the controller had to be able to drive the timer at a frequency high enough to produce an acceptable resolution on the timer. Initial research suggested that response times on these types of systems are in the low microsecond range. With that in mind it was decided that Brickwall should be able to maintain a resolution of at least $1\mu s$. By the Nyquist theorem we deduce that to distinguish two signals with a resolution of $1\mu s$ we need to sample that signal at least twice as often. This means that the mechanism sampling the Brickwall input signal must run at a frequency higher than 2MHz. Finally, the host system had to be able to compile software, debug and transfer it to the device.

The initial plan was to use an Atmel AVR micro controller. AVR is a family of 8-bit micro controllers with built-in hardware timers and UART which are capable of running at speeds up to 20MHz[1]. The reason was that the author has previous experiences with this family of products and knew for a fact that a working toolchain, debugger and programmer existed for these devices that ran under Ubuntu GNU/Linux, the operating system under which development would take place. That meant that it fulfilled all the requirements of the hardware. Therefore possibilities to acquire such a micro controller at the Combitech office where pursued. The result was that no such hardware was available.

This brought the project to a cross-road. One option was to order a development board for an Atmel AVR. The other one was to use prototype board designed for a different project based on a Philips LPC2138 micro controller. Some research showed the prototype board to be a much more qualified candidate for the job.

The Philips LPC2138 is based around an ARM7 32-bit processor, capable of running at speeds up to 60MHz. Perhaps the most important difference between the two micro controllers was the features of the built-in timers. The Atmel AVRs' timers have 16-bit counter registers at the most[1], meaning that pulses over 2^{16}timer ticks cannot be measured. Even when using a crystal with a frequency of 4MHz, the maximum pulse would be 16.384ms. It would be possible to measure longer pulses by using a prescaler, meaning that the timer count is only incremented every forth tick for example, if the prescaler was set to 4. However, this effectively lowers the timer frequency, the result being that the resolution drops with a factor of 4. In contrast, the LPC2138 has two 32-bit counters which results in a maximum measurable pulse of over 71s, using a crystal with a 60MHz frequency.

Moreover, the LPC2138 had 4 input capture channels per timer while the Atmel only had one. An input capture channel is an I/O-pin that can be configured to perform timer related tasks when the level of the pin changes. A minimum of to two channels was needed to be able to measure the pulses accuratly since two signals would be sent back from the target device. Both controllers had built-in UART, so communication with the logging device would not have been a problem whichever path was taken. Research also showed that the GNU Toolchain was available as a cross-compiler for x86 hosts, which would build ARM compatible code. For more information on the GNU Toolchain and cross-compilation, see Section 5.2.2.

The prototype board is equipped with a 14.7456MHz crystal oscillator. Using a phase locked loop (PLL), the oscillator frequency is multiplied by 4 to supply the CPU with a clock frequency of 58.9824MHz. The PLL also outputs a separate pheriphiral clock frequency at 18.432MHz which is used to drive the timer, UART and other pheriphirals. The resulting timer resolution is approximately 108ns, with the ability to measure pulses with a length of almost 4 minutes.

On the LPC2138, the majority of the I/O-pins can be configured for 4 different operation modes, one general purpose I/O mode and three specific functions such as input capture channel or external interrupt pin. For each pin that is configured as an input capture channel to Timer0, the value of the timer will automatically be copied to a register dedicated to this purpose only. Three additional options can be configured for each input capture pin. The first two decides if the capture should be triggered on a low-to-high transition (rising flank), on a high-to-low transition (falling flank) or if, by choosing both described options, it should be triggered by both. The final option decides if a capture should generate an interrupt, allowing some interrupt service routine to be run.

In the Brickwall implementation, two input capture channels have been wired together and connected to the signal cable originating at the target. These are connected using pins 15 and 13 on the LPC2138. One general purpose I/O-pin is used to transport the trigger pulse to the target, pin 14 is used to fulfill this purpose. Four additional GPIO-pins are used for diagnostic purposes, controlling three LEDs and a buzzer already present on the prototype board. A yellow LED is used to indicate communication between Brickwall and the target, this is connected to pin 55. A red LED is used to signal that an error has occurred, connected on pin 53. There is also a green LED that signals the end of a measurement connected on pin 54, together with a buzzer indication connected to pin 3. The UART connects to the logging machine via a 10-pin header connector together with a cable that adapts to a female D-SUB 9 connector. Finally, to program and debug the device, there is a 20-pin JTAG header connector available on the board. For more information on the JTAG interface see Section 6.3.

6.1.2 Software

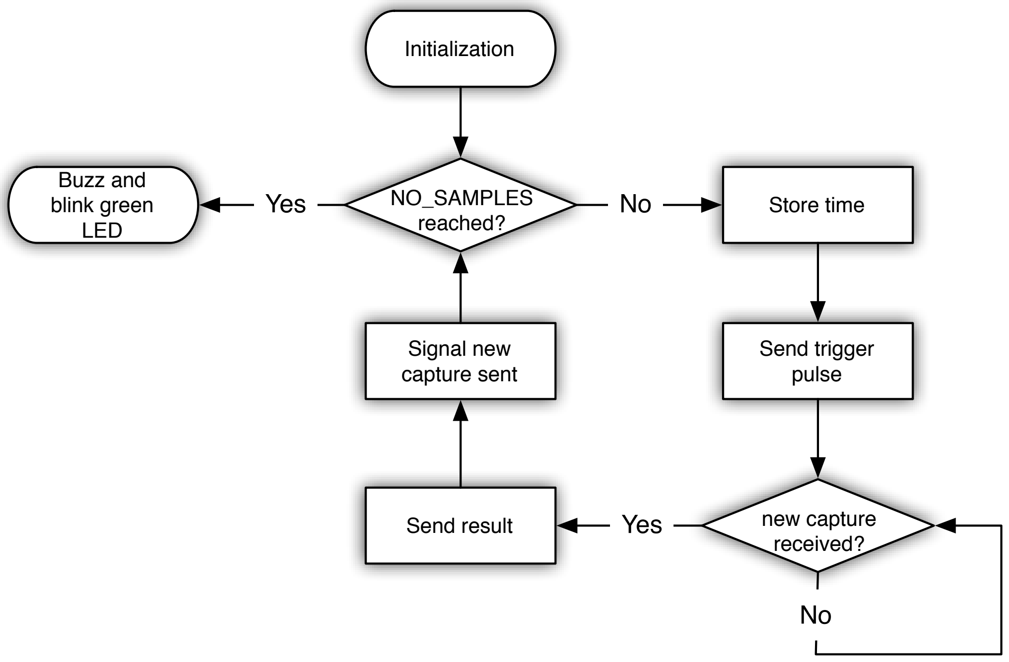
Operating systems are very useful tools when dealing with complex computing systems. If one is trying to use a system solely for the purpose of measuring time however, there are a number of disadvantages to using an operating system. An application may not be allowed direct access to hardware timers, the operating system may preempt the application to serve some other service resulting in unexpected latencies, etc. This is why the Brickwall software was designed to run directly on the hardware without an in between system. However, with more power comes more responsibility. Tasks that are usually carried out by the operating system at startup must now be taken care of by the application. These tasks mainly consists of setting up the processor registers to some sane initial values, initializing important peripherals and setting up a call stack to be able to run C code. Due to the fact that there is no context for running C code, this has to be written in assembler.

As a developer, you want to focus on the application code and not spend time bringing the hardware up and running. Philips provides application examples complete with startup assembly code but unfortunately these examples are made for proprietary toolchains. Fortunately, a man by the name of Martin Thomas has adapted these examples to work with the GNU toolchain and made these available at his website[23]. By using this startup code to perform the tasks recently discussed, development of the actual Brickwall code could begin almost instantly.

The Brickwall software does not contain a lot of code, 166 lines in total excluding empty lines. This is because the hardware has built-in support for many of the time-keeping features that are used in this application. The first procedure to be run is `main`, where all application specific initialization is done. This involves setting up the UART to operate at a compatible baud rate. Brickwall sets up the UART with a baud rate of 115200kbps, using 8 data bits, one stop bit and no parity. Next, the output directed pins are setup. This includes the three LEDs, the buzzer and the pin to be used to generate a pulse to send to the target, also called the `PING_PIN`. The final initialization step is setting up Timer0, which is the timer used for measuring the pulses returned by the target.

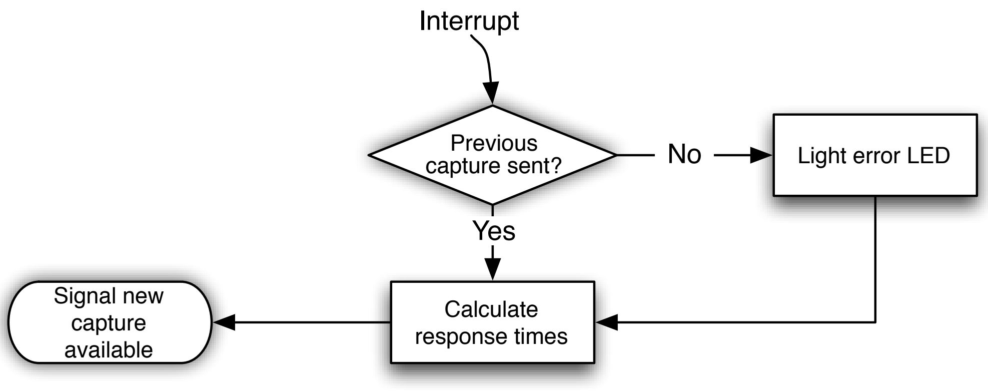
As mentioned previously, most I/O-pins can be configured to serve four different purposes and most are configured as GPIO-pins at startup. Therefore, pins 13 and 15 must first be configured. This is done by writing a value corresponding to the input capture channel function in the Pin function Select register, to the bits associated with pins 13 and 15. Then the Prescaler register is assigned the value 0. This means that no prescaling will take place and that the maximum resolution of the timer will be maintained. Next it is time to configure the input capture channel behavior. The behavior is controlled by the Capture Control Register where each tupel of three bits control a different channel. The first channel is configured to store the value of the Timer Counter Register into Capture Register 0 on each rising flank, but not on a falling flank and no interrupt should be generated. The third channel is set up to store the value of the Timer Counter Register into Capture Register 2 on each falling flank, but never on a rising flank, additionally an interrupt should be generated upon seeing a high-to-low transition. The final step of the timer initialization is to register an interrupt handler in such a way that the `timer_handler` routine is called to service the interrupt.

Now all the necessary initialization is done and the system is ready to start measuring. First off we have to store the current timer value and then generate a pulse that the target can reply to. This is done by first storing the value of the Timer Counter Register to the variable `capture_start`. Then, by setting the `PING_PIN` high, running one cycle in an empty for statement to pass some time and then setting the `PING_PIN` low again, generate a pulse of approximately $1\mu s$. Using a software implementation to first store the timer value and then changing the level of the `PING_PIN` introduces some extra delay in each measurement. This delay is however both short and constant, for details see Section 6.1.3. Furthermore, in the first implementation of Brickwall, the level of the `PING_PIN` was toggled on each iteration. Respectively, the target system was configured to trigger on both rising and falling flanks as oppose to only rising in the current version. Both solutions performed equally well, the latter was chosen in order to verify the systems performace against a known signal source, for more information see Section 6.1.3. After the trigger pulse has been sent, the `main` function will wait in a busy loop, for the variable `new_capture` to become non-zero. A non-zero value signals that a rising and a falling flank has been seen on the input capture channels and that the time of their arrivals relative to the rising flank of the trigger pulse has been recorded. After seeing a non-zero value of `new_capture`, the captured response times are sent to the logging computer using the `uprintresult` procedure. `uprintresult` will convert the response times to strings using a slightly modified version of the int-to-ascii implementation from The C Programming Language, Second Edition[16]. It will then send the resulting strings over the UART by writing each character to the UART0 Transmit Holding Register. Finally, the `new_capture` variable will be assigned the value 0, which represents that the recently acquired capture has been sent to the logging computer. This marks the end of a measurement cycle. The number of iterations to cycle through is determined by a `#define` called `NO_CAPTURES`, which in the current implementation is set to ten million samples. After the measurement is done, the buzzer will ring for about a second after which the green LED will start blinking indefinitely until the micro controller is reset. For an overview of the program flow, see Figure 6.3.



Figur 6.3: *main procedure flow chart.*

From the program flow of the `main` procedure it follows that some function or procedure must be run asynchronously in order for `main` to escape the busy loop. This is achieved via the timer interrupt. The interrupt service routine is invoked upon receiving a falling flank on input capture channel three. The first task that is carried out, is to make sure that the previously captured latencies have been transmitted to the logging computer, i.e. that `new_capture` is zero. If this is not the case, the red error LED will be lit. It is important to note that the error LED is never switched off once it has been lit which ensures that an error will not go unnoticed. There is no attempt to recover from this error other than to make sure that it is detected. The reason is that it should never occur since a new trigger pulse is not sent to the target until the `uprintresult` procedure has completed execution at which time the data has been sent. Thus, if the `timer_handler` routine is called when `new_capture` is zero, it means that there is some external communication error with the target device. In such a situation the entire measurement session is considered tainted, and must be discarded. After this sanity check, the resulting response times are calculated using the two values stored in the Capture Registers together with the value of `capture_start`. Since a new capture is available at this time, the `new_capture` is assigned the value 1 to reflect this, allowing the `main` procedure to escape the busy-wait loop. A flow chart of the `timer_handler` interrupt service routine can bee viewed in Figure 6.4.



Figur 6.4: *timer_ handler interrupt service routine flow chart.*

6.1.3 Verification of functionality

When designing a test tool, it would be irresponsible to assume its correctness just because it is performing the way one would expect. That is to say there could be some unexpected behavior on the target system which, together with some defect of the test tool produces reasonable, but ultimately incorrect data. To insure the functionality of Brickwall, so called black-box testing was used. What that means is that the internal implementation of

Brickwall is ignored in the verification. Instead, using a known input, the resulting output is compared to the expected result. Since nothing is assumed about the internal structure of Brickwall, the same verification method can be used if some part of the implementation changed. The opposite method is called white-box testing, in which a system is verified by measuring its reactions at strategic points. The black-box method was chosen since the implementation of Brickwall was likely to change due to its short age and therefore, the test would have had to change with it using a white-box approach.

In practice the verification was accomplished using two tools that were assumed to be functioning properly. The first one was a Hewlett-Packard 33120A 15MHz Arbitrary Function Generator. This function generator has the ability to produce very complicated waveforms, to verify Brickwall however a simple square wave was sufficient. By varying the frequency and duty cycle of the output signal of the function generator, a response from the target system could be simulated. In order for Brickwall to measure this signal however, it had to control when a new pulse from the function generator was to be sent. Thankfully the 33120A has support for an external trigger. It has the ability to generate a specified number of arbitrary waveforms on the rising edge of a TTL pulse[12]. Because of this mode of operation, the implementation of brickwall had to be changed. As discussed in Section 6.1.2, a change in logic level was originally used to trigger the target system. However, to be able to run exactly the same code when verifying Brickwall as when performing response time measurements, the implementation was changed to only use the rising flank as the trigger.

By connecting the `PING_PIN` to the external trigger of the waveform generator, Brickwall could trigger a response from the device at will. The function generator was configured to produce a single square wave on each trigger, which would represent the simulated response from the target device. The output of the function generator was connected to the input capture channels of Brickwall so that the response could be measured. In order to make sure that the function generator did not introduce any internal delays or other errors, the verification was monitored by the second tool. This device was an Agilent 54622D 100MHz Mixed Signal Oscilloscope. Connecting one of the oscilloscopes input channels to monitor the `PING_PIN`'s line and the other to the input capture channels' line, the response times could be measured manually. The resulting values was compared to the output produced by

Brickwall. Table 6.1 shows the simulated interrupt and scheduling latencies as measured by the oscilloscope in contrast to the output produced by Brickwall. The interrupt latency corresponds to the time when the rising flank is received from the function generator and the scheduling latency to the time between the rising and the falling flank.

Interrupt Latency (μs)			Scheduling Latency (μs)		
Oscilloscope	Brickwall	Error	Oscilloscope	Brickwall	Error
2.12	2.55	+0.43	1.00	1.46	+0.46
3.00	3.42	+0.42	2.00	2.43	+0.43
4.70	5.10	+0.40	4.04	4.47	+0.43
8.10	8.52	+0.42	8.10	8.56	+0.46
18.3	18.72	+0.42	20.2	20.65	+0.45
35.2	35.64	+0.44	40.5	40.97	+0.47
69.1	69.50	+0.40	80.9	81.5	+0.60
171	171.2	+0.20	202	202.7	+0.70
340	340.3	+0.30	405	405.6	+0.60

Tabell 6.1: *Brickwall output compared to a trusted oscilloscope.*

From Table 6.1 it is observed that Brickwall tend to report response times to be longer than they actually are. It is likely that this offset in time originates from the way the start time of a measurement is stored as discussed in Section 6.1.2. In spite of that, since the offset is small and close to constant over time, Brickwall is considered to produce reliable results.

6.2 The Racquet Module

One of the most prominent reasons for creating Tennis was to create a tool that had as little impact as possible on the system it was to measure. Unfortunately, there has to be some code on the target that can reply to the incoming trigger pulses sent from Brickwall. This is the responsibility of Racquet. Upon receiving a trigger pulse from Brickwall it immediately sends a reply back, thus providing Brickwall with the system's interrupt latency. Then it schedules a task to be run as fast as possible. When this task is run another signal is immediately sent to Brickwall, thus relaying the scheduling latency.

Since this report will try to examine the differences in response times in three different kernels, two different versions of Racquet was built. The first one is used for measuring the response times of the standard kernel and the PREEMPT_RT patched kernel. The Xenomai kernel could also run this code, however to make use of the micro-kernel architecture, another API has to be used.

6.2.1 Standard Module

Since the purpose of this report is to examine response times in relation to communicating with other devices, Racquet had to be implemented as a kernel module. There are several reasons for this. For one, direct access to hardware registers is more straight-forward in kernel-space, though not impossible to implement in user-space[7]. Also, in the effort of acquiring authentic data, the code should be placed at the point where it would be placed in a real situation. In a Linux based system, that means a device driver implemented as a kernel module. Racquet has not been compiled into the static kernel image running on the target. Instead it is dynamically loaded after the system has booted. The reason is that Racquet is not required by the system during the startup phase, therefore development is more efficient using a loadable module. Using that model, it is possible to load the module, test it, unload it, correct some issue and load it again, without the need to recompile the whole kernel or reboot the system.

The Racquet source code is even smaller than the Brickwall code, weighing at 48 lines in total, excluding empty lines. The reason is that the Linux kernel provides functions and macros for registering interrupt handlers, controlling I/O-pins and scheduling kernel tasks, etc.

In the initial implementation of Racquet, a character device was registered in the file system by Racquet that could be opened, closed, written to and read by a user-space application. The idea was that a user-space application would open the device and then try to read data from the device. Racquet was built to use so called blocking I/O, meaning that the read function would not return until an interrupt had been received. So when an interrupt arrived, the first signal was sent back to Brickwall by setting the level of the output line high. The read function would then be allowed to return. The user-space application would then write some data to the device. When the Racquet write function was called, the level of the output pin would be set to low again. Thus, Brickwall could measure the time it took for the Linux kernel to schedule a user-space application. However, the focus of this report is to examine response times in the device driver layer. In Linux, the convention when writing drivers is to split the task of handling an interrupt between a so called top half and bottom half[7]. The top half will be the handler actually responding to the interrupt, doing the most time critical work. The rest of the work is delegated to the bottom half which is scheduled to be run at a later time. In light of this, it seemed more relevant to have Racquet reflect this model.

When a module is dynamically loaded, the kernel will call the function assigned as the initializing function by the `module_init` macro which takes the address of the initializer function as its only argument[7]. If no function has been assigned to this purpose it will call a function called `init_module` by default. The same method is used when removing a module from the kernel. By default, a procedure named `cleanup_module` will be called unless some other procedure has been assigned this task by use of the `module_exit` macro.

In Racquet the `racquet_init` is assigned the task of initializing the driver. The function has three main tasks to perform. It has to configure the two I/O pins used, one to capture the incoming trigger pulse from Brickwall called `RACQUET_IRQ` and the other to send signals back to Brickwall called `RACQUET_REPLY`. Using built-in functions for configuring general purpose I/O-pins present on the PXA270, `RACQUET_IRQ` is configured an input and `RACQUET_REPLY` as an output pin. Secondly, the irq line is configured to be rising-flank sensitive using the `set_irq_type` function. In the third step, the interrupt handler `racquet_irq` has to be registered with the kernel for the relevant interrupt line. Again, Linux provides a function for this purpose called `request_irq`.

If a module is unloaded it is important for the clean-up function of that module to give any resources it has allocated back to the kernel so that they can be assigned to other modules that might need them. This can involve freeing memory used by local data structures, freeing device numbers and so on. The only resource requested by Racquet is the use of the interrupt line connected to `RACQUET_IRQ`. Therefore `racquet_exit` simply gives back the interrupt line to the kernel by calling `free_irq`.

Bottom halves of Linux drivers are usually implemented by the use of so called tasklets[7]. A tasklet is a procedure that can be scheduled by any code in the kernel, it will then be run at a time chosen by the kernel scheduler. A tasklet can be scheduled using two different functions, `tasklet_schedule` and `tasklet_hi_schedule`. The difference between the two, as implied by their names, is that all tasklets scheduled using `tasklet_hi_schedule` will be run before any tasklet scheduled with `tasklet_schedule`. Since the goal is to evaluate the kernels performance in the most demanding situations, Racquet schedules its tasklet with high priority. This is the last action taken in the interrupt handler. Prior to this it only performs one other task which is to set the `RACQUET_REPLY`-pin to a logical one.

At some point in time after scheduling the tasklet, depending on scheduling priorities and policies, the scheduler will execute the tasklet. Once the tasklet is run, it has only one purpose to fulfill and that is to set the level of the `RACQUET_REPLY`-pin to a logical zero. This concludes one cycle in Racquet. Due to the asynchronus structure of Racquet, a flow chart is not ideal to demonstrate the program on a conceptual level. The source code is short enough to serve this purpose and is available in Appendix B.

6.2.2 Xenomai Module

During the design phase of Tennis, one of the biggest issues was how to implement two versions of Racquet in such a way that it was possible to compare them. For all non-real-time critical actions such as initialization and removal of the module, the Xenomai module relies on the standard Linux API. Registering of interrupt handlers are done in a way very similar to the Linux API and were therefore directly portable. All the hardware access could also be achieved via the Linux API. The Xenomai API does not however, provide a tasklet sub-system, which makes it hard to translate the standard module into a Xenomai equivalent. Using the Linux API was not an option in this case since this would mean that the tasklet was scheduled in Linux and not in the Xenomai realm.

Some examining of the Xenomai API Manual produced two possible implementations. The first one was to use the Alarm framework since it was referenced as being the equivalent of RTAI tasklets[26]. It was later discovered that what RTAI refers to as tasklets are the equivalent of the timer framework in Linux. The Xenomai Alarm framework works by first creating an `RT_ALARM` structure containing the address to the handler that should be called when the alarm is triggered. This is done via the `rt_alarm_create`-function. Firing the alarm is done via the `rt_alarm_start`-function, which takes a pointer to the `RT_ALARM` structure to be fired, a relative time in the future when the alarm is to be fired and the interval of firing. Using `TM_INFINITE` as the interval allows the alarm handler to be run only once, therefore by choosing the relative time to be zero, the handler should ideally be run instantly. Unfortunately, the manual did not explicitly state that the alarm handler would be run immediately upon receiving the relative time zero, or wether the scheduler would be invoked. Due to this ambiguity in the manual, this approach was laid aside.

Instead, a combination of the Task and RTDM Event frameworks where used. In the module initialization, an `rtdm_event_t` object is initialized using the `rtdm_event_init`-function. The next step is to create and start a Xenomai task with the highest priority by initializing an `RT_TASK` using the `rt_task_create` and `rt_task_start` functions respectively. I/O-initialization found in the standard kernel module is then reused in the Xenomai module. The last task of the initialization code is to register an interrupt handler. In this case, the interrupt handler is registered with Xenomai instead of Linux using the `rtdm_irq_request` function which registers an `rtdm_irq_t` object with Xenomai containing the address of the interrupt handler `racquet_irq`, the IRQ number of the `RACQUET_IRQ` line and human readable name of the interrupt among other things. When the module is unloaded, allocated resources are given back to the system in reverse order of allocation. First the IRQ line is released by calling `rtdm_irq_free` with the `rtdm_irq_t` object. By calling `rt_task_delete` and `rtdm_event_destroy` with their respective handles, the task and event are removed from the Xenomai system.

The newly started Xenomai task will start by calling `rtdm_event_wait` with the `rtdm_event_t` handle. Calling this function with an event that has not been signaled will cause the task to be blocked. The task will be unblocked by calling `rtdm_event_signal` with the same handle as the argument. This call is made in the interrupt handler after it has set the `RACQUET_REPLY` line to a logical one. When the task is unblocked, it will immediately set the line to a logical zero again. Continuing in this cycle, it will call the `rtdm_event_wait` function again and wait for the next signal.

Although the implementation of this module is somewhat different than the standard module when it comes to measuring scheduling latency, actions have been taken to make them as comparable as possible. The results are two drivers that use different frameworks to implement a design that is conceptually equivalent. However, it should be noted that differences in the measured scheduling latencies could, to some degree, be correlated with the implementation of Racquet.

6.3 Development Environment

During the development phase of embedded software, there are many different components involved. Tools are needed to edit the source code, compile it into machine code, debug it, transfer it to the intended device and so on. Making these tools work together in a seamless fashion saves a lot of time, rather than having to edit the code in one application, compile it in another window and transfer it to the device in a third and so on. This is the purpose of an Integrated Development Environment (IDE).

Tennis was developed using the Eclipse IDE. Originally intended for Java developers, Eclipse now has extensions for other languages. One of the more evolved extensions is called C/C++ Development Tools (CDT) which allows integration with the GNU Toolchain and the GNU DeBugger (GDB). CDT was used in combination with Zylin Embedded CDT, which extends CDT even further, aiming to satisfy the needs of embedded developers by allowing more in depth debug configurations for example.

The extra debug configuration options made available by the Zylin extension come in handy when Brickwall is transfered to the target. The software is downloaded to the target's volatile memory via a *JTAG debugger*. JTAG stands for Joint Test Action Group, originally designed to test the functionality of circuit boards it is now very popular in the embedded world to program and debug embedded systems. A JTAG also provides possibility to set breakpoints in the code, monitor relevant registers and so on, making it a powerful debugging tool on embedded systems.

The JTAG used for programming and debugging Brickwall is called Olimex ARM-USB-OCD. As the name suggests the debugger connects to the computer via USB. Communication with the device is made by connecting to it through GDB. A software called OpenOCD (Open On-Chip Debugger), handles the actual connection to the JTAG and presents an interface to the developer in form of a GDB server. Using this interface, it is possible to issue standard GDB commands just as if debugging on a local machine.

Kapitel 7

Real-Time Measurements

As discussed in Section 4.3, several measurements where carried out on the same test subject while inflicted with different degrees of load. All candidates where included all measurements except for the one which utilizes Xload, since that is only applicable to the Xenomai system. In fact, Xload does not even run on a non-Xenomai target. The different measurements where done in the following manner:

- **Idle Test**. The target was booted with the kernel in question, after which the system was allowed to settle down for 5 minutes. The Racquet module was then inserted by entering:

  ```
  $ insmod racquet.ko
  ```

  Once the module had been loaded, the logging computer was set to record the traffic coming from Brickwall over the serial port in this way:

  ```
  $ cat /dev/ttyS0 > logfile
  ```

  At this point both the target and the logging computer are just waiting for Brickwall to send the first pulse. Therefore, the code is downloaded to the Brickwall hardware via the JTAG and a measurement is initiated.

- **Stress Test**. The Stress Tests where performed in exactly the same way as the Idle Tests, except for the fact that the stress application was launched before the Racquet module was inserted. It was given the following arguments:

```
$ stress --cpu 4 --io 4 --vm 4 --vm-bytes 16M --hdd 4 --hdd-bytes 64M
```

This causes four workers of each type to be dispatched. The custom sizes where added since the default values where larger than the amount available on this platform.

- **Stress and Network Test**. This test is an extension of the Stress Test. In an attempt to increase the interrupt rate on the target, a file containing only zeroes, 64MB in size, was continuously sent to the target by executing this script from the logging machine:

```
#!/bin/bash
LIMIT=1000000
for ((i=1; i <= LIMIT ; i++))
do
    scp zeroes.bin root@target-ip:.
done
```

- **Xload Test**. Identical to the Stress and Network test with the addition of running the Xload application on the target before inserting the Racquet module.

When performing the measurements for the rt-patched kernel, advantage was taken of the fact that the PREEMPT_RT patch provides the possibility to assign custom priorities to interrupt handlers, hard as well as soft interrupts. This was done by using an application called `chrt`, which can change the priority and scheduling policy of a process. Both the interrupt handler registered by the Racquet module and the high priority tasklet soft interrupt handler where bound to the First In First Out (FIFO) scheduling policy and both were assigned the highest priority level, 99.

In order to ensure that no lingering stress was evident on the system, a reboot was made after each measurement had been completed. Each measurement lasted for about two hours depending on the average latency of the system, since the measurement is restricted in the number of iterations and not in time.

The following figures will present the result of the measurements for the different kernels. Data will be presented as histograms. That means that the x-axis will represent response times, and the y-axis will show the number of occurrences of a certain response time. Another possibility would be to plot the response time as a function of the iteration. Histograms where chosen since they give a better sense of distribution between the response times. It was found that all systems have a significant part of their response times concentrated in a narrow range. In order not to loose sight of the surrounding frequencies, the y-axis uses a logarithmic scale. Some figures also use a logarithmic x-axis in order to preserve readability in plots exposing large jitter.

7.1 Xenomai

Table 7.1 shows the maximum response times and the jitter for the respective measurements. All values are in μs. Figures 7.1 through 7.4 provides a graphical representation of the data.

	Idle		Stress		Stress & Network		Xload	
	IRQ	Sched	IRQ	Sched	IRQ	Sched	IRQ	Sched
Latency	28	47	59	68	58	76	247	271
Jitter	23	43	54	64	54	72	242	267

Tabell 7.1: *Summary of Xenomai's results.*

7.2 PREEMPT_RT

Table 7.2 shows the maximum response times and the jitter for the respective measurements. All values are in μs. Figures 7.5 through 7.7 provides a graphical representation of the data.

To track the impact of the custom scheduling applied to the PREEMPT_RT kernel, measurements without the use of `chrt` were also performed. Table 7.3 shows the results produced during these measurements.

	Idle		Stress		Stress & Network	
	IRQ	Sched	IRQ	Sched	IRQ	Sched
Latency	98	97	268	155	273	153
Jitter	86	90	256	148	261	146

Tabell 7.2: *Summary of the rt-patched kernel's results.*

	Idle		Stress		Stress & Network	
	IRQ	Sched	IRQ	Sched	IRQ	Sched
Latency	169	197	653	197	13122	1373
Jitter	157	190	641	190	13110	1366

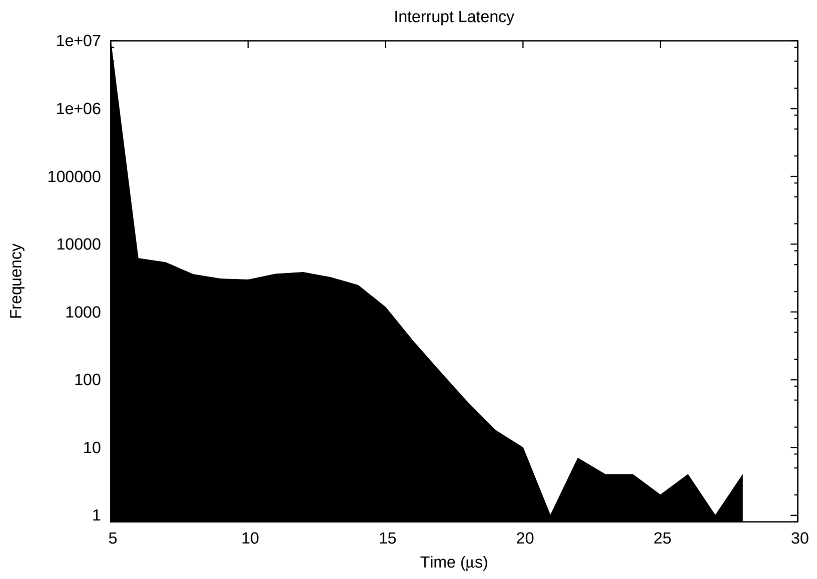
Tabell 7.3: *Summary of the rt-patched kernel's results without the use of chrt.*

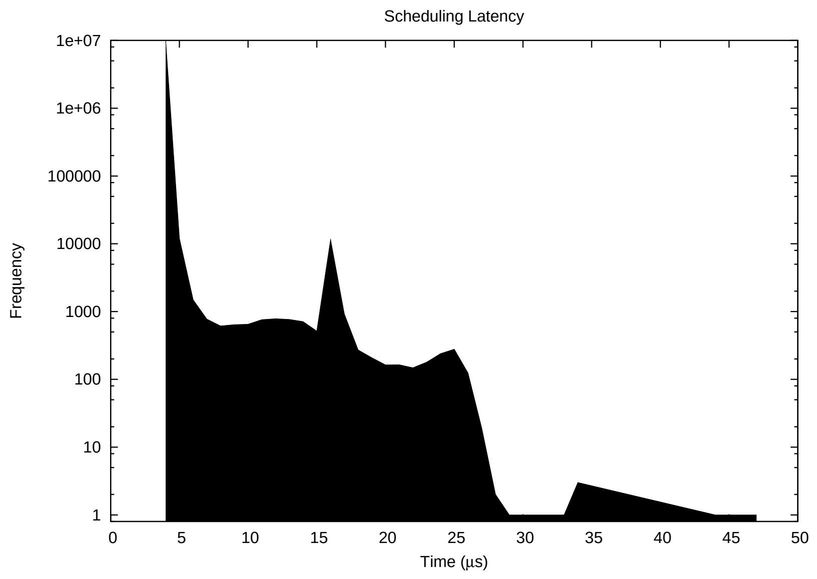
7.3 Mainline Kernel

Table 7.4 shows the maximum response times and the jitter for the respective measurements. All values are in μs. Figures 7.8 through 7.10 provides a graphical representation of the data.

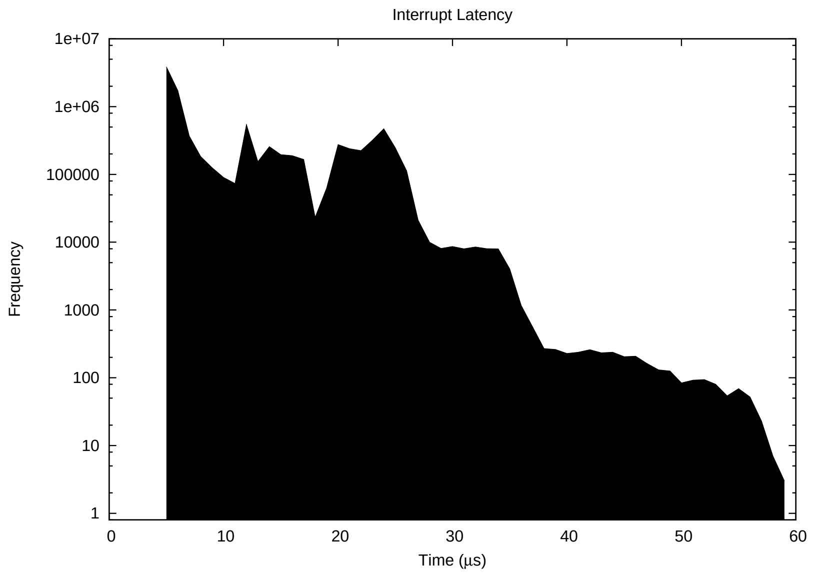
	Idle		Stress		Stress & Network	
	IRQ	Sched	IRQ	Sched	IRQ	Sched
Latency	51	105	17194	100	10705	2694
Jitter	48	101	17191	96	10703	2690

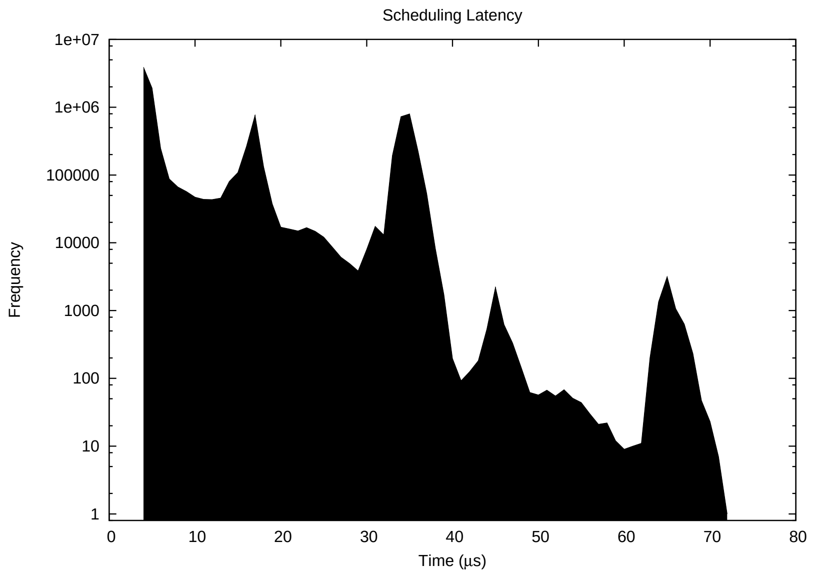
Tabell 7.4: *Summary of the mainline kernel's results.*



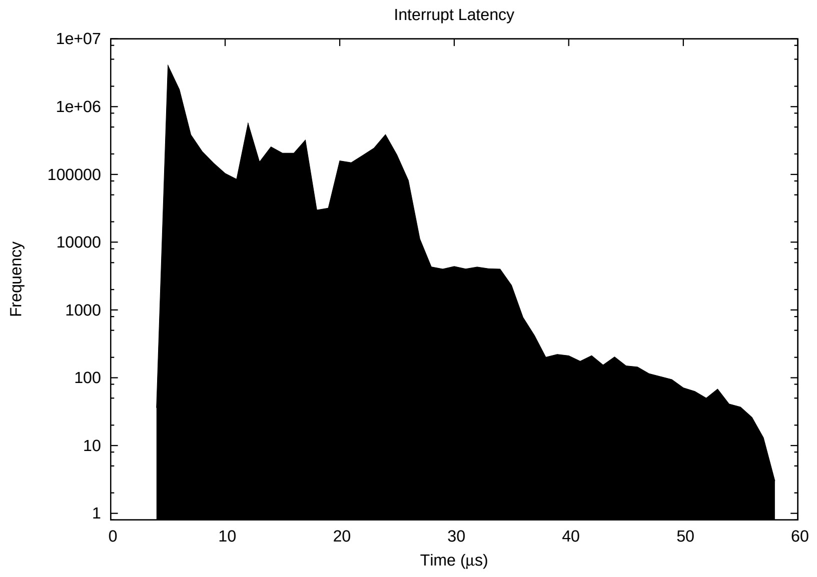


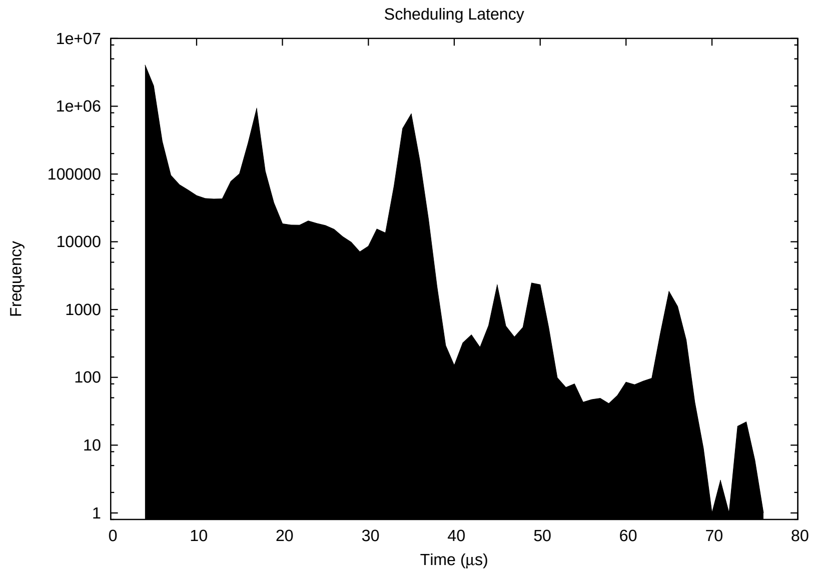
Figur 7.1: *Xenomai kernel results of the idle test.*



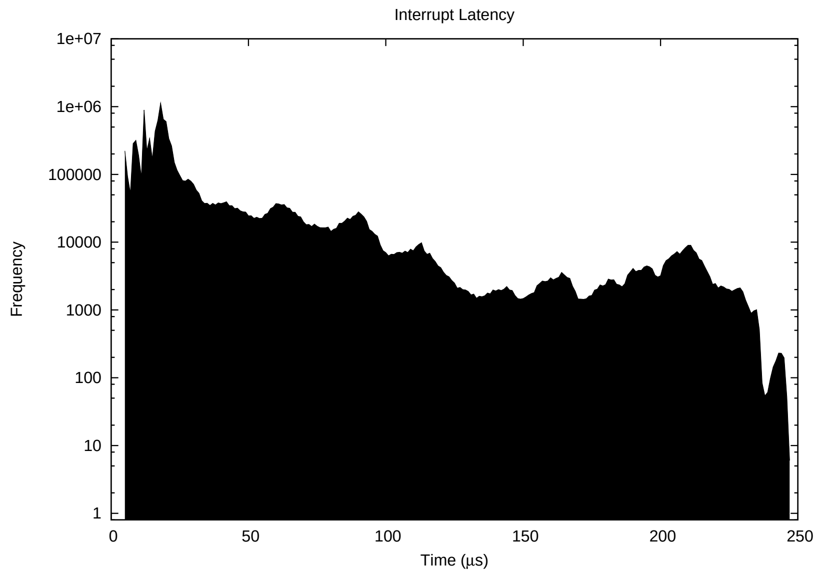


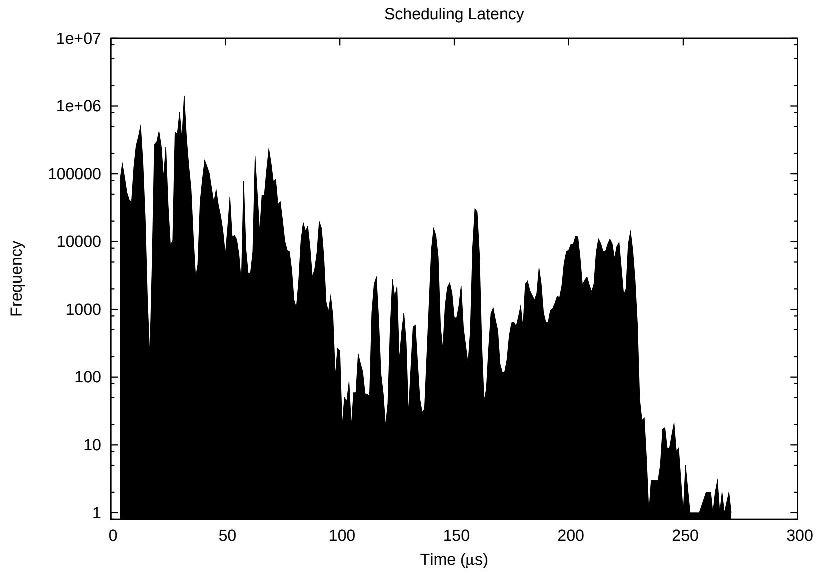
Figur 7.2: *Xenomai kernel results of the stress test.*



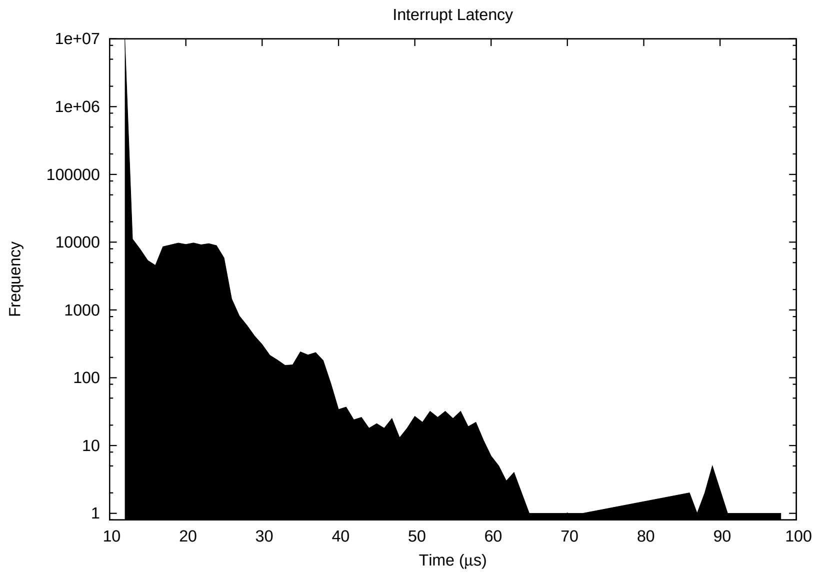


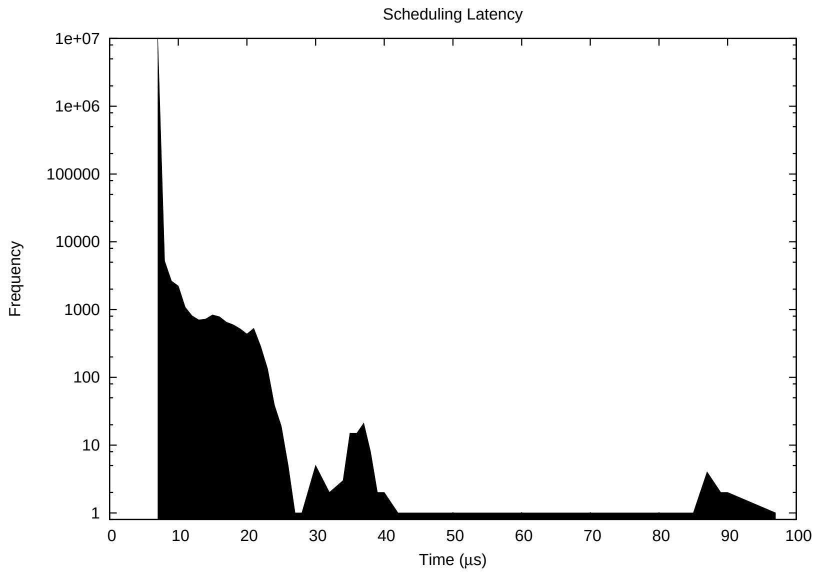
Figur 7.3: *Xenomai kernel results of the stress and network test.*



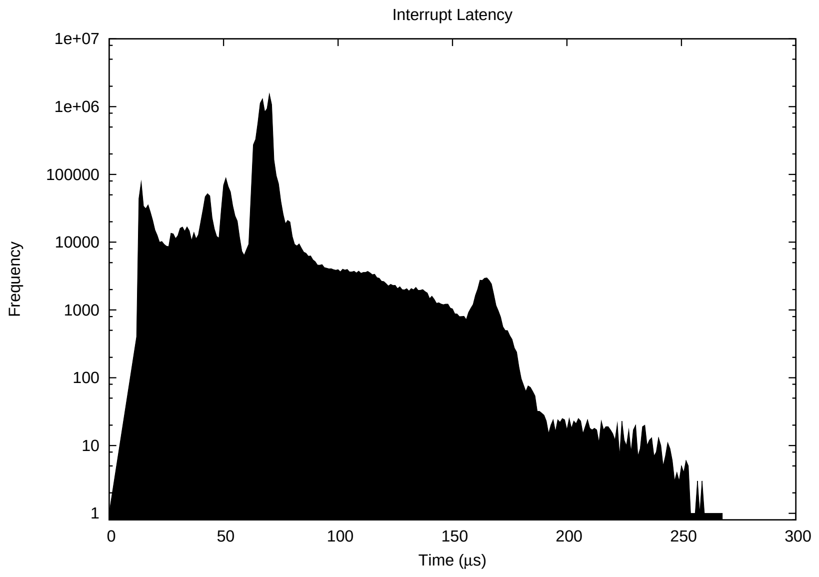


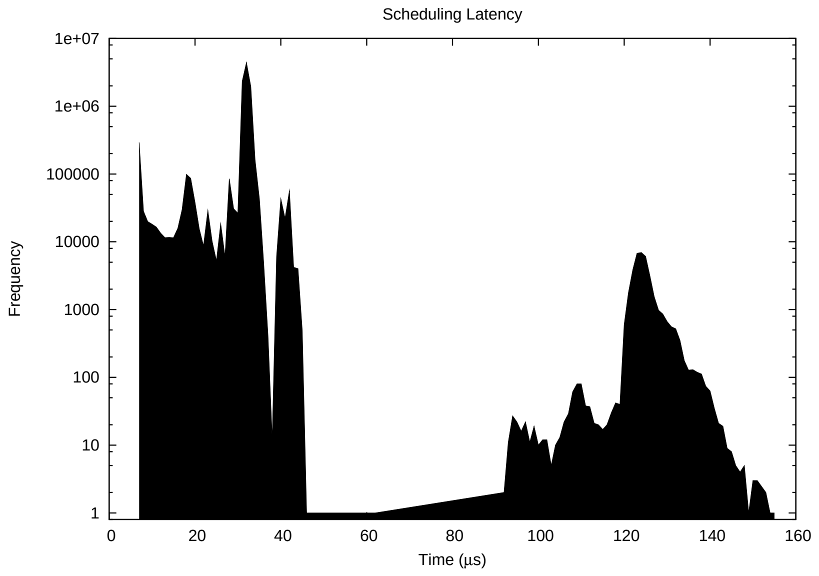
Figur 7.4: *Xenomai kernel results of the Xload test.*



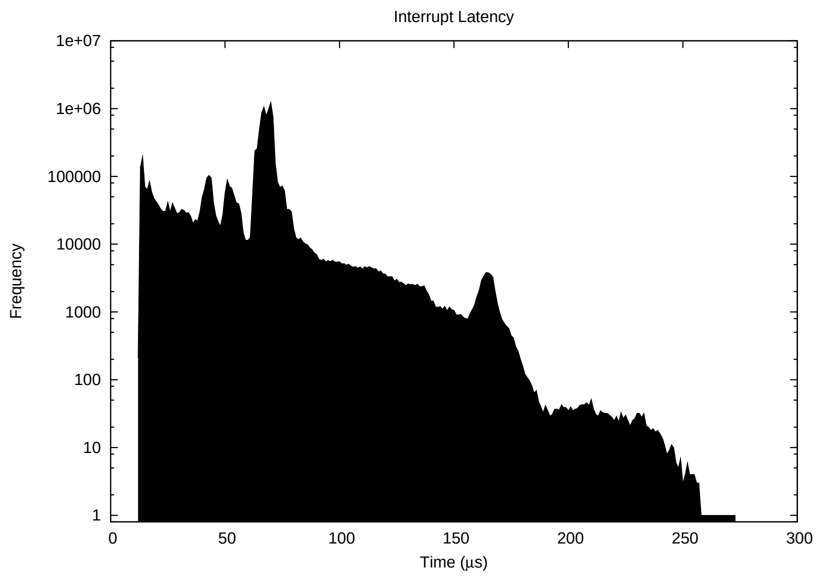


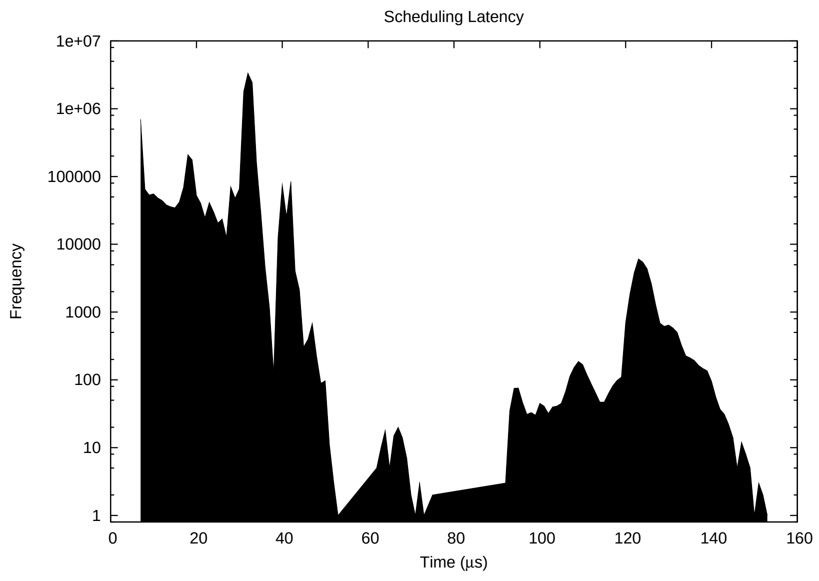
Figur 7.5: *PREEMPT_RT kernel results of the idle test.*



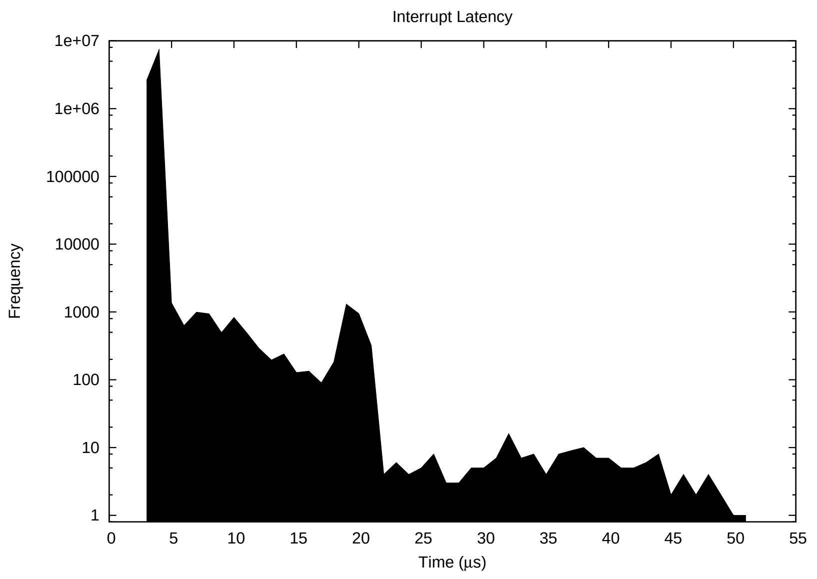


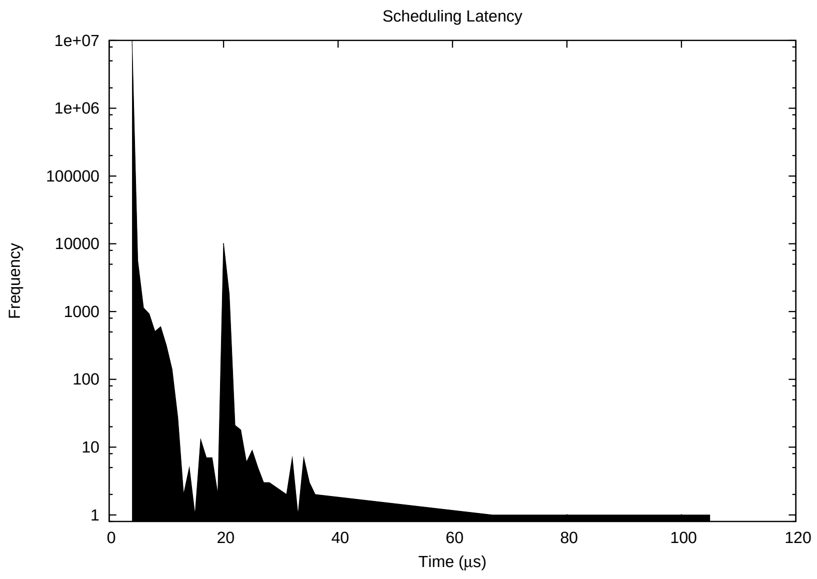
Figur 7.6: *PREEMPT_RT kernel results of the stress test.*



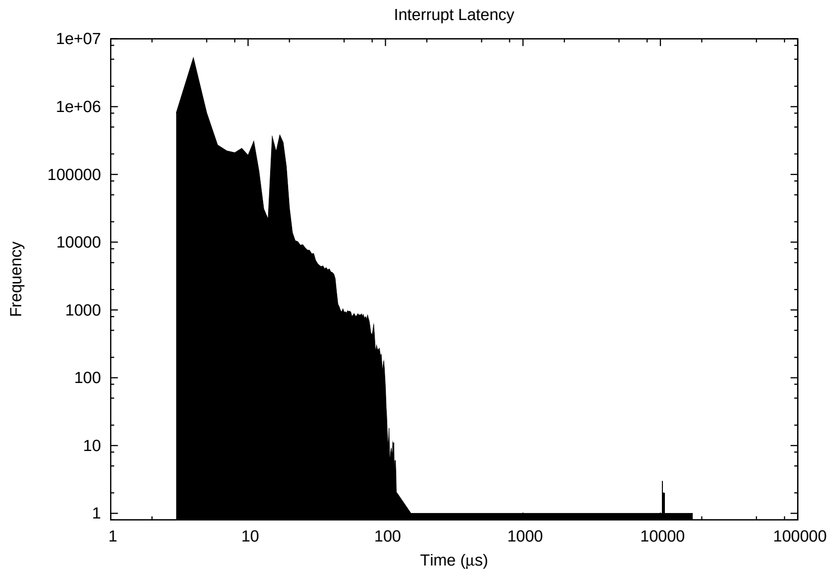


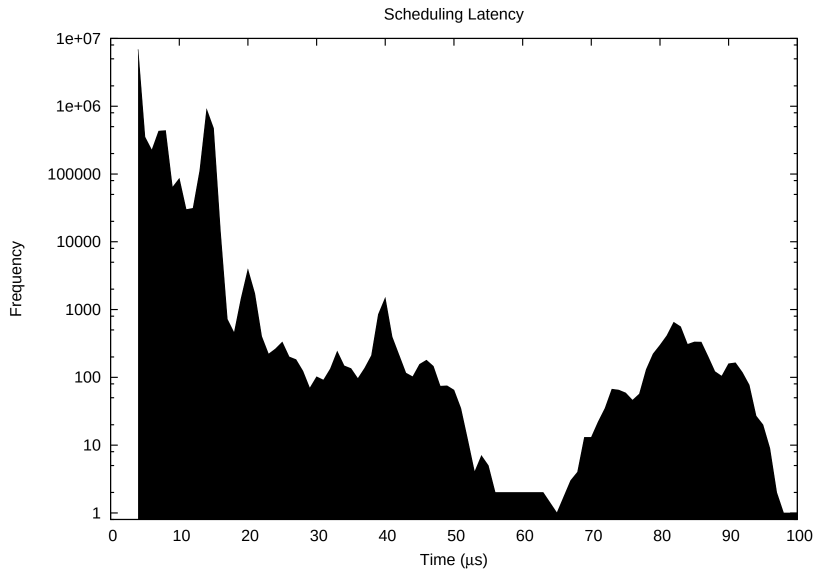
Figur 7.7: *PREEMPT_RT kernel results of the stress and network test.*



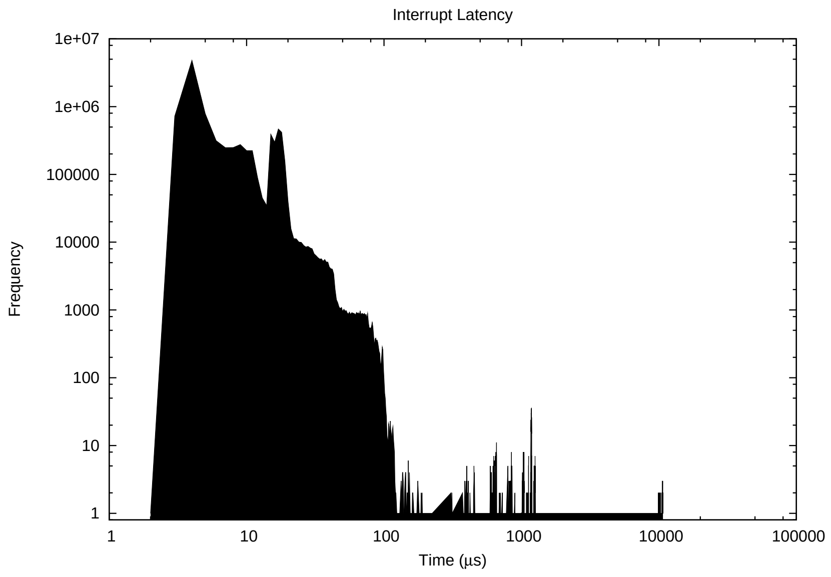


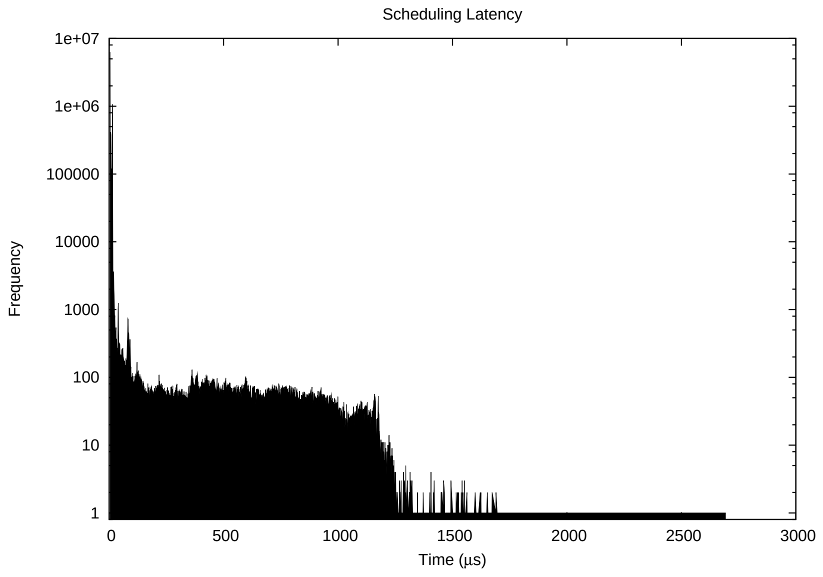
Figur 7.8: *Mainline kernel results of the idle test.*





Figur 7.9: *Mainline kernel results of the stress test.*





Figur 7.10: *Mainline kernel results of the stress and network test.*

Kapitel 8

Conclusions

The Goal of this report was to find out how different alternatives of achieving real-time capabilities using a GNU/Linux system measured up against each other. Especially, it was questionable what could be expected from an approach which did not make use of a micro-kernel. The results presented in Chapter 7 varied between the expected and the unexpected. Every test candidate brought some element of surprise to the table.

8.1 PREEMPT_RT

Expectations on the rt-patched kernel were high. It was expected that it would beat the mainline kernel in every test and that it, in turn, would be beaten in every test by Xenomai. The results show that even though the mainline kernel produces better results in the Idle Test, the mainline kernel's worst-case response times increases in many orders of magnitude when subjected to workload. Applying the same load to the rt-patched kernel only marginally increases the interrupt and scheduling latency.

More surprising was the difference between the rt-patched kernel and Xenomai, or rather the lack thereof. It seems most fair to compare the Stress and Network Test of the PREEMPT_RT kernel to the Xload Test of Xenomai. After all, these are the tests that produces load throughout the entire systems. Comparing these tests shows that Xenomai maintains a slightly better interrupt latency, with a maximum of 247μs compared to 273μs for the rt-patched kernel. Looking at the scheduling latency however, PREEMPT_RT comes out as the winner, with a maximum scheduling time of 153μs against Xenomai's 271μs. Thus, while Xenomai has a faster interrupt response, in a system where the scheduling of processes matter, the rt-patched kernel is actually more responsive. This could be an effect of the different implementations used in the different versions of Racquet. However, since the interrupt latency also increases substantially with the Xload Test, it seems more likely that the Linux scheduler is simply more rigid.

The conclusion is that the high expectations on the PREEMPT_RT patched kernel were met, and in some ways even exceeded. Though ten million samples does not prove that these numbers represents the actual worst-case response times for the respective systems, they provide a solid approximation. As such this report has proven that the Linux kernel, with some enhancements, is capable of serving as the operating system in quite demanding real-time solutions. If interrupt latency is the most critical factor, then there are faster alternatives. However, one must weigh the advantage of that against fast scheduling. Perhaps even more important is the fact that using this approach does not require learning a new API. From a developers point of view, it is still Linux.

It is important to remember that applying the PREEMPT_RT patch to a Linux kernel does not in itself produce a real-time system. The patch provides the capability schedule most of the kernel jobs. One must take advantage of this in order for the patch to have any effect. Comparing Table 7.2 with Table 7.3 clearly shows that without proper scheduling directives, there is little the patched kernel can do to the real-time performance. In this case, worst-case response times looks more like the results of the mainline kernel as seen in Table 7.4.

8.2 Xenomai

One major advantage in favor of Xenomai against the other two competitors is of course the fact that Xenomai uses a micro-kernel. The results of this approach shines through in the first three tests that Xenomai was put through. Both in the Idle, Stress, and Stress and Network Test, Xenomai displays extremely low latencies, both in response to interrupts and in scheduling time. When Xload is used to introduce extra load in the Xenomai domain however, results reveal that both the interrupt and scheduling latency increases substantially.

One might argue that forking 64 real-time processes is unrealistic. That might be, but in an effort to expose the worst-case response time, it felt necessary to impose as large amounts of load that the systems could handle. Thus, on a system that incorporates some real-time sub-systems but in which most processes will be running in a non-real-time context, Xenomai does have its clear segment. Not even the response times of the Idle Test for the rt-patched kernel can compete with Xenomai, even when compared with Xenomai during the Stress and Network Test. Even during heavy load of the Linux domain, Xenomai shows a peak interrupt response time of 58μs with a maximum scheduling latency of 76μs.

Using Xenomai, or any micro-kernel approach for that matter, requires the developers to learn a new API. This could be a complication if developers are used to the normal Linux API. On the other hand, developers who have worked with another real-time operating system that Xenomai has support for via a so called skin, the alternative API might be a positive feature.

8.3 Mainline Kernel

The decision to include the mainline kernel in the test was made in order to evaluate how far the Linux kernel could be pushed towards the real-time world. Also, by using the same kernel as the basis for the rt-patched kernel, the changes made by that patch would be easy to view.

At first, it was quite surprising to see that the mainline kernel actually produced better real-time measures than the rt-patched kernel in the Idle Test. After some thought however, that was to be expected. In order to maintain responsiveness no matter what the system load might be, the rt-patch introduced locking mechanisms that has more handling overhead than the ones they replace. Thus, when the system load is nonexistent, the extra overhead of the new locking mechanisms will be of little use. When the workload increased however, the rt-patch's rugged locks come into their right environment, while the standard spinning locks crumble under the pressure.

So is it possible to use the mainline kernel for real-time applications? The short answer would be; no. However, in a system that will normally not experience high workloads, it is certainly possible to run soft real-time systems, and systems with quite high demands on latencies at that. One must also remember that a chain is never stronger than its weakest link. That is to say, in order for a real-time system based on GNU/Linux to work as expected, all the modules loaded into the kernel must be tested for indeterministic behavior. Even if the kernel itself works well, it cannot be held responsible for the modules it is told to use.

Bilaga A

Brickwall Source Code

```
#include <string.h>

#include "LPC214x.h"
#include "type.h"
#include "irq.h"

#include "brickwall.h"

/* itoa:  convert n to characters in s */
void itoa(unsigned long n, char s[])
{
    unsigned long i, sign;

    if ((sign = n) < 0)  /* record sign */
        n = -n;          /* make n positive */
    i = 0;
    do {       /* generate digits in reverse order */
        s[i++] = n % 10 + '0';   /* get next digit */
    } while ((n /= 10) > 0);     /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}

/* reverse:  reverse string s in place */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

void uart_init(void)
{
    /* Initialize Pin Select Block for Tx and Rx */
    PINSEL0 |=  ((1<<2) | (1<<0));

    /* Enable FIFO's and reset them */
    U0FCR=0x7;

    /* Set DLAB and word length set to 8bits */
    U0LCR=0x83;

    /* Baud rate set to 115200 */
    U0DLL=0x0A;
    U0DLM=0x00;

    /* Clear DLAB */
    U0LCR=0x3;
}

void uprintresult (void)
{
  char intstr_irq[32], intstr_task[32];

  itoa(capture_irq, intstr_irq);
  itoa(capture_task, intstr_task);
  int i=0;

  while(intstr_irq[i])
  {
    U0THR=intstr_irq[i];
    i++;
  }

  i=0;
  U0THR='\t'; //HT
  while(!(U0LSR & (1<<6)));

  while(intstr_task[i])
  {
    U0THR=intstr_task[i];
    i++;
  }

  U0THR=10; //LF;
  U0THR=13; //CR;

  while(!(U0LSR & (1<<6)));
}

void timer_init(void)
{
  T0TCR = 0x02; //disable and reset

  PINSEL1 |=  (3<<28) | (2<<24);

  T0PR  = 0;      //no prescaler
  T0CTCR  &=  ~0x03;  //timer mode
  T0CCR |=  0x181;  //trigger on rising flank, irq on

  // install irq handler (light red led on error)
  if(!install_irq(TIMER0_INT, (void *)timer_handler))
  {
    IOSET0 = LED_RED;
  }
  T0TCR = 0x01; //enable timer
}

void timer_handler (void) __irq
{
  T0IR  |=  (1<<4); //clear interrupt

  IENABLE;

  if(new_capture) //we will overrun the previous capture
  {
    IOSET0 = LED_RED;
  }

  capture_irq  = T0CR0 - capture_start;
  capture_task = T0CR2 - capture_start;

  new_capture = 1;

  IDISABLE;

  VICVectAddr = 0;   //ack irq
}

void delay(void)
{
  int j, k;

  for(k=0; k < 100000; k++)
    for(j=0; j < 5000; j++);
}

int main (void)
{
  int i, capture;

  init_VIC();
  uart_init();


  // set LEDs to output and low
  IODIR0 |= LED_RED | LED_GREEN | LED_YELLOW | BUZZER;
  IOCLR0 |= LED_RED | LED_GREEN | LED_YELLOW | BUZZER;

  //enable ping pin
  PINSEL1 &= ~(0x03<<26);
  IODIR0  |= PING_PIN;

  timer_init();

  for(capture=0; capture < NO_CAPTURES; capture++)
  {
    capture_start = T0TC;

    IOSET0 = PING_PIN;
    for(i=0; i < 1; i++);
    IOCLR0 = PING_PIN;

    while(!new_capture);

    if(IOSET0 & LED_YELLOW)
      IOCLR0 = LED_YELLOW;
    else
      IOSET0 = LED_YELLOW;

      uprintresult();
      new_capture = 0;
   }

   IOSET0 = BUZZER;
   delay();
   IOCLR0 = BUZZER;

   while(1)
   {
      if(IOSET0 & LED_GREEN)
         IOCLR0 = LED_GREEN;
      else
         IOSET0 = LED_GREEN;

      delay();
   }

   return 0;
}
```

Bilaga B

Racquet Source Code

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/irq.h>
#include <linux/interrupt.h>

#include <asm/arch/pxa-regs.h>
#include <asm/arch/pxa2xx-regs.h>
#include <asm/arch/hardware.h>

#include "racquet.h"

static void racquet_irq_tl(unsigned long dummy)
{
  GPCR(RACQUET_REPLY) = GPIO_bit(RACQUET_REPLY);
}

static irqreturn_t racquet_irq(int i, void *d, struct pt_regs *r)
{
  GPSR(RACQUET_REPLY) = GPIO_bit(RACQUET_REPLY);

  tasklet_hi_schedule(&racquet_tl);

  return IRQ_HANDLED;
}

static int racquet_init(void)
{
  printk(KERN_ALERT "racquet:␣init()\n");

  pxa_gpio_mode(RACQUET_REPLY | GPIO_OUT);
  pxa_gpio_mode(RACQUET_IRQ | GPIO_IN);
  set_irq_type(IRQ_GPIO(RACQUET_IRQ), IRQT_RISING);

  if(request_irq(IRQ_GPIO(RACQUET_IRQ),
      (void *)&racquet_irq,
      0, "racquet irq", NULL))
    return -1;

  return 0;
}
module_init(racquet_init);

static void racquet_exit(void)
{
  free_irq(IRQ_GPIO(RACQUET_IRQ), NULL);

  printk(KERN_ALERT "racquet: exit()\n");
}
module_exit(racquet_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tobias Knutsson");
```

Bilaga C

Xenomai Racquet Source Code

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/irq.h>
#include <linux/interrupt.h>
#include <asm/arch/pxa-regs.h>
#include <asm/arch/pxa2xx-regs.h>
#include <asm/arch/hardware.h>
#include <native/task.h>
#include <rtdm/rtdm_driver.h>
#include "racquetx.h"

static int racquet_irq(rtdm_irq_t *irq_context)
{
  GPSR(RACQUET_REPLY) = GPIO_bit(RACQUET_REPLY);

  rtdm_event_signal(&racquet_irq_event);

  return RTDM_IRQ_HANDLED;
}

static void racquet_tl(void* cookie)
{
  while(!rtdm_event_wait(&racquet_irq_event))
  {
    GPCR(RACQUET_REPLY) = GPIO_bit(RACQUET_REPLY);
  }
}

static int racquet_init(void)
```

```
{
  printk(KERN_ALERT "racquet: init()\n");

  rtdm_event_init(&racquet_irq_event, 0);

  if(rt_task_create(&racquet_tasklet,
          "Racquet IRQ Tasklet", 0, 99, 0))
    goto fail_task;

  rt_task_start(&racquet_tasklet, racquet_tl, 0);

  pxa_gpio_mode(RACQUET_REPLY | GPIO_OUT);
  pxa_gpio_mode(RACQUET_IRQ | GPIO_IN);
  set_irq_type(IRQ_GPIO(RACQUET_IRQ), IRQT_RISING);

  if(rtdm_irq_request(&racquet_irq_handle, IRQ_GPIO(RACQUET_IRQ),
          racquet_irq, 0, "racquet", 0))
    goto fail_irq;

  return 0;

fail_irq:
  rt_task_delete(&racquet_tasklet);

fail_task:
  rtdm_event_destroy(&racquet_irq_event);

  return -1;
}
module_init(racquet_init);

static void racquet_exit(void)
{
  rtdm_irq_free(&racquet_irq_handle);

  rt_task_delete(&racquet_tasklet);

  rtdm_event_destroy(&racquet_irq_event);

  printk(KERN_ALERT "racquet: exit()\n");
}
module_exit(racquet_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tobias Knutsson");
```

Bilaga D

Xload Source Code

```
#include <sys/mman.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include <native/task.h>
#include <rtdm/rtdm.h>

#define NO_TASKS  64
#define TASK_PRIO 98
#define TASK_PERIOD 5000000 //ns

RT_TASK cpu_task[NO_TASKS];

void processing_task(void* cookie)
{
  rt_task_set_periodic(NULL, TM_NOW, TASK_PERIOD);

  while(1) {
    rt_task_wait_period(NULL);
    sqrt(rand());
  }
}

void quit(int sig)
{
  unsigned int i;

  for(i=0; i < NO_TASKS; i++)
```

```
        rt_task_delete(&cpu_task[i]);

    printf("Deleted rt-tasks.\n");

    exit(0);
}

int main(void)
{
    unsigned int i, j;

    mlockall(MCL_CURRENT|MCL_FUTURE);

    signal(SIGINT, quit);
    signal(SIGKILL, quit);

    for(i=0; i < NO_TASKS; i++) {
        if(!rt_task_create(&cpu_task[i], NULL, 0, TASK_PRIO, 0))
            rt_task_start(&cpu_task[i], &processing_task, 0);
        else {
            printf("Could not create rt-task %i, killing all.\n", i);
            for(j=0; j < i; j++)
                rt_task_delete(&cpu_task[j]);
            exit(-1);
        }
    }

    while(1);

    return 0;
}
```

Litteraturförteckning

[1] Atmel Corporation. *8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash - Summary*, 2007. `http://www.atmel.com/dyn/resources/prod_documents/2545S.pdf`.

[2] A. Barbalce, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. *Performance Comparison of VxWorks, Linux, RTAI and Xenomai in a Hard Real-time Application*, 2007. `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4382787&isnumber=4382726`.

[3] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Second Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[4] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages, Second Edition.* Addison Wesley Longman Limited, 1996.

[5] Fabrizio Caccavale, Vincenzo Lippiello, Bruno Siciliano, and Luigi Villani. *RePLiCS: An Environment for Open Real-Time Control of a Dual-Arm Industrial Robotic Cell Based on RTAI-Linux*, Februa-ry 2005. `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1545520&isnumber=32977`.

[6] Jonathan Corbet. *A new approach to kernel timers*, September 2005. `http://lwn.net/Articles/152436/`.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition.* O'Reilly Media, Inc., February 2005.

[8] Sven-Thorsten Dietrich and Daniel Walker. *The Evolution of Real-Time Linux.* Monta Vista Software Inc, 2005. `http://www.fabiotec.com/file/real_time_preempt.pdf`.

[9] Lorenzo Dozio and Paolo Mantegazza. *Real Time Distributed Control Systems using RTAI*, May 2003. `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1199229`.

[10] Philippe Gerum. *Xenomai - Implementing a RTOS emulation framework on GNU/Linux*, April 2004. `http://dslab.lzu.edu.cn/docs/summer_school_2005/rtai/rtai-doc/generated/pdf/xenomai.pdf`.

[11] Christopher Hallinan. *Embedded Linux Primer: A Practical Real-World Approach.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[12] Hewlett-Packard Company. *HP 33120A Function Generator/Arbitrary Waveform Generator - User's Guide*, August 1997. `http://www.me.berkeley.edu/ME135/Lab/HP33120A_UsersGuide.pdf`.

[13] Robert Hill, Balaji Srinivasan, Shyam Pather, and Douglas Niehaus. *Temporal Resolution and Real-Time Extensions to Linux.* Department of Electrical Engineering and Computer Sciences, University of Kansas, June 1998. `http://www.ittc.ku.edu/kurt/papers/techreport.ps.gz`.

[14] M. Tim Jones. *Inside the Linux scheduler*, June 2006. `http://www.ibm.com/developerworks/linux/library/l-scheduler/`.

[15] M. Tim Jones. *Anatomy of real-time Linux architectures*, April 2008. `http://www.ibm.com/developerworks/linux/library/l-real-time-linux/`.

[16] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Inc, 1988.

[17] Robert Love. *Linux Kernel Development, Second Edition.* Novell Press, June 2005.

[18] Paolo Mantegazza. *DIAPM RTAI for Linux: WHYs, WHATs and HOWs.* Dipartimento di Ingegneria Aerospaziale Politecnico di Milano, December 1999. `https://www.rtai.org/index.php?module=documents&JAS_DocumentManager_op=downloadFile&JAS_File_id=31`.

[19] Motorola Inc. *MVME5500 Series Single-Board Computer*, February 2002. https://mcg.motorola.com/us/ds/pdf/ds0179.pdf.

[20] Egan Orion. *Linux source code passes 10 million lines*, October 2008. http://www.itnews.com.au/News/87317,linux-source-code-passes-10-million-lines.aspx.

[21] Steven Rostedt and Darren V. Hart. *Internals of the RT Patch*, June 2007. http://ols.108.redhat.com/2007/Reprints/rostedt-Reprint.pdf.

[22] Richard Stallman. *The GNU Project*, January 2008. http://www.gnu.org/gnu/thegnuproject.html.

[23] Martin Thomas. *gcc-Port of the LPC2000 Example-Code-Bundle for LPC213x/LPC214x (Philips LPC2000 ARM7TDMI-S controller series)*, January 2007. http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/lpc2k_bundle_port/index.html.

[24] University of Kansas. *KURT-Linux News*, February 2005. http://www.ittc.ku.edu/kurt/.

[25] Wind River Systems. *What Is RTLinux?*, October 2007. http://www.rtlinuxfree.com/.

[26] The Xenomai Project. *A Tour of the Native API, Revision C*, March 2006. http://www.xenomai.org/documentation/branches/v2.3.x/pdf/Native-API-Tour-rev-C.pdf.

[27] The Xenomai Project. *Xenomai API Documentation*, December 2007. http://www.xenomai.org/documentation/trunk/html/api/.

[28] The Xenomai Project. *FAQs*, August 2008. http://www.xenomai.org/index.php/FAQs.

[29] Karim Yaghmour. *Building a Real-Time Operating System on top of the Adaptive Domain Environment for Operating Systems*, XXX 0. http://www.opersys.com/ftp/pub/Adeos/adeos.pdf.

[30] Karim Yaghmour. *Building Embedded Linux Systems.* O'Reilly Media, Inc., April 2003.

[31] Victor Yodaiken and Michael Barabanov. *A Real-Time Linux*. New Mexico Institute of Technology, 1997. `http://www.idt.mdh.se/kurser/ct3660/html/PeterOliver2.pdf`.