NTNU
Det skapende universitet

Better Real-Time Capabilities For The AVR32 Linux Kernel

Morten Engen

Master i teknisk kybernetikk
Oppgaven levert: August 2007
Hovedveileder: Amund Skavhaug, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Oppgavetekst

Formålet med oppgaven er å forbedre sanntidsegenskapene til Linux-kjernen for AVR32 ved å ta i bruk teknikker fra Ingo Molnars realtime-preempt patchsett (-rt) som er tilgjengelig fra http://people.redhat.com/~mingo/realtime-preempt/

Noen mekanismer som vil være interessante å se på er:

- High-resolution timers
- Dynamic ticks. Isteden for å sette opp et timer-interrupt som trigger med en fast periode, programmeres timeren om til å kun generere interrupts for faktiske timer events.
- Fully preemptible kernel

Oppgaven består i:

- Forstudie for å bli kjent med AVR32.
- Forstudie for å bli kjent med Linux kjernen, spesielt hvordan timere og ulike preemption modeller er implementert i dag.
- Evaluering av hva som må gjøres for å kunne ta i bruk realtime-preempt patchsettet.
- Implementasjon av mekanismene nevnt ovenfor.
- Måling av interrupt- og scheduling-latency for å demonstrere grad av forbedring.

Oppgaven gitt: 19. mars 2007
Hovedveileder: Amund Skavhaug, ITK

Better Real-Time Capabilities for the AVR32 Linux Kernel

Morten Engen

Master of Science Thesis
performed at

NTNU

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrial Engineering
Department of Engineering Cybernetics

Trondheim, August 13, 2007

Preface

This thesis is presented as a partial fulfilment of the degree *Sivilingeniør i Teknisk Kybernetikk* at NTNU.

I would first and foremost like to thank Amund Skavhaug for his continuous guidance and support during the thesis. I would also thank Håvard Skinnemoen from Atmel Norway for his help and support. They have both been great inspirational sources.

Secondly, I would like to thank all the my fellow students David Karnå, Anders Fougner and Lars Vråle in the creation of an excellent working environment.

Finally, thanks to you Siv, for so much.

Morten Engen
Trondheim, August 13, 2007

Summary

Over the past few years, a small group of Linux core developes have worked on a project known as the realtime-preempt patch, that incorporates real-time capabilities into the Linux kernel. The realtime-preempt patch incorporates a fully preemptible kernel, high resolution timers and dynamic ticks. However, the patch supports a limited number of architectures, excluding the AVR32 architecture among others.

In this thesis, the real-time capabilities of the AVR32 Linux kernel have been improved. This were done by implementing techniques adopted from the realtime-preempt patch. The AVR32 time related code has been reworked to support preliminary subsystems such as generic time-of-day and clock events, for high resolution timers and dynamic ticks solution. Thus, high resolution timers and dynamic ticks have been successfully ported in its entirety to AVR32 Linux 2.6.21.

The preemptibility of the AVR32 Linux has been improved. Many of the architecture-dependent changes required by a fully preemptible kernel are implemented. However, in the process of transforming AVR32 Linux to a fully preemptible kernel, the investigation identified the need of a more advanced interrupt controller for the AVR32 architecture. With the current controller, hard interrupts cannot be converted into kernel threads.

Benchmark results are provided for the verification of performance for the AVR32 supported high resolution timers and dynamic ticks. The results shows a significant improvement in timer latencies over the standard AVR32 Linux kernel.

With high-resolution timers, applications can utilize time driven and event driven events with microsecond accuracy, allowing more efficient use of CPU resources and finer grained control of the system.

Contents

Chapter 1

Introduction

Linux is a portable general purpose operating system that supports a wide range of computer architectures, including the AVR32 architecture provided by Atmel. Over the past few years, a small group of Linux core developers, led by Ingo Molnar, have worked on a project that incorporates real-time capabilities into the kernel. The work is distributed in a patch, referred to as the RT-preempt patch in this document. As the RT-preempt patch develops, several of its methods are incorporated into the upstream kernel. However, the patch requires changes in the architecture-dependent source code to fully utilize its functionality. The AVR32 architecture lacks such support[1]. However, as being a embedded architecture, better real-time capabilities desirable. Hence, the topic of this Master of Science thesis is *improving the real-time capabilities for the AVR32 Linux* by incorporate mechanisms used in the RT-preempt patch to this architecture.

1.1 Motivation

Many embedded applications need predictable and deterministic behavior in order to meet real-time requirements, while others have such requirements only for a small part of the overall system. Therefore, to meet the demands from a wide range of embedded applications, the operating system must create such a behavior. The primary purpose of the operating system is not to increase the speed of the application, or lower the latency between an action and response unconditionally, but provide a system which can operate under any given load with deterministic results. However, increasing performance and minimizing latency improves the quality of the system, as long as deterministic behaviour is retained.

The official Linux kernel available at `http://www.kernel.org` is a general purpose operating system. One of the goal of such operating systems is to maximize throughput and minimize response time. These objectives are often conflicting because minimizing latency requires more context switching. This means more

[1] At the time of this writing, the 2.6.21 RT-preempt patch set supports; ARM/StrongARM, Intel x86, IA-64, MIPS, PowerPC, Hitachi SH, and x86-64.

overhead, which can decrease throughput. In addition, the scheduler attempts to make scheduling decisions to meet the total set of requirements. In Linux, the scheduler makes these decisions dynamically to ensure efficient scheduling.

Soft real-time systems may have their requirements satisfied by general purpose operating systems. Such operating systems strive to have a good average performance and high throughput. However, hard real-time systems require guarantees in correct timing beyond that general purpose operating systems usually provide. Such facilities are provided by real-time operating systems. An real-time operating system objective is to minimize interrupt latency and predictable scheduling. An RTOS does not necessary have high throughput compared with general purpose operating systems.

In spite of the aforementioned, Linux is attractive for embedded and real-time applications because of its low cost, adaptability, configurability, and portability. Linux is used in embedded products such as PDAs and handhelds, cell phones and audio/video devices (Mosnier, 2005). Further, Linux based operating systems are used in robots and in the vehicle industry. Nevertheless, the real-time capabilities of Linux is a reason for not to use Linux in embedded applications.

The 2.6 Linux kernel is *preemptive*. That is, in brief, when a process becomes runnable, the kernel checks whether its priority is higher than the priority of the currently running process in kernel mode. If it is, the scheduler is invoked to preempt the currently executing process and replace it with the newly runnable process. Kernel preemption has improved reaction time and lowered latencies. However, the current implementation contains code where kernel preemption is disabled. A low priority process is not preempted by a high priority process while it is in a critical section. Moreover, preemption are disabled when the kernel is executing an interrupt service routine or the deferred portion of the interrupt processing. For embedded application with real-time requirements, this is unacceptable. The goal of the RT-preempt patch is to minimize the amount of kernel code that is non-preemptible. With the patch included, nearly all kernel code is preemptive, with the exception of a few very small regions of code.

Standard Linux can provide a maximum resolution of 1 millisecond. In a wide range of embedded applications, it is desirable to achieve a higher resolution, i.e. high resolution timers.

In the beginning, the RT-preempt patch was not accepted into mainline Linux by the kernel maintainers. The problem was that large intrusive changes were made throughout the kernel without correlation to the mainline development. However, Ingo Molnar, as a kernel maintainer, noticed the beneficial part of the RT-preempt patch for Linux. Therefore, he began to incorporate methods from the RT-preempt patch into the kernel in small steps. Since then, several major features of the RT-preempt patch have already been incorporated into the mainline kernel.

The AVR32 processor by Atmel targets embedded multimedia systems developed in cell phones, digital cameras, PDAs, automotive infotainment, as well

as network switches and routers. As already mentioned, such systems may have real-time requirements which the mainline kernel fail to comply. In sum, two broad arguments justifies porting the RT-preempt patch to the AVR32 architecture:

- First, the architecture may be capable to utilize the features already incorporated into the mainline kernel by the patch.
- Second, the architecture may be capable to meet stricter real-time demands, and, hence, be more attractive for the embedded market.

1.2 The approaches to real-time Linux

The RT-preempt patch incorporates three different categories:

1. *Fully preemptible kernel* – By transforming spin locks into mutexes that support priority inheritance, and converting interrupt service routines into threads, nearly all of the kernel is preemptible, with the exception of a very small regions of code.
2. *High resolution timers* – By separating timers from timeouts and introducing a new clock event manager, the clock event resolution is no longer bounded to jiffies, but to the underlining hardware itself.
3. *Dynamic ticks* – By reprogramming the clock event manager to trigger on the next expiring timer or on the completion of an I/O operation, the clock event frequency is no longer bounded to be periodic.

Although the components are implemented with minimum of architecture dependence, they require changes in architecture-dependent source code.

The fully preemptible kernel feature differs notably from the others. Indeed, the high resolution timers and dynamic ticks patch, referred to as the HRT patch in this thesis, is independently maintained under the leadership of Thomas Gleixner. However, the RT-preempt patch pulls the HRT patch regularly.

The steps towards a fully preemptible kernel, and previous approaches to implement high-resolution timers and dynamic ticks for Linux will be discussed in the rest of this section. Subsequently, a section giving a brief overview of commercial Linux distributions and vendors is included. The Chapter finishes off by giving an overview of the contribution of this thesis, and an overview of material covered in this thesis.

1.2.1 Fully preemptive kernel

Linux was originally designed as a non-preemptible kernel, and hence not very suited for real-time applications. One of the earliest attempts of improving kernel preemption was introduced as a patch by MonteVista (Anzinger and Gamble, 2000). This approach allows the scheduler to reschedule a task while

it is in the kernel, except while the kernel handles interrupts, executes deferred interrupt handling, holds a spin lock, or executes the scheduler itself. In the development of the 2.5 kernel, the patch was adopted by the Linux kernel project as a mainline feature (Gamble, 2001). However, long latencies are produced when spin locks were held for a long time. Some of the longest critical sections have been cut into smaller non-preemptible sections by using the approach used by Low-Latency Linux (Morton, 2001). In Linux 2.6 the preemptible kernel has been integrated as a kernel configuration option.

In the work of Heursch et al. (2004), two concepts to increase the preemptibility of the Linux kernel are developed. First, spin locks are converted into mutexes. Since mutexes might sleep, they can cause priority inversion. To avoid priority inversion situations, a priority inheritance protocol is implemented under the GPL license. Second, all interrupt service routines besides the timer interrupt is handled as kernel threads. Similar concepts to increase the preemptiability of the kernel is also reported in (Yang et al., 2005). However, the problem with the implemetaions were that large intrusive changes were made throughout the kernel without correlation to the mainline development. However, Ingo Molnar, as a kernel maintainer, noticed the beneficial part of the RT-preempt patch for Linux. Therefore, he began to incorporate methods from the RT-preempt patch into the kernel in small steps. His work is documented in (McKenney, 2005a) and (Rostedt and Hart, 2007).

1.2.2 High-resolution timers

The first approach to implementing high-resolution timers in Linux is the UTIME component of the Kansas University Real-Time project (KURT)[2]. These extensions provide on-demand, microsecond resolution and real-time scheduling capabilities to standard Linux, version 2.0 to 2.4 (Niehaus et al., 1998, 2005). The approach is based on the observation that even though real-time tasks are scheduled with microsecond level deadlines, events are rarely scheduled to occur *every* microsecond. Thus, UTIME implements a mechanism by which allows the timer interrupts to occur at *any* microsecond, not necessary *every* microsecond.

In order to achieve microsecond accuracy, a *fractional expiration* field is added in the timer data structure. This structure already holds the timeout value of the timer specified by the *expiration* field. Now, the fractional expiration value specifies the microsecond within the expiration value at which the timer will expire. Hence, this field allows the user to specify how many microseconds after the expiration value the timer should expire. This mechanism for having microseconds accuracy is designed and implemented into the existing timer manager while maintaining compatibility (Niehaus et al., 1998, chap. 3)[3]. Results. The UTIME implementation is restricted and nanosleep and itimers only.

[2]Home page `http://www.ittc.ku.edu/kurt/`

[3]Doubly linked list sorted in ascending order of time. In newer versions of the kernel, the timer is now a heap based data structure. The change introduces no logical changes of the time-handling functions in UTIME

The approach of supporting POSIX high-resolution timers is introduced by the High-Resolution Timers project [4], hereafter referred to as HRT. The design of HRT was extended from the design used by UTIME, thus, the high-resolution part of the timer has its own field in the timer structure. Moreover, the high-resolution timers are kept in the timer manager until they reach the value of their expiration field (Dietrich and Walker, 2005). Then, however, on expiry they are moved to a separate list for high-resolution timers. A separate timer interrupt is used to trigger the timers in the high-resolution timer list. In other words, the HRT introduces a separate interrupt to maintain microsecond resolution.

The implementation of high-resolution timers in UTIME and HRT are tightly bounded to the timer management of the longer-term generic timers. As pointed out in (Gleixner and Niehaus, 2006), the conclusions drawn from both projects demonstrated that the timer management trying to integrate high-resolution timers into the existing framework introduced significant overhead, and variable latencies. Moreover, the necessary changed in the affected code were scattered and unreliable. Hence, the authors conclude that a separate support for high-resolution timers and longer-term generic timers is necessary. In (Gleixner and Molnar, 2005), such an approach for implementing high-resolution timers is described. The approach, known as *Ktimers*, introduces a new subsystem for the high-resolution timers that does not interact with the existing timer management for reasons already mentioned. Ktimers are entierly based on nanoseconds timeouts specifications, since the primary users of high-resolution timers are user-space applications which request for timeouts in some form of human time unit. Like HRT, Ktimers uses it's own interrupt to triggers times. However, since the subsystem uses the existing timer interrupt and not utilizes an independent high-resolution clock source, Ktimers does *not* provide better resolution than previous timers.

Clock source management, clock synchronization, time-of-day representation (Stultz et al., 2005). Ktimers does not rely on work provided by Stultz et al. (2005), but the usage is simpler with the work in place.

Later, in (Gleixner and Niehaus, 2006), the work of Ktimers, now called *hrtimers*, are extended. A new component called *clock events* is described. Together with the new time-of-day approach proposed by Stultz et al. (2005), this component provides a base for high-resolution timer support for hrtimers. In order to achieve better resolution than existing timers, the interrupt that triggers the high-resolutions timers is separated from the existing tick bound timer interrupt. Hence, if the system has necessary hardware support, high-resolution timers is achieved.

1.2.3 Dynamic ticks

A work which can be characterized as dynamic ticks is announced in Anzinger (2003). This patch called Variable Scheduling Timeouts (VST), provides the suppression of timer ticks during idle periods. When the system goes into idle

[4] Home page `http://sourceforge.net/projects/high-res-timers`

state, the strategy is to find the next expiring timer, and if it is reasonable far into the future, turn off the periodic timer interrupt. Then, the timer interrupt is reprogrammed to trigger at this timer's expiration time. On the next interrupt which could be either the reprogrammed timer interrupt or some other interrupt, the periodic timer interrupt is restarted and the elapsed time is properly accounted for. The patch is closely related to and depending on HRT, and supports Intel x86 platforms.

The Dynamic Ticks patch (Kalivas, 2005) is an implementation similar to the VST patch, but does not depend on other patches. Since the implementation contains some Intel x86 architecture specific bits, it is tightly bound to this platform.

1.3 Commercial real-time Linux distributions

Commercial real-time Linux distributions adopt one of the following two approaches (Marchesotti et al., 2006):

1. Modifying and patch mainline Linux to provide the behavior close of a full-fledged real-time kernel.

2. Modifying and patch mainline Linux to cooperate with a hard real-time sub-kernel. The sub-kernel controls the system and schedules all the hard real-time tasks, while Linux runs as one low-priority process of the real-time kernel, and takes care of non real-time tasks.

MontaVista Software's real-time Linux solution is an example of the first approach. As seen, their solutions are jointly developed with the Open Source community. In 1999, the MontaVista Preemptible Linux Kernel patch was adopted as a mainstream Linux feature. In addition, in 2002, MontaVista introduced High Resolution Timers for better timing resolution than the standard Linux, that time, 10 milliseconds timebase.

The second approach is used by solutions such as RTAI[5], BlueCat Linux by Lynxworks[6], and Xenomai[7].

1.4 Contribution of this thesis

The main contribution achieved and reported in this thesis is the incorporation of high resolution timers and dynamic ticks for AVR32 Linux. Secondly, the thesis starts the work of converting the AVR32 Linux kernel into a fully preemptible kernel.

[5] `http://www.rtai.org/`
[6] `http://www.lynuxworks.com/`
[7] `http://www.xenomai.org/`

1.5 An overview of the content

Chapters 2–5 provide necessary background to understand the RT-preempt patch and a thorough treatment of the implementation of the RT-preempt patch on the AVR32 architecture. Chapter 2 investigates the current implementation of the preemptive kernel as regard process preemption, interrupt service routine, and deferred interrupt handling. Then, Chapter 3 provides implementation details concerning improved AVR32 Linux kernel preemptibility.

The Linux timer management and kernel timers are treated in Chapter 4, before the components of hrtimers are investigated in details in chapter 5. Since the implementation is architecture dependent, this chapter describes the required changes in source architecture-dependent code to implement high-resolution timers on and dynamic ticks for AVR32 Linux.

Chapter 6 provides experimental results to demonstrate the level of improvement. The discussion is found in Chapter 7. Conclusion drawn from the project is present in Chapter 8, and potential future work is described in Chapter 9.

Chapter 2

Preemptive Kernel

This chapter introduces all the essential concepts about the preemptive Linux kernel. In these pages, scheduling latency caused by non-preemptable sections in the kernel or in the drivers is described. This latency includes critical sections, interrupts service handlers and other kernel constructs such as bottom halves and tasklets. Understanding the concept of preemptivity in the mainline kernel is an essential foundation to understand the extension done by the RT-preempt patch.

2.1 Process scheduling

In Linux, a process is one of the fundamental abstraction besides files. A process is a program in execution, i.e. running program code. However, a process also includes a set of resources, internal kernel data, processor state, an address space, one or more threads of execution, and a data section containing global variables (Love, 2005, chap. 3).

Linux provides a *priority-based* scheduling. Hence, processes with a higher priority runs before those with lower priority. Moreover, the scheduling scheme is *dynamic*. Processes begin with an initial base priority and then the scheduler is enable to increase or decrease the priority dynamically to fulfill scheduling objectives, i.e. high throughput and low latency.

The kernel schedules individual threads, not processes. This is done by invoking the `schedule()` function defined in `kernel/sched.c`. The scheduler algorithm and supporting code were rewritten during the 2.5 kernel development series. The new scheduler implements fully `O(1)` scheduling, that is, the scheduler is constant in execution time with respect to number of processes. This is essential for real-time applications, since the `O(1)` algorithm provides deterministic performance.

Linux processes are *preemptable*. When a process enters the `TASK_RUNNING` state[1], the kernel checks whether its priority is higher than the priority of the currently running process. If it is, the execution of the current process is

[1]Each process on the system is in exactly one of the following five states: `TASK_RUNNING`,

interrupted and the scheduler is invoked to select another process to run. This is usually the newly runnable process.

The kernel provides a `TIF_NEED_RESCHED` flag to notify the kernel that a reschedule should be performed. The flag is stored in the `flag` field of the `thread_info` structure. The flag signify the kernel that the scheduler must be invoked as soon as possible because another process deserves to run. The scheduler is invoked when the kernel knows it is in a safe quiescent state.

The `TIF_NEED_RESCHED` flag is checked before returning to user-space either from a system call or from an interrupt handler. If it is set, the scheduler is invoked to perform a reschedule, and an user preemption has occurred. Both return paths for returning from kernel space and interrupt are architecture dependent, and implemented in assembly in `arch/avr32/kernel/entry-avr32b.S` for the AVR32 architecture.

Prior to version 2.5, the kernel was non-preemptible. Kernel code ran until it was finished or explicitly blocked. During the 2.5 kernel development series, the kernel became preemptible. In the mainline 2.6 kernel, the kernel is preemptive when the kernel is executing in process context and does not hold a lock. The acquired locks are accounted for in a preemption counter field, `preempt_count`, in each process's `thread_info` structure. The counter is initialized at zero and is incremented once for each lock that is acquired and is decremented once for each lock that is released. When the counter is zero, the kernel is preemptible. Upon returning to kernel-space from an interrupt handler, the kernel checks the value of `TIF_NEED_RESCHED` flag and `preempt_count`. If the flag is set and `preempt_count` is zero, the scheduler is invoked to perform a reschedule. If the `preempt_count` is nonzero, the current process holds a lock and it is unsafe to reschedule. When the current process releases its last lock and its `preempt_count` returns to zero, the unlock code checks whether the `TIF_NEED_RESCHED` flag is set. If it is, the scheduler is invoked.

In sum, kernel preemption can occur

- When returning from an interrupt handler to kernel-space.
- When `preempt_count` is zero.
- If a process in the kernel explicitly calls `schedule()`.
- If a process in the kernel blocks which results in explicit call to `schedule`.

2.2 Scheduling latency

Consider a task τ enters its `TASK_RUNNING` state at time t. If preemption is disabled for some reason, τ will not be scheduled until preemption is re-enabled at time t'. Hence, τ experiences a scheduling latency equal $t' - t$, caused by non-preemptable sections in the kernel or in the drivers. Such sources of non-preemptable include;

`TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_ZIMBIE`, or `TASK_STOPPED`.

- interrupt handlers,
- bottom halves and deferring work,
- critical sections, and
- code sequences where interrupts are disabled.

In the following, each of the sources listed above are examined in order to identify their impact on scheduling latency and kernel preemption.

2.3 Interrupts and interrupt handlers

An *interrupt* is simply a signal that the hardware can generate when it wants the processor's attention. The function the kernel runs in response to a specific interrupt is called an *interrupt handler* or *interrupt service routine* (ISR). Because an interrupt can occur at any time, interrupt handlers by their nature, run concurrently with other code. To resume execution of the interrupted code as soon as possible, it is desirable that the handler runs quickly. However, interrupt handlers have often a large amount of work to perform. The goals of executing quickly and perform a large amount of work are in conflict. Linux resolves this problem by dividing the processing of interrupts into two halves; *top half* and *bottom half*. The top half performs the time-critical, hardware-specific work immediately upon receipt of an interrupt. The bottom half performs work that may be delayed for a time without affecting the kernel operations. Non-critical deferrable work are discussed in the later Section 2.4 on page 13.

2.3.1 Interrupt context

When executing an interrupt handler, the kernel is in *interrupt context*. This context is not associated with a process, and therefore, suffers some restrictions on what it can do. Most importantly, handlers cannot call any function that might sleep, because the handlers should be quick and simple. Secondly, handlers cannot block or otherwise invoke the scheduler.

2.3.2 Interrupt handling in the AVR32 architecture

The implementation of the interrupt handling system in Linux is architecture dependent. This section examines the internals of hardware and software AVR32 32 interrupt handling.

Interrupt controller

A peripheral issues an interrupt by sending an electrical signal over its interrupt request line to the interrupt controller. The interrupt controller collects the requests, prioritizes them, and delivers an interrupt request to the processor. The AVR32 architecture supports four priority levels for regular, maskable interrupts, and a Non-Maskable Interrupt (NMI).

Figure 2.1 gives an overview of the interrupt controller in the AVR32 architecture (Atmel, 2006, chap. 13). The interrupt controller supports up to 64 groups of interrupts. Each group has an Interrupt Request Register (IRR) and an Interrupt Priority Register (IPR). The IRR register has 32 bits, that is 32 interrupt request lines, used to identify active interrupt requests within each group. Consequently, the AVR32 supports a total of 2048 possible interrupt lines. The IPR are used to assign a *priority level* and an *autovector* to each group. The priority level is from INT0 to INT3, and the autovector specifies the address offset of the interrupt handler.

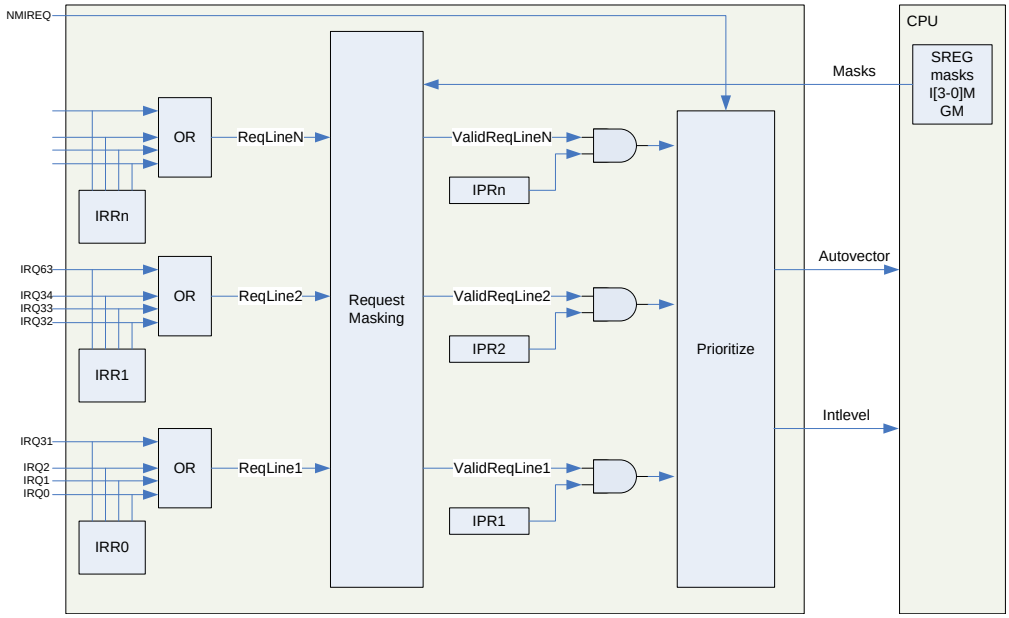

Figure 2.1: Interrupt controller (Atmel, 2006).

All of the incoming interrupt requests are sampled into the corresponding IRR. In each group, the IRR bits are logically-ORed together to form a group request line. This line indicates if there is a pending interrupt in the corresponding group.

The Request Masking hardware maps each group request line to a priority level from INT0 to INT3 specified by the corresponding IPR register. Then, the request line is masked by the interrupt level mask bit associated with its priority level, and the global interrupt mask (GM) bit. When a request line is masked, all interrupt delivery on this line is disabled for the entire processor. These bits are part of the processor's status register. Since AVR32 supports four different priority levels, there are four interrupt level masks: I0M to I3M. An interrupt request is masked if either the GM or the corresponding I[3-0]M bit is set. The Request Masking hardware asserts the valid request line to the pending interrupt if its priority level is not masked by the CPU status register.

The Prioritize hardware selects the pending interrupt of the highest priority

based on the valid request lines and the priority level in the IPRs. If a NMI interrupt is pending, it automatically gets highest priority of any pending interrupt. The interrupt level and the handler autovector of the selected interrupt is transmitted to the CPU for further interrupt handling.

Software interrupt handling

Consider Figure 2.2, which shows the basic actions an interrupt takes through the kernel. An interrupt triggers at t1. Unless interrupts are disabled in the processor, the kernel is interrupted and jumps to the predefined *entry point* for interrupt handlers found in `arch/avr32/kernel/entry-avr32b.S`. On AVR32 , there are four entry points defined as `irq_level0`, `irq_level1`, `irq_level2`, and `irq_level3`. On the AVR32 architecture, all interrupts are initialized to level 0, that is lowest priority. Further, The kernel is now in interrupt context of operation.

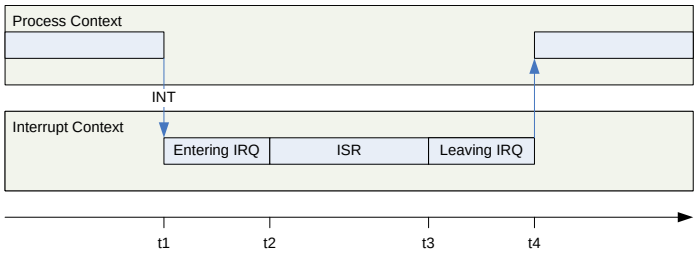

Figure 2.2: Interrupt handling

The initial entry point saves the interrupt number and the current registers values which belongs to the interrupted process, on the stack. Then, the kernel calls `do_IRQ()` defined in `arch/avr32/mach-at32ap/intc.c`. First, `do_IRQ()` disable interrupts on the processor. This ensures that the consecutive sequence of kernel code is treated as a critical section. Next, `do_IRQ()` calls the generic interrupt handler for this interrupt line. On the AVR32 architecure, all interrupts are handled through the `handle_simple_irq()` function declared in `kernel/irq/chip.c`. This function runs the installed interrupt handler for the line. Back in `do_IRQ()`, the function cleans up and returns to the initial entry point, which then terminate the the interrupt.

The termination phase of interrupts are written in assembly in `arch/avr32/kernel/entry-avr32b.S`. Here, `schedule()` is called only if the `preempt_count` is zero and a reschedule is pending. At last, the initial registers are restored and the kernel resumes whatever was interrupted.

2.4 Bottom halves and deferring work

Bottom halves are responsible to perform any interrupt-related work not performed by the interrupt handler itself. Recall from 2.3 that interrupts are disabled during interrupt handling. Bottom halves run with all interrupts enabled.

The Linux provides three mechanisms that may be used to implement bottom-half processing: softirqs, tasklets, and work queues. In the following, the individual mechanisms are briefly introduced.

2.4.1 Softirqs

Softirqs runs in interrupt context, and therefore cannot sleep. In the current kernel, softirqs are used for the most timing-critical and important bottom-half processing on the system.

In the mainline kernel, pending softirqs are checked for and executed in the following places:

- In the return from hardware interrupt code.
- In the `ksoftirqd` per-processor kernel thread.
- In any code that explicitly checks for and executes pending softirqs.

The softirq execute in threaded context only if the `ksoftirqd` method is used for invocation. The kernel thread helps in processing softirqs when the system is overwhelmed with softirqs. However, regardless of method of invocation, the pending softirqs are executed in `do_softirq()`. In this function, the pending softirqs' handlers are invoked in a loop until there are no more pending softirqs, and the work is done. Consequently, a softirq never preempts another softirq. Since softirqs runs in interrupt context with interrupt enable, only an interrupt handler can preempt a softirq.

2.4.2 Tasklets

Tasklets builds on top of softirqs, and hence *are* softirqs. They have, however, a simpler interface and are, compared to softirqs, a much more common form of bottom half.

2.4.3 Work queue

Work queues defer work into a kernel thread. Thus, this mechanism benefits of process context and can therefore be scheduled and preempted as normal processes.

A kernel thread called the *worker thread*, handles deferred work that is queued from elsewhere. By default, one worker thread is created per processor, called the `event/n` where `n` is the processor number. However, the work queue subsystem supports more than one worker threads per processor. Processor-intense and performance-critical work might benefit from its own thread.

2.5 Kernel synchronization methods

The kernel has shared resources that require proper protection from concurrent access from multiple threads of execution. Otherwise, threads may overwrite

each other's changes. This would produce incorrent results or data might be accessed when it is in an inconsistent state. A sequence of code that access and manipulate shared data is called a *critical section*. If two threads of execution are simultaneously in the same critical section, a *race condition* is said to occur. The synchronization required to protext a critical section from race conditions is known as *mutual exclusion*.

In a uniporcessor Linux system, there are numerous sources of concurrency, and therefore, possible race conditions. They are (Love, 2005, chap. 8)

- *Interrupts* – An interrupt can occur asynchronously at any time unless interrupts are explicity disabled, and therefore at any point in the currently executing code.

- *Softirqs and tasklets* – The kernel can raise or schedule a softirq or tasklets at any time unless the kernel already handles an interrupt, and therefore at any point in the currently executing code.

- *Kernel preemption* – Because the kernel is preemptive,

- *Sleeping and synchroization with user-space* – A task in the kernel can sleep and invoke the scheduler.

Linux implements a large number of synchronizations methods. In the following, two of the most common methods used in the kernel, spin locks and semaphores, are described. In addition, the kernel implements more specialized locking methods like atomic actions, completion variables and seq locks. Further, the big kernel lock (BKL), preemption disable and tacklet barriers completes the list.

2.5.1 Spin lock

Spin locks are shared variables acting as flags. When a thread of execution enters a critical section, it attemps to acquire the lock. If the lock is not *contended*, that is not already held, the thread immendiatly acquire the lock and continues. If the lock is contented, the thread loops around and recheck the lock. This is busy waiting, also known as spinning and hence the name spin lock. When the thread that first acquired the lock leaves the critical section, the spin lock is released. Then, the spinning thread can acquire it and enter the section. Thus, the spinning prevents more than one thread of execution to be simultaneously in the same critical section. This provides the necessary protection from concurrency on multiprocessing machines. A uniprocessor system running a preemptive kernel behaves like multiprocessor systems, as far as concurrency is conserned. Therefore, on uniprocessor machines, spin locks simply disable and enable kernel preemption when acquired and released. Consequently, the duration that spin locks are contended is equivalent to the scheduling latency of the system.

In Linux, each spin lock is represented by a `spinlock_t` structure defined in `<linux/spinlock_types.h>`. Table 2.1 gives a list of standard spin lock methods.

Table 2.1: Listing of spin lock methods

Method	Description
`spin_lock_init()`	Dynamically initializes given spin lock.
`spin_lock()`	Acquires given lock.
`spin_lock_irq()`	Disable local interrupts and acquires given lock.
`spin_lock_irqsave()`	Saves current state of local interrupts, disable local interrupts and acuires given lock.
`spin_unlock()`	Releases given lock.
`spin_unlock_irq()`	Releases given lock and enables local interrupts.
`spin_unlock_irqrestore()`	Releases given lock and restores local interrupts to given previous state.
`spin_trylock()`	Tries to acquire given lock; if unavailable, returns nonzero.
`spin_is_locked()`	Returns nonzero if the given lock is currently acquired; otherwise returns zero.

2.5.2 Semaphore

Semaphores are sleeping lock in Linux. When a thread attempts to acquire a resource already held by a semaphore, the thread is suspended. It becomed runnable again when the resource is released. Since the thread of execution sleeps on lock contention, semaphores can only be obtained in process context. Therefore, interrupt handlers and deferred functions that execute in interrupt context cannot use them. Unlike spin locks, semaphores do not disable kernel preemtion. Consequently, the duration that semaphores are contended does not affect scheduling latency.

Semaphores allow for an arbitrary number of simultaneous lock holders. At declaration time, the number of simultaneous lock holders that are permitted is specified in a `count` variable. Usually, the `count` value is set to 1 or 0, that is, a free resource with exclusive access or busy resource with exclusive access currently grated to the thread that initialize the semaphore. In such cases, the semaphores is called either a *binary semaphore* or a *mutex*. Alternatively, the semaphore can be initialized with a `count` nonzero value greater than one. Such general semaphores are often called *counting semaphores*. Counting semaphores allow multiple threads of execution to be in the critical section at once. Hence, they do not enforce mutual exclusion, but are used for enforce limits in certain code.

Semaphores are represented by a `semaphore` structure defined in the architecture-dependent `<asm/semaphore.h>`. Table 2.2 gives a list of standard semaphore methods.

Table 2.2: Listing of semaphore methods

Method	Description
`sema_init(struct semaphore *, int)`	Initializes the dynamically created semaphore to the given count.
`init_MUTEX(struct semaphore *)`	Initializes the dynamically created semaphore with a count of one.
`init_MUTEX_LOCKED(struct semaphore *)`	Initializes the dynamically created semaphore with a count of zero.
`down_interruptible(struct semaphore *)`	Tries to acquire the given semaphore and enter interruptible sleep if it is contended.
`down(struct semaphore *)`	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended.
`down_trylock(struct semaphore *)`	Tries to acquire the given semaphore and immediately return nonzero if it is contended.
`up(struct semaphore *)`	Releases the given semaphore and wakes a waiting task, if any.

Chapter 3

Real-Time Preemptive Kernel

This chapter delves into the fully preemptible kernel proposed by the RT-preempt patch. The techniques used by the patch are explained in detail and why they matter. Throughout the chapter, code fragments are presented to show how the AVR32 Linux kernel adopt these techniques.

Note that the fully preemptible kernel is just one out of three mechanisms used to incorporate real-time capabilities into the Linux kernel. The work concerning high-resolution timers and dynamic ticks are deferred to Chapters 5.

3.1 Overview of the RT-preempt patch

The RT-preempt patch allows nearly all of the kernel to be preempted, with the exception of a few critical sections. The fully preemptible kernel included in the RT-preempt patch features:

- Preemptible critical sections.
- Priority inheritance for in-kernel spinlocks and semaphores.
- Preemptible interrupt service routines.

According to (McKenney, 2005b), the amount of architecture-dependent code inspection required are:

- The low-level interrupt-handling code.
- Any code that disable interrupts.
- Any code that disable preemption.
- Any code that holds a lock, mutex, semaphore, or other resource that is required by the RT-preempt patch. Further, the code that implements the lock, mutex, semaphore, or other resources are subject to inspection.

- Any code that manipulates hardware that can stall the bus, delay interrupts, or otherwise interfere with forward progress. Further, it is also necessary to inspect user-level code that directly manipulates such hardware.

In the following, the techniques from the RT-preempt patch are described together with code fragments from the AVR32 Linux kernel code. Clearly, the code needs modifications in order to implement the mechanisms proposed by the RT-preempt patch . Therefore, the code fragments are coloured to separate the deleted code from the added code. The *red* labeled code are deleted, the *blue* labeled code are inserted. Code in *black* are unchanged.

3.1.1 Preemptible critical sections

As seen in Section 2.5.1, the idea behind spin locks, i.e. `spinlock_t` and `rwlock_t`, are to protect short critical sections. A problem with the use of spin locks in Linux is that they also protect large critical sections. Since the duration that locks are held is equivalent to the scheduling latency of the system, the unpredictable latencies caused by spin locks becomes a problem for real-time applications. In the RT-preempt patch, this problem is solved by converting spin locks into mutexes. Since mutexes are sleeping locks, critical sections within the kernel protected by spin locks are now preemptible. This preemptibility means that calls to `spin_lock()` might block, and, hence, is illegal to acquire with either preemption or interrupts disabled[1]. Further, `spin_lock_irqsave()` does not disable hardware interrupts when used on a `spinlock_t`.

There are spin locks in the RT-preempt kernel that must remain a traditional spin lock, and not converted into a sleeping spin lock. Examples include inside the scheduler and in the implementation of mutexes themselves. In the RT-preempt patch, such spin locks need to change their type from `spinlock_t` to `raw_spinlock_t`. When the existing spin lock functions are invoked on a `raw_spinlock_t`, the spin lock will act as a traditional spin with preemption disabled. However, when the functions are invoked in a `spinlock_t`, the spin lock will become a mutex.

Consider the following code sequence taken from McKenney (2005a), supplied by Ingo Molnar:

```
spinlock_t mylock1;
spinlock_t mylock2;

spin_lock(&mylock1);
current->state = TASK_UNINTERRUPTIBLE;
spin_lock(&mylock2);
foo();
spin_unlock(&mylock2);
spin_unlock(&mylock1);
```

[1] There is one exception to this rule; calls to `_trylock` variants are permitted with either preemption or interrupts disabled, unless they are repeatedly invoked in a tight loop (McKenney, 2005a).

Since the `spin_lock(&mylock2)` function can sleep, the value of `current->state` may change, which might affect the `foo()` function. Therefore, an additional task state was added in the RT-preempt kernel; `TASK_RUNNING_MUTEX`. This state is used to allow the scheduler to preserve the prior value of `current->state`.

As mentioned, it is illegal to invoke the `spin_lock()` function while preemption or interrupts are disabled. However, there are situations were a task cannot make progress until a spin lock is acquired. In the RT-preempt patch, this has been solved by deferring the operation requiring the `spin_lock()` until preemption has been re-enabled. Therefore, several functions are extended with versions which end in `_delayed`. These functions queue up their non-delayed counterpart calls to be executed at a later time when it is legal to acquire spin locks. For example `put_task_struct_delayed()` queues up calls to `put_task_struct()` if preemption are disabled and the function acquires a spin lock.

The scheduling may be affected by deferred operations. Say, for example, that a task preempts a lower-priority task, but cannot make any progress because it waits for a lock held by the lower-priority task. Thus, the task would immediately enter block state waiting for the lock to be released. To avoid such unnecessary preemption, the RT-preempt patch extends this `flag` field by a new flag; `TIF_NEED_RESCHED_DELAYED`. This flag does a reschedule just like the `TIF_NEED_RESCHED` flag, but waits until the task is ready to return to user space, or until the next `preempt_check_resched_delayed()`, whichever comes first.

The `TIF_NEED_RESCHED_DELAYED` flag is defined in `include/asm-avr32/thread_info.h` together with a corresponding `_TIF_NEED_RESCHED_DELAYED` bit mask:

```
1  #define TIF_SINGLE_STEP          6       /* single step after next break */
2  #define TIF_MEMDIE               7
3  #define TIF_RESTORE_SIGMASK      8       /* restore signal mask in do_signal */
4  #define TIF_NEED_RESCHED_DELAYED 9       /* reschedule on return to userspace */
5  #define TIF_USERSPACE            31      /* true if FS sets userspace */

6  #define _TIF_SINGLE_STEP         (1 << TIF_SINGLE_STEP)
7  #define _TIF_MEMDIE              (1 << TIF_MEMDIE)
8  #define _TIF_RESTORE_SIGMASK     (1 << TIF_RESTORE_SIGMASK)
9  #define _TIF_NEED_RESCHED_DELAYED (1 << TIF_NEED_RESCHED_DELAYED)
```

`arch/avr32/kernel/entry-avr32b.S` contains a number of kernel control paths that checks the `TIF_NEED_RESCHED` flag. In the RT-preempt patch these paths must be extended to check the `TIF_NEED_RESCHED_DELAYED` flag as well. Since the `bld` intruction in 3.1 only can check one bit, it cannot check the `TIF_NEED_RESCHED` and `TIF_NEED_RESCHED_DELAYED` flag at the same time. Instead, the flags bits are shifted into a unused register. Then, this register is compared to register `r1` which holds the flags.

Code Sequence 3.1: `arch/avr32/kernel/entry-avr32b.S`

```
10   cp.w   r2, 0
```

```
11    brne   1b
12    ld.w   r1, r0[TI_flags]
13    bld    r1, TIF_NEED_RESCHED
14    brcc   1b
15    mov    r4, _TIF_NEED_RESCHED | _TIF_NEED_RESCHED_DELAYED
16    tst    r1, r4
17    brne   1b
18    lddsp  r4, sp[REG_SR]
19    bld    r4, SYSREG_GM_OFFSET
20    brcs   1b
```

The `cpu_idle()` function in 3.2 is affected by the `TIF_NEED_RESCHED_DELAYED` flag. Now, the processor may sleep only is the `TIF_NEED_RESCHED` *and* `TIF_NEED_RESCHED_DELAYED` flags are not set.

Code Sequence 3.2: `arch/avr32/kernel/process.c`

```
21  void cpu_idle(void)
22  {
23    /* endless idle loop with no priority at all */
24    while (1) {
25      /* TODO: Enter sleep mode */
26        while (!need_resched())
27        while (!need_resched() && !need_resched_delayed())
28          cpu_relax();
29
30        preempt_enable_no_resched();
31        schedule();
32        preempt_disable();
33        local_irq_disable();
34        __preempt_enable_no_resched();
35        __schedule();
36        preempt_disable();
37        local_irq_enable();
38    }
39  }
```

3.1.2 Priority inheritance for in-kernel spin locks and semaphores

Priority inversion occurs when a higher-priority process is forced to wait on a lower-priority process (Stallings, 2005). A simple example of priority inversion occurs when a lower-priority process holds a resource and a higher-priority process attempts to lock the same resource. The higher-priority process is blocked until the lower-priority process is finished with the resource and releases it. Then, the higher-priority may quickly resume, and acquires the resource.

Unbounded priority inversion occurs when the higher-priority process must wait an undetermined amount of time for the lower-priority process to release the resource. Then, the duration of the priority inversion depends not only on the time required to handle the shared resource, but also on the unpredictable

actions of other unrelated processes as well. Figure 3.1 shows the sequence that causes the priority inversion:

t1: T3 begins executing.

t2: T3 enters its critical section and locks semaphore `sem`.

t3: T1 preempts T3 and begins executing.

t4: T1 attempts to enter critical section, but is blocked because T3 holds semaphore `sem`; T3 resumes execution in its critical section.

t5: T2 preempts T3 and begins executing.

t6: T2 is suspended for reasons unrelated to T1 and T3, and T3 resumes execution.

t7: T3 leaves its critical section and unlock semaphore `sem`. T1 preempts T3, locks the semaphore, and enters its critical section.

t8: T1 leaves its critical section, and unlocks the semaphore.

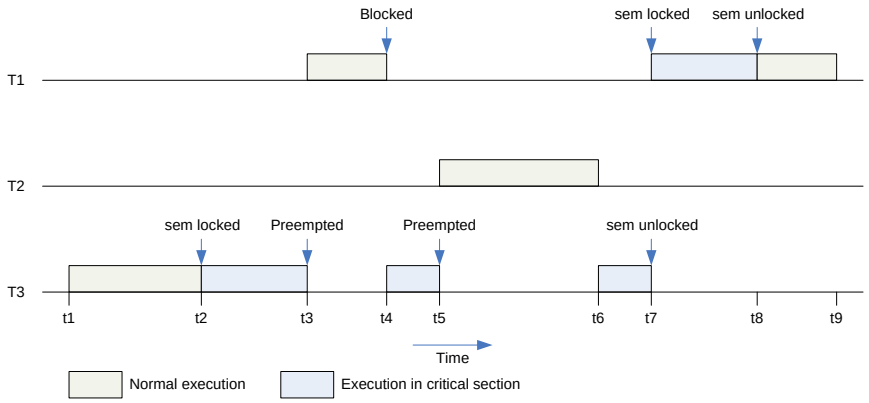

Figure 3.1: Unbounded priority inversion.

In the current mainline kernel, preemption is disabled when processes protect share resources with spin locks. This prevents priority inversion to occur. Thus, T2 cannot preempt the lowest-priority T3 task at t5 in Figure 3.1, and unbounded priority inversion is avoided. However, because T1 cannot preempt the T3 at t3, this scheme is inadequate for some real-time workloads, due to the impact of scheduling latencies. Additionally, unlike spin locks, semaphores does not suppress preemption because a task might sleep while holding a semaphore. When another higher-priority process acquires the same semaphore, it goes to sleep and eventually let the lower-priority process continue. Hence, priority inversion may occur even in the absence of preemption.

In practical systems, there are two approaches to prevent unbounded priority inversion and still allow preemption; priority ceiling protocol and priority

inheritance protocol. In the priority ceiling approach, each resource must knows the highest priority process that will acquire it. Then, the resources are assigned priorities one level higher than the priority of its highest-priority user. When a process acquires a resource, the process is assigned the priority of the resource. This prevents any other process that might acquire the same resource from preempting the process. Since almost any resource or lock in Linux may be taken by any process, this approach highly suppress preemption while a shared resource is held.

The RT-preempt patch implements *priority inheritance*. The basic idea of priority inheritance is that lower-priority tasks that are holding shared resources, temporarily inherits the priority of any higher-priority pending on the same resources. The priority change takes place when the higher-priority task acquired the resource, and ends when the lower-priority releases the resource. Figure 3.2 shows how the unbounded priority inversion illustrated in Figure 3.1 is solved by priority inheritance. The sequence is as follows:

t1 T3 begins executing.

t2 T3 enters its critical section and locks semaphore `sem`.

t3 T1 preempts T3 and begins executing.

t4 T1 attempts to enter critical section, but is blocked because T3 holds semaphore `sem`. T3 is temporarily assigned the same priority as T1. T3 resumes execution in its critical section.

t5 T2 is ready to run, but because T3 has inherited T1's priority, T2 is unable to preempt T3.

t6 T3 ends its critical section, and unlocks the semaphore. Then its priority level is downgraded to its previous default level. T1 preempts T3, locks the semaphore, and enters its critical section.

t7 T1 leaves its critical section, and unlocks the semaphore.

t8 T1 is suspended unrelated to T2, and T2 begins execution.

Semaphores allow one or more tasks access to a resource, and, hence, semaphores have no concept of an owner. Therefore, a semaphore is never bounded to a thread. In Linux, semaphores are often used for maintaining mutual exclusion to a resource or to coordinate two or more threads. For mutual exclusion, the ability of a semaphore to handle multiple threads produces an unnecessary overhead when only acting as a mutex. The RT-preempt patch introduces a new primitive for the kernel called *mutex*. Because of its simpler design, the mutex is much cleaner and slightly faster than a semaphore. In contrast to a semaphore, the mutex may be owned by one, and only one, thread at a time. This property of the mutex is one of the requirements of the priority inversion algorithm implemented by the RT-preempt patch. Since locks have a one-to-one relationship with its over, the priority inheritance chain stays a single path, and does

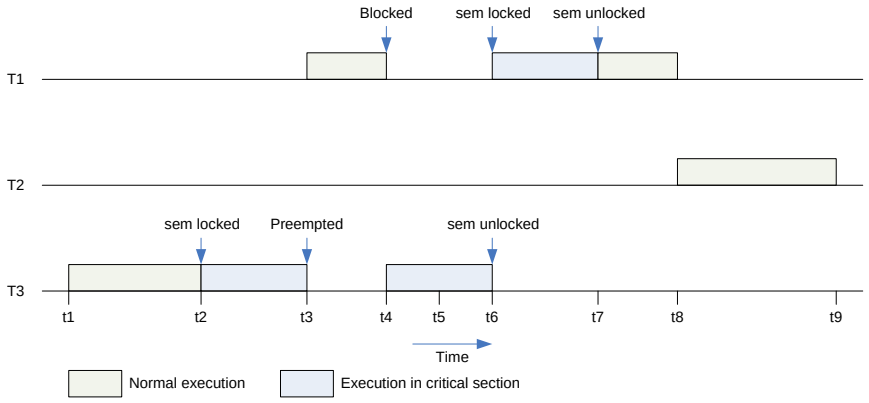

Figure 3.2: Priority inheritance.

not branch with multiple lock owners needing to inherit a priority of a waiting thread.

3.1.3 Preemptible interrupt handlers

When executing an interrupt handler or bottom half in the mainline kernel, the kernel is in interrupt context. Although code executed in interrupt context should be quick and simple, the servicing of low priority devices, for examples hard-drives, interrupt handling can cause large latencies for all tasks. Generally speaking, in the mainline kernel a high priority task may be greatly affected by a low priority task. The RT-preempt patch converts interrupt service routines into *kernel threads* to address this issue. Hard vs. soft interrupt.

Kernel threads

Kernel threads are used by the kernel to perform operations in the background. For example, flushing disk caches and running tasklets are executed intermittently by the `pdflush` and `ksoftirqd` kernel thread, respectively. Kernel threads differs from standard processes that they run only in kernel space, and do not have an address space. However, they are *schedulable* and *preemptable* as normal processes.

Threaded top halves handlers

A device driver registers an interrupt handler and enables a given interrupt line for handling by calling `request_irq()`. The `request_irq()` calls `setup_irq()` to register the `struct irqaction`. Among other work, `setup_irq()` calls `start_irq_thread()` to create a kernel thread to service the interrupt line. The thread's work is implemented in `do_irqd()`. Only one thread can be created per interrupt line, and shared interrupts are still handled by a single thread. Since kernel threads execute in process context, the interrupts handlers are now preemptable and schedulable.

Almost all interrupt handlers runs as kernel threads in the RT kernel . However, there are cases where the handler must be serviced in interrupt context. By setting the interrupt descriptor flag `IRQ_NODELAY`, the interrupt handler is forces to run in interrupt context and not as a thread. An `IRQ_NODELAY` interrupt can greatly degrade both interrupt and scheduling latencies if the associated interrupt handler does to much work. The most notable use of this is the timer interrupt due to its tie to schedule and other core kernel components. On AVR32 Linux, the timer interrupt is defined in `arch/avr32/kernel/time.c`:

```
40  static struct irqaction timer_irqaction = {
41    .handler    = timer_interrupt,
42    .flags      = IRQF_DISABLED,
43    .flags      = IRQF_DISABLED | IRQF_NODELAY,
44    .name       = "timer",
45  };
```

The `do_IRQ()` declared in `arch/avr32/mach-at32ap/intc.c` invokes the function `handle_simple_irq()`, which in the RT-preempt patch performs the following work:

1. Acquires the spin lock `desc->lock` to protect access to the IRQ descriptor.
2. Checks whether the interrupt really must be handled. There are three cases in which nothing have to be done:
   (a) `IRQ_DISABLED` is set.
   (b) `IRQ_INPROGRESS` is set.
   (c) `desc->action` is `NULL`.
3. Clears the `IRQ_REPLAY`, `IRQ_WAITING`, and `IRQ_PENDING` flags. Then, sets the `IRQ_INPROCESS` flags.
4. Invokes `redirect_hardirqs()`, which calls `wake_up_process()` on the associated thread iff. threaded interrupt are enabled *and* the current IRQ is threaded.
5. If `wake_up_process()` is *not* called in step 4, i.e. either threaded interrupt are disabled or the current IRQ is flagged `IRQ_UNDELAY`, releases the interrupt spin lock. Then, executes the interrupt service routine by invoking `handle_IRQ_event()`. Next, acquires the spin lock again and clears the `IRQ_INPROGRESS` flag.
6. Finally, releases the spin lock.

If an interrupt is redirected in 4, the interrupt will terminate before its interrupt handler is invoked. Therefore, the interrupt line must be masked; otherwise the interrupt will trigger again. On the AVR32 architecture, only groups of interrupts can be masked either by the GM or the I[3-0]M in the processor's status register. Individual interrupt lines cannot be masked. Consequencly, when a threaded interrupt triggers on the AVR32 architecture, the interrupt triggers in

loop without being handled. Hence, the processor services this interrupt line over and over again without any progress.

An interrupt controller for the AVR32 architecture that is capable of masking specific interrupt lines for the entire processor is the most best solution to this problem.

Threaded bottom halves and deferring work handlers

As seen in 2.4, bottom halves and deffering work handlers can not assume to be run in threaded context. In the RT-preempt patch, the softirqs are only handled in kernel threads. Each handler has its own thread to service them.

3.1.4 Concluding remark

The RT-preempt patch by Ingo Molnar incorporates many logical changes in mainline Linux and introduces new concepts. The focus in this chapter has been to describe some of these new consepts and how they affect the architecture-dependent code. Therefore, the description of the fully preemptible kernel implementaion is not complete, nor the required changes in the AVR32 Linux kernel. The CD-ROM contains the proposed changes in AVR32 Linux.

Chapter 4

Timer Management and Timers

This chapter consists of four parts. The first section describes the hardware devices used for timing. The next section gives an detailed picture of the Linux Time System. Then, the AVR32 Linux kernel is transformed to fully support John Stult's Generic Time-of-day approach. The last section discusses the timers in Linux.

4.1 Timer Circuits

The kernel interact with a programmable timer hardware circuit called the *system timer*, that issues an interrupt at a fixed frequency. The interrupt handler for this timer, called the *timer interrupt* updates the system time and performs periodic work.

On the architecture, the system timer functionality is set up by using the `COUNT` together with the `COMPARE` system registers. Both registers can be read and written by using the privileged `mfsr` and `mtsr` instructions. The `COUNT` register increments once every clock cycle, and the incrementation can not be disabled. The `COMPARE` register holds a value that the `COUNT` register is compared against. When the `COMPARE` and `COUNT` registers match, a compare interrupt request is generated. Then, the interrupt request is routed to the Interrupt Controller (INTC), which forwards the request back to the processor at a priority level determined by the INTC.

The frequency of the system timer, called the *tick rate* is programmed on system boot based on the static preprocessor defined `HZ` variable. The kernel defines the value in `<asm/param.h>`. The value of `HZ` is architecture dependent, and in `include/asm-avr32/param.h`, the AVR32 architecture defines:

```
#define HZ 250
```

Hence, since the period is defined as 1/`HZ` seconds, the timer interrupt on AVR32 has a frequency of 250 `HZ` with a period of $1/250 = 0.004$, that is,

every fourth millisecond.

Clearly, an increasing tick rate results in a system timer with a higher resolution. Stating with the initial version of Linux, the default frequency of the timer interrupt was 100 `HZ`. During the 2.5 development series, the frequency was raised for some architectures to 1000 `HZ`. However, the raising caused certain regressions, and the `HZ` value was changed to 250 `HZ`.

4.2 Linux Time System

Linux executes several time-related activities periodically.

- Updates the time of day.
- Determines whether the current process has exceeded its timeslice, and, if is has, causing a reschedule.
- Checks whether the interval of time associated with each software timer has expired.
- Updates the system uptime.
- Updates resource usage and processor time statistics.

All time-related activities are triggered by the timer interrupt raised by the system timer, as seen in Figure 4.1. Some of this work is done on *every* timer interrupt, while others functions executes periodically, but only every n timer interrupts.

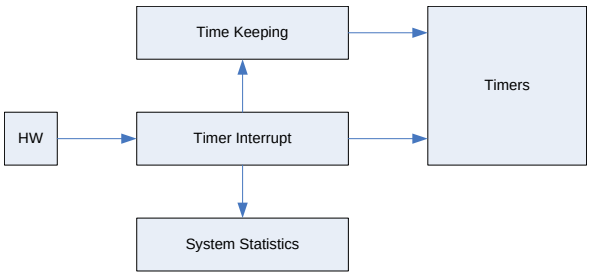

Figure 4.1: Linux Time System.

4.2.1 Data Structures of the Time System

The jiffies variable

The global variable `jiffies` is a counter that holds the number of ticks since the system booted. It is incremented by one when a timer interrupt occurs, that is, on every tick. Since the interrupt interval differs between architectures, the amount of time one `jiffy` represents is not absolute.

The `jiffies` variable is decleared in `<linux/jiffies.h>` as:

```
extern unsigned long volatile jiffies;
```

Since the AVR32 is a 32-bit architecture, the `unsigned long jiffies` variable is 32 bits in size. With a tick rate of 250 `HZ`, the `jiffies` variable wraps around in approximately 199 days. However, the time management code needs the real number of system ticks since the system boot, regardless of the overflow of `jiffies`. Hence, a 64-bit variable `jiffies_64` is defines in `linux/jiffies.h` as:

```
extern u64 jiffies_64;
```

With any reasonable `HZ` value, the `jiffies_64` variable will never overflow in anyone's lifetime.

The `jiffies` variable is overlaid the by the linker script to the 32 less significant bits of the `jiffies_64` variable. Thus, any incrementation of the `jiffies_64` also increase the `jiffies` variable, since the latter correspond to the lower half of `jiffies_64`. Moreover, code can access the `jiffies` variable atomically, but not the `jiffies_64`. Hence, read and write operations on the `jiffies_64` variable requires synchronization techniques to ensure that the operations are done atomically. This is done by the `xtime_lock` seqlock.

The xtime variable

The current time of day, as seen on a wrist-watch is stored in the `xtime` variable. It is a structure of type `timespec` defined in `kernel/timer.c`:

```
struct timespec xtime;
```

The `timespec` data structure is defined in `<linux/time.h>` as:

```
struct timespec{
  time_t tv_sec; /* seconds */
  long tv_nsec;  /* nanoseconds */
}
```

The `xtime.tv_sec` value holds the number of seconds that have elapsed since January 1, 1970 (UTC). This date is called the *epoch*. The `xtime.tv_nsec` value holds the number of nanoseconds that have elapsed in the last second.

The `xtime_lock` seqlock avoids any race conditions that could occure due to concurrent access to the `xtime` variable. The `xtime_lock` also protects the `jiffies_64` variable.

4.2.2 Timekeeping

A clocksource is a driver-like architecture generic abstraction of a free-running counter. This code defines the clocksource structure, and provides management code for registering, selecting, accessing and scaling clocksources.

An abstraction layer and associated API are required to establish a common code framework for managing various clock sources. The centralization of this functionality allows the system to share significantly more code across architectures.

Clock source data structure

The clock source management code specifies a `clocksource` structure in order to handle possible clock sources in a uniform way. The structure is defined in `<linux/clocksource.h>` as:

```
struct clocksource {
  char *name;
  struct list_head list;
  int rating;
  cycle_t (*read)(void);
  cycle_t mask;
  u32 mult;
  u32 shift;
  unsigned long flags;

  /* timekeeping specific data, ignore */
  cycle_t cycle_last, cycle_interval;
  u64 xtime_nsec, xtime_interval;
  s64 error;

#ifdef CONFIG_CLOCKSOURCE_WATCHDOG
  /* Watchdog related data, used by the framework */
  struct list_head wd_list;
  cycle_t wd_last;
#endif
};
```

In this structure, the `rating` field allows the best registered clock source to be chosen by the clock source management. To avoid rating inflation, the clock source should be rated according to the list defined in Table 4.1. The clock

Table 4.1: Clock source rating

Rating	Description
1-99	Unfit for real use. Only available for bootup and testing purposes.
100-199	Base level usability. Functional for real use, but not desired.
200-299	Good. A correct and usable clocksource.
300-399	Desired. A reasonably fast and accurate clocksource.
400-499	Perfect. The ideal clocksource. A must-use where available.

source's cycle value can be read from the `read` function pointer. The `mask` value ensures that subtraction between counters values from non 64 bit counters do not need special overflow logic. The `mult` and `shift` are used to convert the

clock source's period to nanoseconds per cycle. The `flags` member describes special properties of the clock source.

Table 4.2: Clock source flags

Value	Description
CLOCK_SOURCE_IS_CONTINUOUS	Free running counter
CLOCK_SOURCE_MUST_VERIFY	The clock source must be for reliability.
CLOCK_SOURCE_WATCHDOG	Instead of using hardwired assumptions of available hardware, a generic verification mechanism is provided.
CLOCK_SOURCE_VALID_FOR_HRES	The clock source is capable for high resolution.

The `jiffies` based clock source is the lowest common denominator clock source that should function on all systems. The `clocksource` structure for the jiffies clock source is initialized in `jiffies.c` as:

```
struct clocksource clocksource_jiffies = {
  .name       = "jiffies",
  .rating     = 1, /* lowest valid rating*/
  .read       = jiffies_read,
  .mask       = 0xffffffff, /*32bits*/
  .mult       = NSEC_PER_JIFFY << JIFFIES_SHIFT,
  .shift      = JIFFIES_SHIFT,
};
```

Here, the function `jiffies_read` returns the number of `jiffies` in terms of `cycle_t`. The jiffies clock source uses a simple `NSEC_PER_JIFFY` multiplier conversion to specify the nanosecond over cycle ratio, i.e. the `shift` value should be 0. This conflics with the Network Time Protocol (NTP) adjustments code for synchronizing clocks since they are in units of $1/2^{\texttt{shift}}$. However, by shifting the `mult` and `shift` values by a chosen constant, the values are compatible with the NTP adjustments code.

The `clocksource_avr32` clock source exploits the hardware timers provided by the AVR32 architecture. The structure is initialized in `arch/avr32/kernel/time.c` as:

```
static struct clocksource clocksource_avr32 = {
  .name               = "avr32",
  .rating             = 350,
  .read               = read_cycle_count,
  .mask               = CLOCKSOURCE_MASK(32),
  .shift              = 16,
  .flags              = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

The `clocksource_avr32` has a rating of 350, which is a desired clock source according to Table 4.1. The function `read_cycle_count` reads the `COUNT` system register:

```
static cycle_t read_cycle_count(void)
{
  return (cycle_t)sysreg_read(COUNT);
}
```

Clock source management

The clock source management code provides the interface for clock source registration and selection. Clock sources are registered by calling `clocksource_register()` during kernel initialization or from a kernel module. The former is used for the jiffies and avr32 clock sources. During registration, the clock source management code will choose the best clock source available in the system using the `rating` field. Alternatively, a `sysfs` interface allows the user to list registered clock sources, and manually override the default clock source selection.

4.2.3 The Timer Interrupt

In a uni-processor Linux system, all time-related activities are triggered by the interrupts raised by the system timer.

Initialization phase

During kernel initialization, the `time_init()` function is invoked to set up the necessary time-related infrastructure. On the AVR32 architecture, the following operations are performed:

1. Initializes the `xtime` variable. The `tv_sec` field of the `xtime` is read from the Real Time Clock by means of the `rtc_get_time()` function. Since the RTC is not supported, the function points to a `null_rtc_get_time`, which returns number of seconds since the midnight of January 1, 1970 to the midnight of January 1, 2004. The `tv_nsec` is set.

2. Initialize the `wall_to_monotonic` variable. This variable is of the same type `timespec` as `xtime`, and it stores the number of seconds and nanoseconds to be added to `xtime` on order to get a monotonic flow of time.

3. Calculates the `cycles_per_jiffy` and invokes the `avr32_hpt_init()` function to set up the high precision timer for the first timer interrupt.

4. Calculates the `mult` member of the avr32 clock source by means of the `clocksource_hz2mult()` function. Then, invokes `clocksource_register()` to register the avr32 clock source.

5. Invokes `setup_irq(0, &timer_irqaction)` to set up the interrupt handler corresponding to IRQ0. The `timer_irqaction` irqaction structure is defined as:

```
static struct irqaction timer_irqaction = {
  .handler  = timer_interrupt,
  .flags    = IRQF_DISABLED,
  .name     = "timer",
};
```

Now, the `timer_interrupt` interrupt handler is registered, and, thus, runs every time the timer interrupt hits.

The timer interrupt handler

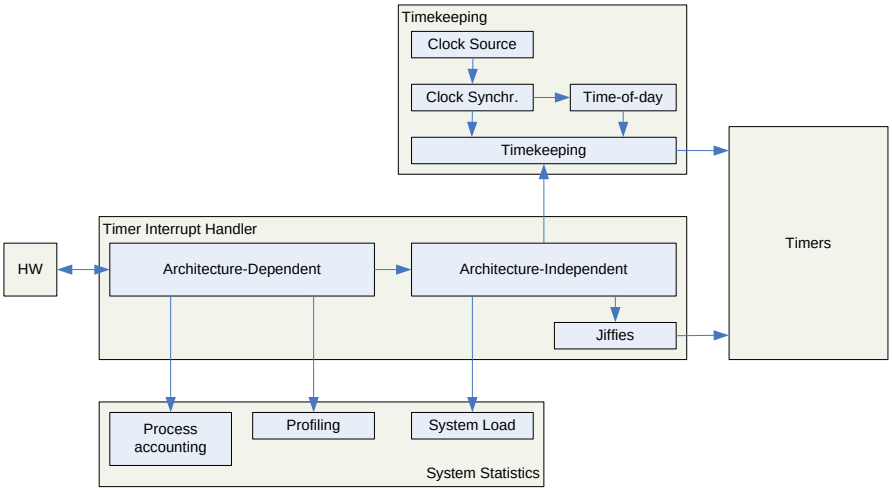

Figure 4.2: Linux Time System.

The timer interrupt is devided into an architecture-dependent part and an architecture-independent part since its exact job depends on the given architecture. On the system at hand, the interrupt handler is declared in `arch/avr32/kernel/time.c`, and does the following work:

1. Acknowledge the current timer interrupt and set the next one by increase the `COMPARE` system register by `cycles_per_jiffy`.
2. Obtain the `xtime_lock` seqlock by invoking `write_seqlock()`, to protect access to `jiffies_64` and `xtime`.
3. Invokes the architecture-independent timer routine, `do_timer()`, which in turn performs the following work:
   (a) Increments the `jiffies_64` by one. This is safe since the `xtime_lock` lock is previously obtained in the interrupt handler.
   (b) Invokes the `update_time()` function to update the system date and time, and to compute the current system load.
4. Releases the `xtime_lock` seqlock by invoking `write_sequnlock()`.

5. Invokes the `local_timer_interrupt()` routine to perform profiling and process-accounting on a per-CPU basis. This is done by the following architecture-independent functions:

   (a) Invokes the `profile_tick()` function to profile kernel code.

   (b) Invokes the `update_process_times()` to check how long the current process has been running.

Updating the time and date

The `update_time()` function from step 3b in the list on the preceding page, is defined in `timer.c` as:

```
static inline void update_times(unsigned long ticks)
{
  update_wall_time();
  calc_load(ticks);
}
```

The `update_wall_time()` function uses the current clock source to increment the wall time stored in the `xtime` variable. Each invocations adds 4,000,000 nanoseconds to the `xtime.tv_nsec`. Then, if the value of `xtime.tv_nsec` becomes greater than 999,999,999, the `xtime.tv_sec` is incremented. The `calc_load()` function updates the average system load.

Profiling

Linux includes a code profiler to discover where the kernel spends its time in Kernel Mode. The `profile_tick()` function in step 5a in the list on the previous page collects the data for the code profiler.

Process accounting

The `update_process_timers()` function in step 5b updates some kernel statictics. The `run_local_timers()` that raises the `TIMER_SOFTIRQ` softirq is called from this function. The execution of timers are covered in Section 4.4 on page 39.

4.3 Transforming the AVR32 Linux clock source related code

The Generic Time-of-day subsystem (GTOD) project (Stultz et al., 2005) adresses jiffy and architecture independent timekeeping. This subsystem provides an *architecture-independent* function `timekeeping_init()` defined in `kernel/timer.c`, which is called at initialization time. Code Sequence 4.1 shows the function.

Code Sequence 4.1: `timekeeping_init()`

```
1  void __init timekeeping_init(void)
2  {
```

```
3       unsigned long flags;
4       unsigned long sec = read_persistent_clock();
5
6       write_seqlock_irqsave(&xtime_lock, flags);
7
8       ntp_clear();
9
10      clock = clocksource_get_next();
11      clocksource_calculate_interval(clock, NTP_INTERVAL_LENGTH);
12      clock->cycle_last = clocksource_read(clock);
13
14      xtime.tv_sec = sec;
15      xtime.tv_nsec = 0;
16      set_normalized_timespec(&wall_to_monotonic,
17        -xtime.tv_sec, -xtime.tv_nsec);
18
19      write_sequnlock_irqrestore(&xtime_lock, flags);
20    }
```

In Code Sequence 4.1, the lines 14 through 17 initialize the `xtime` and the `wall_to_monotonic` variables. Therefore, Points 1 and 2 in the timer interrupt initialization phase on page 34 are redundant, and can be removed. The changes in the `arch/avr32/kernel/time.c` are shown in Code Sequence Sequence 4.2:

Code Sequence 4.2: Remove

```
21    unsigned long mult, shift, count_hz;
22    int ret;
23
24    xtime.tv_sec = rtc_get_time();
25    xtime.tv_nsec = 0;
26
27    set_normalized_timespec(&wall_to_monotonic,
28      -xtime.tv_sec, -xtime.tv_nsec);
29
30    printk("Before time_init: count=%08lx, compare=%08lx\n",
31      (unsigned long)sysreg_read(COUNT),
32      (unsigned long)sysreg_read(COMPARE));
```

In the architecture-independent code given in Code Sequence 4.1, the `xtime.tv_sec` member is initialized by calling the `read_persistent_clock()` function. Hence, the `null_rtc_get_time()` function-name is replaced by the `read_persistent_clock()` name as shown in 4.3. In addition, the function is not declared `static` since its name should be visible outside of the file in which it is declared.

Code Sequence 4.3: `read_persistent_clock()`

```
33    static unsigned long null_rtc_get_time(void)
34    unsigned long read_persistent_clock(void)
35    {
36      return mktime(2004, 1, 1, 0, 0, 0);
37    }
```

At last, the AVR32 clock source registration is moved out of the `time_init()` function to streamline the code and to create a solid foundation for high resolution timers and dynamic ticks. The required changes are shown in Code Sequence 4.4, 4.5, and 4.6.

Code Sequence 4.4: Removed code from `time_init()`

```
38  count_hz = clk_get_rate(boot_cpu_data.clk);
39  shift = clocksource_avr32.shift;
40  shift = 16;
41  mult = clocksource_hz2mult(count_hz, shift);
42  clocksource_avr32.mult = mult;
43
44  printk("Cycle counter: mult=%lu, shift=%lu\n", mult, shift);
```

Code Sequence 4.5: Remove `clocksource_register()` from `time_init()`

```
45    (unsigned long)sysreg_read(COUNT),
46    (unsigned long)sysreg_read(COMPARE));
47
48  ret = clocksource_register(&clocksource_avr32);
49  if (ret)
50    printk(KERN_ERR
51           "timer: could not register clocksource: %d\n", ret);
52
53    ret = setup_irq(0, &timer_irqaction);
54    if (ret)
```

Code Sequence 4.6: `init_avr32_clocksource()`

```
55  static int __init init_avr32_clocksource(void)
56  {
57        unsigned long  shift, count_hz;
58        int ret;
59
60        count_hz = clk_get_rate(boot_cpu_data.clk);
61        shift = clocksource_avr32.shift;
62        clocksource_avr32.mult = clocksource_hz2mult(count_hz, shift);
63
64        ret = clocksource_register(&clocksource_avr32);
65        if (ret)
66              printk(KERN_ERR
67                     "timer: could not register clocksource: %d\n", ret);
68
69        return ret;
70    }
71  module_init(init_avr32_clocksource);
```

4.4 Timers

In software, a *timer* allows functions to be invoked at some future moment, after a certain time interval has elapsed, specified by a *expiration* value. In Linux, three types of timers are considered; *dynamic timers*, *hrtimers* and *interval timers*. The hrtimers subsystem was incorporated into 2.6.16 Linux.

Dynamic timers are used by the kernel to primary manage *timeouts* (Gleixner and Molnar, 2005; Gleixner and Niehaus, 2006). Timeouts timers are used to detect error condition inside the kernel, such as stalled and stuck I/O or network operation. They are alomst always removed before they expire. Thus, their demand on resolution are usually low. Hrtimers are used for *timers*. Timers is used to force the execution of specific functions upon the specified expiration, and hence, they usually expire. Timers are mostly related to applications, and therefore may demand high resolution timing. Interval timers may be created by processes in user space.

4.4.1 Dynamic timers

Dynamic timers may be dynamically created and destroyed. A dynamic timer is represented by a `struct timer_list` structure, which is defined in `linux/timer.h`:

```
struct timer_list {
  struct list_head entry;
  unsigned long expires;
  void (*function)(unsigned long);
  unsigned long data;
  struct tvec_t_base_s *base;
};
```

The `expires` field specifies when the timer expires. The expiration time is expressed in jiffies. The `function` field contains a pointer of the function to be executed when the timer expires, and the `data` field specifies the parameter to be passed to this function.

The `entry` field is used to insert the dynamic timer into one of the doubly linked circular lists that forms the data structure used to organize the timers according to the value of their `expiration` field. This algorithm is called the *cascading timer wheel* and is described next.

Cascading timer wheel

The cascading timer wheel data structure consists of doubly linked circular lists that group together dynamic timers based theirs `expires` values. The wheel groups timers into five linked lists. Each list represents the set of timers within a certain region of jiffies, where the size of the regions grow exponentially. Thus, the further into the future the timer is set to expire, the larger the region of the list in which it is stored. As the times goes on, the timers may be removed and reinserted, or cascaded, hence the name, from lists with larger `expires` values

to lists with smaller ones. Figure 4.3 illustrates in a schematic way the five groups of lists.

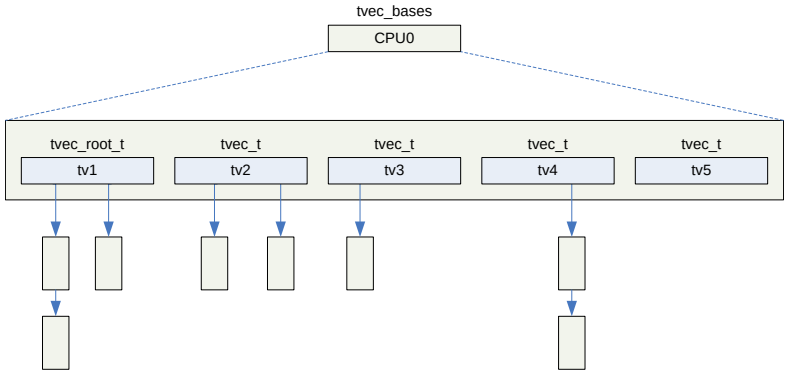

Figure 4.3: Linux Dynamic Timers.

Data structure for dynamic timers

The `struct tvec_base_t` defined in `kernel/timer.c`, contains all the data needed to manage the dynamic timers.

```
typedef struct tvec_t_base_s {
  spinlock_t lock;
  struct timer_list *running_timer;
  unsigned long timer_jiffies;
  tvec_root_t tv1;
  tvec_t tv2;
  tvec_t tv3;
  tvec_t tv4;
  tvec_t tv5;
} tvec_base_t;
```

The `tv1` field is a structure of type `tvec_root_t`, which includes an list of 256 `list_head` elements, where each entry represents a single jiffy. Hence, this list contains all dynamic timers, if any, that will expire within the next 255 ticks. The `tv2`, `tv3`, `t4`, and `tv5` fields are structures of type `tvec_t`, which includes an list of 64 `list_head` elements. These lists contain all timers that will expire within the next $2^{14}-1$, $2^{20}-1$, $2^{26}-1$, $2^{32}-1$, respectively. The properties of the different timer categories is summarized in Table 4.3.

Dynamic timer handling

Dynamic timer handling are performed by the `run_timer_softirq()` deferred function that is raised by the `TIMER_SOFTIRQ` softirq. The function essentially performs the following:

1. Acquires the `base->lock` spin lock and disables local interrupts.

Table 4.3: Cascading timer wheel list ranges

List	Start	Stop	Granularity
1	1	$2^8 - 1$	2^0
2	2^8	$2^{14} - 1$	2^8
3	2^{14}	$2^{20} - 1$	2^{14}
4	2^{20}	$2^{26} - 1$	2^{20}
5	2^{26}	$2^{32} - 1$	2^{26}

2. While `base->timer_jiffies` is smaller or equal than the value of `jiffies` , performs the following substeps:
   (a) Computes the index of the list in `base->t1` that holds the next timer to expire.
   (b) If `index` is zero, all lists `base->t1` in are empty. Then, the timers in the `base->t2` list is cascaded to the proper list of `base->t1`. If all `base->t2` lists are empty after the cascading, the timers of `base->t3` is cascaded into proper list of `base->t2`, and so on.
   (c) Increments the `base->timer_jiffies`.
   (d) For each timers in the `base->tv1.vec[index]` list, executes the corresponding timer function. When all timers are handled, continues with the next iteration of `while` loop.
3. Releases the `base->lock` spin lock and enables local interrupts.

Since softirqs are a form of bottom half, they may be executed a long time after they have been activated. Therefore, the kernel runs timer functions *equal to or greater than* their expiration times. Although the kernel guarantees to run no timers functions *prior* to their expiration, there may be a delay in running timers. For this reason, timers are not appropriate for any sort of hard real-time processing.

The cascading timer wheel provides O(1) inserting and removing times. However, when inserting a timer with an expiration time larger than capacity of the first list, the timer has to be cascaded into a lower list at least once. The cascade operation is time-consuming if a large set of timers have to be moved. It is done with interrupt disabled, and thus, the cascade operation can increase the interrupt latency?. The cascading timer wheel has excellent average performance. However, the worst case performance is unacceptable for high resolution timers.

4.4.2 Hrtimers

The hrtimer subsystem is optimized for timers with high resolution requirements. The primary purpose to separate such timers from the cascading timer wheel was to eliminating the overhead and variable latency associated with the wheel (Gleixner and Molnar, 2005).

Hrtimers uses the `xtime` variable as its time domain, and hence, is not bounded to jiffies. Figure 4.4 shows in a schematic way how hrtimers and dynamic timers are connected to the timer management. The timers are organized

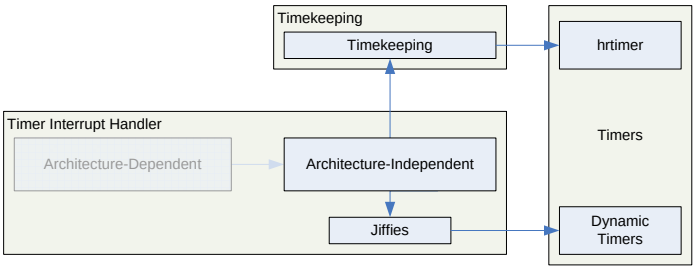

Figure 4.4: Linux Timers.

into a per-CPU red-black tree. Red-black trees provide O($log(N)$) insertion and removal, i.e. slower than the cascading timer wheel. However, since the majority of timers actually, the cascading penalty is to high for such timers. Further, red-black trees are considered to be effective enough for high-resolution timers, as they are already used in other performance critical parts of the kernel.

The hrtimer subsystem was merged into 2.6.16 Linux. The inclusion of this subsystem does not change the tick based resolution, however. Hrtimers, together with the Generic Time-of-day subsystem described in Section 4.3 create a solid foundation for support of high-resolution timers and dynamic ticks.

4.4.3 System calls for POSIX timers

The POSIX 1003.1b standard introduced software timers for user-space applications. These timers are often reffered to as POSIX *timers*. Prior to 2.6.16 Linux, the kernel implements the POSIX timers by means of dynamic timers. After the incorporation of the hrtimer subsystem, however, the POSIX timers is implemented by using hrtimers.

POSIX timers make use of POSIX *clocks*, that is, virtual time sources with predefined resolutions and properties. Hrtimers supports two types of POSIX clocks:

- `CLOCK_REALTIME`
- `CLOCK_MONOTONIC`

System calls for POSIX timers and clocks are given in Table 4.4

Table 4.4: Cascading timer wheel list ranges

System call	Description
`clock_settime()`	Sets the time of a POSIX clock.
`clock_gettime()`	Gets the time of a POSIX clock.
`clock_getres()`	Gets the resolution of a POSIX clock.
`clock_nanosleep()`	Suspends the process.
`timer_create()`	Creates a new POSIX timer based on a POSIX clock.
`timer_settime()`	Sets the time until the next expiration.
`timer_gettime()`	Gets the time until the specified timer expires.
`timer_getoverrun()`	Gets the time expiration overrun count for the specific timer.
`timer_delete()`	Destroys a POSIX timer.

Chapter 5

High Resolution Times and Dynamic Ticks

This chaper introduces high resolution timers and dynamic ticks for AVR32 Linux. In Chapter 4, the AVR32 kernel was transformed to fully support John Stult's Generic Time-of-day approach. Further, information about the hrtimer implementation was provided. The first part of this chapter covers how a new component called clock events will complement and complete the hrtimer and the Generic Time-of-day components to create a foundation for high resolution timers and dynamic ticks. The clock events component is introduced in (Gleixner and Niehaus, 2006). Then, the implementation of the clock events component on AVR32 Linux is covered in details.

5.1 Clock event management

5.1.1 Clock event source

Clock events sources are used to schedule the next event interrupt. As seen in Section 4.2.3, the current timer interrupt is acknowledged and the next one set by increasing the `COMPARE` system register by `cycles_per_jiffy`. Hence, the next event is currently defined to be periodic, with a pre-defined frequency of 250 `HZ`. Further, how much work that is done by the architecture-dependent code of the timer interrupt differs across architectures. The clock events component provides a generic solution to manage clock events sources and the time-related activities in the kernel.

Clock event sources can be registered by the architecture-dependent boot code or at module insertion time. Based on its property parameters, the clock events management decides what functionality the clock source is set to support. On a uni-processor system, this includes per-CPU functionality such as process accounting, profiling, high resolution timers, dynamic ticks and the traditional periodic tick based clock events.

The `struct clock_event_device` defined in `include/linux/clockchips.h`, contains all the element to manage the clock event sources:

```
struct clock_event_device {
  const char          *name;
  unsigned int        features;
  unsigned long       max_delta_ns;
  unsigned long       min_delta_ns;
  unsigned long       mult;
  int                 shift;
  int                 rating;
  int                 irq;
  cpumask_t           cpumask;
  int                 (*set_next_event)(unsigned long evt,
                                        struct clock_event_device *);
  void                (*set_mode)(enum clock_event_mode mode,
                                  struct clock_event_device *);
  void                (*event_handler)(struct clock_event_device *);
  void                (*broadcast)(cpumask_t mask);
  struct list_head    list;
  enum clock_event_mode mode;
  ktime_t             next_event;
};
```

In Table 5.1.1, the most important fields of the `clock_event_decive` structure are described:

Table 5.1: The fields of the `clock_event_device` structure.

Field name	Description
`features`	Bit-field which describes the features of the clock event source and its preferred usage.
`max_delta_ns`	Maximum event delta (offset into the future) which can be scheduled.
`min_delta_ns`	Minimum event delta (offset into the future) which can be scheduled.
`mult`	Multiplier for scaled math conversion from nanoseconds to clock event source units.
`shift`	Shift factor for scaled math conversion from nanoseconds to clock events units.
`set_next_event`	Architecure-dependent function which schedule the next event.
`set_mode`	Architecture-dependent function which toggles the clock event source operating mode.
`event_handler`	Architecture-independent function assigned by the framework to be called bt the low level handler of the clock event source.

5.1.2 Clock event distribution

The clock event management provides infrastructure for distributing timer-related services in the kernel. The management supports periodic and individual programmable events. The individual programmable event is used to implement high resolution timers and dynamic ticks, but only when appropriate a clock event source has been registered. Otherwise, the traditional periodic tick scheme is used. This ensures that a kernel which is configured for high resolution timers and/or dynamic ticks can run on a architecture which does not have the necessary hardware support.

5.2 Transforming AVR32 Linux to use clock events

Enable clock event devices

The generic clock event subsystem needs to be compiled into the kernel. This is done by adding the `config GENERIC_CLOCKEVENTS` to `/arch/avr32/Kconfig`:

Code Sequence 5.1: `GENERIC_CLOCKEVENTS`

```
1  config GENERIC_CLOCKEVENTS
2          bool
3          default y
4
5  config RWSEM_XCHGADD_ALGORITHM
6          bool
```

Clock event device data structure

The clock event source for AVR32 Linux is defined in `arch/avr32/kernel/time.c` as `hpt_clockevent`[1]. Its clock-specific property parameters and callback functions are set in Code Sequence 5.2.

Code Sequence 5.2: `clock_event_device`

```
7    static struct clock_event_device hpt_clockevent = {
8          .name           = "avr32",
9          .features       = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT,
10         .set_mode       = hpt_set_mode,
11         .set_next_event = hpt_next_event,
12         .shift          = 32,
13     };
```

Clock event device registration

The `hpt_clockevent` clock event source is registered in `time_init()` at boot time by calling the `hpt_enable()` function given in Code Sequence 5.3. This function determines the `mult`, `min_delta_ns`, and `max_delta_ns` fields of the

[1] `h(igh)p(recision)t(imer)_clockevent`.

`hpt_clockevent` structure by using functions decreared by the clock event framework. Finally, the clock event structure for AVR32 Linux is registered by calling `clockevents_register_device()`.

Code Sequence 5.3: `hpt_enable()`

```
14  int __init hpt_enable(void)
15  {
16    unsigned long hpt_hz;
17
18    hpt_clockevent.cpumask = cpumask_of_cpu(0);
19
20    hpt_hz = clk_get_rate(boot_cpu_data.clk);
21    hpt_clockevent.mult = div_sc(hpt_hz, NSEC_PER_SEC, 32);
22
23    hpt_clockevent.max_delta_ns =
24        clockevent_delta2ns(0x7FFFFFFF, &hpt_clockevent);
25
26    hpt_clockevent.min_delta_ns =
27        clockevent_delta2ns(0xF, &hpt_clockevent);
28
29    clockevents_register_device(&hpt_clockevent);
30
31    return 1;
32  }
```

During system boot is it not possible to use the high resolution timer functionality. The initialization of the clock event framework, the clock source framework and hrtimers itself has to be completed before the kernel can switch to high resolution timers. Before hrtimers is initialized, the system runs in periodic mode. Therefore, when calling `clockevents_register_device()` in Code Sequence 5.3, the function `hpt_clockevent->set_mode=hpt_set_mode()` is called to do necessary work to support periodic mode. The `hpt_set_mode` function given in Code Sequence 5.4, the global `cycles_per_jiffy` variable is initialized by using scaled mathematics. Then, the `COMPARE` system register is set up for the first timer interrupt by calling the `avr32_hpt_init()` function.

Code Sequence 5.4: `hpt_set_mode()`

```
33  static void hpt_set_mode(enum clock_event_mode mode,
34                           struct clock_event_device *evt)
35  {
36    unsigned long long delta;
37    unsigned long mult, shift;
38
39    switch (mode){
40      case CLOCK_EVT_MODE_PERIODIC:
41
42        shift = hpt_clockevent.shift;
43        mult = hpt_clockevent.mult;
44
```

```
45          {
46            delta = ((uint64_t) (NSEC_PER_SEC/HZ)) * hpt_clockevent.mult;
47            delta += hpt_clockevent.mult;
48            delta >>= hpt_clockevent.shift;
49
50            cycles_per_jiffy = delta;
51          }
52
53          avr32_hpt_init(avr32_hpt_read());
54
55          break;
56
57        case CLOCK_EVT_MODE_ONESHOT:
58        case CLOCK_EVT_MODE_UNUSED:
59        case CLOCK_EVT_MODE_SHUTDOWN:
60        case CLOCK_EVT_MODE_RESUME:
61          break;
62      }
63    }
```

The clock source and the clock event framework provide notification functions which informs hrtimer about the timer-related hardware on the system. Hrtimers validates the registered clock sources and clock event sources before switching to one-shot mode, i.e. high resolution timers and/or dynamic ticks.

If a timer has its expiration time before the next interrupt is scheduled to happen in one-shot mode, the event source which triggers the timer interrupt has to be reprogrammed. Thus the timer is scheduled to run at its expiration time. Hence, high resolution timers and dynamic ticks support are achieved.

On AVR32 Linux, the event source is reprogrammed with the `hpt_next_event()` function given in Code Sequence 5.5.

Code Sequence 5.5: `hpt_next_event()`

```
64  static int hpt_next_event(unsigned long delta,
65                            struct clock_event_device *evt)
66  {
67    unsigned long cnt;
68
69    cnt = sysreg_read(COUNT);
70    cnt += delta;
71    sysreg_write(COMPARE, cnt);
72
73    return ((long)(sysreg_read(COUNT) - cnt) > 0) ? -ETIME : 0;
74  }
```

Originally, the `time_init()` function initialized the `cycles_per_jiffy` variable by using the clock source and confederated the `COMPARE` register to trigger the first timer interrupt. Since this work has been moved to the `hpt_set_mode()` function, the code has to be removed from `timer_init()`. However, as already mentioned, AVR32 Linux clock event source is registered in `time_init()` by

calling `hpt_enable()`.

Code Sequence 5.6: `time_init()`

```
75  void __init time_init(void)
76  {
77    unsigned long mult, shift, count_hz;
78    int ret;
79
80    count_hz = clk_get_rate(boot_cpu_data.clk);
81    shift = 16;
82    mult = clocksource_hz2mult(count_hz, shift);
83
84    {
85        u64 tmp;
86
87        tmp = TICK_NSEC;
88        tmp <<= shift;
89        tmp += mult / 2;
90        do_div(tmp, mult);
91
92        cycles_per_jiffy = tmp;
93     }
94
95    /* This sets up the high precision timer for the first interrupt. */
96    avr32_hpt_init(avr32_hpt_read());
97
98    hpt_enable();
99
100   ret = setup_irq(0, &timer_irqaction);
101   if (ret)
```

Timer interrupt

The clock event distribution provides generic functions which allows the association of various clock event services such as jiffies tick, process accounting and profiling to a clock event source. Therefore, such services must be removed from the AVR32 Linux interrupt handler. However, the current timer interrupt must still be acknowledged and reprogrammed. The

Code Sequence 5.7: `timer_interrupt()`

```
102 static irqreturn_t timer_interrupt(int irq, void *dev_id)
103 {
104   unsigned int count;
105
106   count = avr32_hpt_read();
107   avr32_timer_ack();
108
109   write_seqlock(&xtime_lock);
110   do_timer(1);
```

```
111     write_sequnlock(&xtime_lock);
112
113     local_timer_interrupt(irq, dev_id);
114     hpt_clockevent.event_handler(&hpt_clockevent);
115
116     return IRQ_HANDLED;
117   }
```

Dynamic ticks

When the next event is more than one tick into the future, the idle tick is stopped by calling `tick_nohz_stop_sched_tick()`. This function is either called from the idle loop in `cpu_idle()` function defined in `arch/avr/kernel/process.c` as shown in Code Sequence 5.8 or from `irq_exit()` when an idle period was interrupted by an interrupt which did not cause a reschedule. The `tick_nohz_stop_sched_tick()]` function restarts the idle tick when the processor is woken up from idle.

The clock event functionality for dynamic ticks are available whether the high resolution timer is enabled or not.

Code Sequence 5.8: `cpu_idle()`

```
118
119   void cpu_idle(void)
120   {
121     while (1) {
122       tick_nohz_stop_sched_tick();
123
124       while (!need_resched())
125         cpu_relax();
126
127       tick_nohz_restart_sched_tick();
128       preempt_enable_no_resched();
129       schedule();
130       preempt_disable();
131     }
132   }
```

Configuration

High resolution timers and dynamic ticks are added to the AVR32 Linux configuration menu in `arch/avr32/Kconfig`. Figure 5.1 shows the high resolution timer support and dynamic ticks option in the menuconfig screen.

Code Sequence 5.9: `arch/avr32/Kconfig`

```
133   menu "System Type and features"
134
135   source "kernel/time/Kconfig"
136
```

```
137     config SUBARCH_AVR32B
138             bool
```

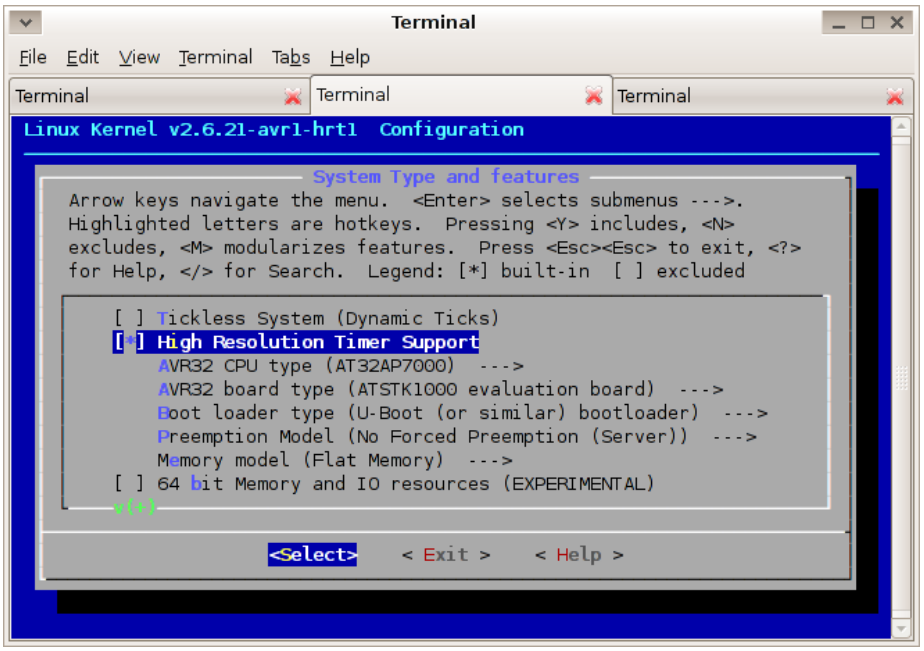

Figure 5.1: Menuconfig.

The complete source code for high resolution timers and dynamic ticks implementation is provided on the CD-ROM.

Chapter 6

Benchmarks

The capabilities of the AVR32 Linux high resolution timers and dynamic ticks are demonstrated in this chapter. The timer latencies involved when using POSIX timers are measured, and the proposed high resolution timers and dynamic ticks solution is compared with the standard AVR32 Linux kernel.

6.1 Setup

6.1.1 Hardware

All tests have been run on a AT32AP7000 which is based on a AVR32B revision 1 processor. The processor runs at 140.000 MHz.

The results of 2.6.16 Linux and 2.6.16-hrt5 are published in (Gleixner and Niehaus, 2006). These results are gathered from a Pentium III 400MHz based PC and used as reference to the results presented in this thesis.

6.1.2 Software

The 2.6.21-avr kernel referrers to a standard configured AVR32 2.6.21 Linux kernel. The 2.6.21-avr-hrt1 referrers to a standard AVR32 2.6.21 Linux kernel configured with high resolution timer and dynamic tick support. Appendix A goes into details about the kernels and theirs configuration. Both kernels mount its root file system over NFS at boot time.

The `cyclictest` utility is written by Thomas Gleixner to measure the timer latencies involved in sleep and wake operation of highly prioritized real-time threads. The utility is designed to test different user-space intervals timers and sleep functions. In particular, the following functions have be used:

- POSIX interval timers
- POSIX `clock_nanosleep()`

6.1.3 Test cases

The 2.6.21-avr and 2.6.21-avr-hrt1 Linux kernels have run through four test cases based on the `cyclictest` program. Table 6.1 shows the test cases and theirs parameters. *Loops* denotes the number of test cycles in one test. `cyclictest` stops once the number of loops have been reached. *Interval* denotes the expiration time of the interval timer and the sleep function.

Table 6.1: Test cases and their properties.

Test Case Number	POSIX timer function	Interval	Loops
1	`timer_settime()`	10000	10000
2	`timer_settime()`	500	100000
3	`clock_nanosleep()`	10000	10000
4	`clock_nanosleep()`	500	100000

The test for intervals less than the jiffy resolution have not been run on 2.6.21-avr Linux (nor 2.6.16 Linux). The test thread runs in all cases with scheduling policy `SCHED_FIFO` and priority 80.

To test the real-time behaviour of the systems, the test cases are run with and without load while measuring. Under load, the test cases are run together with the Cache Calibrator[1] tool. This program produces heavy cache pollutions and causes thus high latency time while switching between applications.

6.2 Results

Table 6.2 through 6.5 show the results of the testes described in Section 6.1. More results are presented in Appendix B. It is easily seen that the average latencies are significant for the 2.6.21-avr-hrt1 compared to 2.6.21-avr1 for all test cases. Hence, the purposed high resolution timers and dynaic ticks solution yields a lower latency than the standard AVR32 Linux kernel, and is an improvement of the this version.

Table 6.2: Test case 1 with no load.

Kernel	Minimum	Maximum	Average
2.6.21-avr	3650	5870	4690
2.6.21-avr-hrt1	103	301	133
2.6.16	21	4073	2098
2.6.16-hrt5	22	120	35

[1] `http://monetdb.cwi.nl/Calibrator/`

Table 6.3: Test case 1 load.

Kernel	Minimum	Maximum	Average
2.6.21-avr	2584	4822	3654
2.6.21-avr-hrt1	106	331	141
2.6.16	82	4271	2089
2.6.16-hrt5	31	458	53

Table 6.4: Test case 2 with no load.

Kernel	Minimum	Maximum	Average
2.6.21-avr-hrt1	74	396	106
2.6.16-hrt5	8	119	22

Table 6.5: Test case 2 with no load.

Kernel	Minimum	Maximum	Average
2.6.21-avr-hrt1	73	1888	360
2.6.16-hrt5	16	489	58

Chapter 7

Discussion

The RT-preempt patch is at the time of this writing, still under heavy development. The majority of the announcements, discussions and debates about the RT-preempt patch (and other Linux features) happen on the *Linux Kernel Mailing List*, or *lkml* for short. Most of the information concerning the RT-preempt patch exists in this mailing list and is quite factional. The patch and its logical changes have not been summarized in detail until article presented by Rostedt and Hart (2007) at the Linux Symposium, ultimo June 2007.

Prior to the RT patches created against 2.6.22 Linux and later releases, the patches were submitted in an $\sim$ 1.7MB single file as plain text. Hence, the patch contains several logical changes in one file, and not broken into chunks, with each chunk representing a logical change. The largest portion of the work with the AVR32 real-time patch has been carried out by using investigating these single files. Therefore, the preliminary work to see how the RT-preempt patch affected the AVR32 architecture might have taken longer time than necessary.

The HRT patch, which is independently maintained, is made out of chunks that represent logical changes. This made it easier to understand the patch, and in turn, pinpoint the changes that is relevant for architecture-dependent code. Further, the preliminary work before porting the patch to the AVR32 architecture was more effective than in the RT-preempt patch case, because of papers such as (Gleixner and Molnar, 2005) and (Gleixner and Niehaus, 2006).

The patches presented in this thesis is up to date as of AVR32 Linux kernel version 2.6.21. The kernel is a moving target, and late in the process of this thesis, Atmel released 2.6.22 AVR32 Linux which includes a real time clock driver. The inclusion of the real time clock into the AVR32 timer management makes the kernel to reject the AVR32 high resolution timers and dynamic ticks patch. Nonetheless, the internals of patch are mature, and can without much difficulty be adopted by higher release numbers than 2.6.21 AVR32 Linux.

Chapter 8

Conclusion

The main objective of this thesis was to improve the real-time capabilities of AVR32 Linux. This was done by implementing features from the RT-preempt patch to AVR32 Linux. The patch includes three different features; fully preemptible kernel, high resolution timers and dynamic ticks.

The fully preemptible kernel investigation identified the need of a more advanced interrupt controller for the AVR32 architecture. With the existing interrupt controller, the current interrupt request cannot be masked individually. When a hardware interrupt which is converted into a kernel thread triggers, the interrupt is not and, therefore, triggers again. Threaded hard interrupts are a prerequisite to implement a fully preemptible AVR32 kernel.

The timer specific code in AVR32 Linux is converted to fully support John Stulz's generic time-of-day and the clock events components. The clock event source operates either in periodic or one-shot mode. The configuration of AVR32 Linux is extended to include high resolution timers and dynamic ticks. When high resolution timers is enabled, the clock event resolution is no longer bounded to jiffies, but to the underlying hardware itself.

The benchmark results for the high resolution timers and dynamic ticks implementaion for AVR32 show significant lower latencies for POSIX timers.

The real-time capabilities of the AVR32 Linux kernel has been imporved. This thesis describes only the first steps towards a fully preemptible AVR32 Linux kernel. However, the implementation considering high resolution timers and dymanic ticks is made mature to be incorporated. Now, applications running AVR32 Linux can utilize time driven and event driven events with microsecond accuracy

Chapter 9

Future Work

As always, heaps of future work remains. The most important work follows in preferred order:

- The 2.6.22 AVR32 Linux version incorporates a Real Time Clock driver. The clock and clock event sources in the AVR32 Linux timer related code should utilize this driver.
- More code inspection are required to implement a fully preemptible kernel implementation for AVR32 Linux. The current implementation is preliminary.
- The AVR32 architecture should provide a more advanced interrupt controller. The controller must provide the ability to mask individual interrupt. This is required to implement threaded hard interrupts. This is a comprehensive work that will require considerable effort.

Bibliography

Bovet, D. P. and M. Cesati (2005). *Understanding the Linux Kernel* (3rd ed.). O'Reilly.

Burns, A. and A. Wellings (2001). *Real-Time Systems and Programming Languages:* (3rd ed.). Addison Wesley.

Love, R. (2005). *Linux Kernel Development*. Novell.

Stallings, W. (2005). *Operating Systems: Internals and Design Principles* (5. ed.). Prentice Hall.

References

Anzinger, G. (2003). Vst (tick elimination) is now available. http://lwn.net/Articles/53428/.

Anzinger, G. and N. Gamble (2000). Design of a fully preemptable linux kernel. http://linuxdevices.com/articles/AT4185744181.html.

Atmel (2006, October). *AT32AP7000 Preliminary Complete*.

Dietrich, S.-T. and D. Walker (2005). The evolution of real-time linux.

Gamble, N. (2001). Linux kernel preemption project. http://sourceforge.net/projects/kpreempt.

Gleixner, T. and I. Molnar (2005). Patch: ktimers subsystem. http://lwn.net/Articles/152363/.

Gleixner, T. and D. Niehaus (2006). Hrtimers and beyond: Transforming the linux time subsystems.

Heursch, A. C., D. Grambow, D. Roedel, and H. Rzehak (2004). Time-critical tasks in linux 2.6 concepts to increase the preemptability of the linux kernel. http://inf3-www.informatik.unibw-muenchen.de/research/linux/hannover/automation_conf04.pdf.

Kalivas, C. (2005). [patch] i386 no-idle-hz aka dynamic-ticks 3. http://lkml.org/lkml/2005/8/3/14.

Love, R. (2005). *Linux Kernel Development*. Novell.

Marchesotti, M., R. Podesta, and M. Migliardi (2006). A measurement-based analysis of the responsiveness of the linux kernel. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, Washington, DC, USA, pp. 397–408. IEEE Computer Society.

McKenney, P. (2005a, August). A realtime preemption overview. http://lwn.net/Articles/146861/.

McKenney, P. E. (2005b). Patch: Attempted summary of "rt patch acceptance" thread, take 2.

Morton, A. (2001). Linux scheduling latency. `http://www.zipworld.com.au/~akpm/linux/schedlat.html`.

Mosnier, A. (2005, July). Embedded/real-time linux survey. `www.4real.se/pdf/linux_ert_survey_4real.pdf`.

Niehaus, D., W. Dinkel, M. Frisbie, and J. Wolterdorf (2005). Kurt-linux user manual. `http://www.ittc.ku.edu/kurt/documents-noframes.html`.

Niehaus, D., R. Hill, B. Srinivasan, and S. Pather (1998). Temporal resolution and real-time extensions to linux. Technical report.

Rostedt, S. and D. V. Hart (2007). Internals of the rt patch.

Stallings, W. (2005). *Operating Systems: Internals and Design Principles* (5. ed.). Prentice Hall.

Stultz, J., N. Aravamudan, and D. Hart (2005). We are not getting any younger: A new approach to time and timers. In *Proceedings of the Linux Symposium*, Volume 1, pp. 219–232.

Yang, J., Y. Chen, H. Wang, and B. Wang (2005). A linux kernel with fixed interrupt latency for embedded real-time system. pp. 127–134.

Appendix A

AVR32 Linux kernel

A.1 2.6.21-avr

```
$ wget
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.tar.bz2
$ tar xfj linux-2.6.21.tar.bz2
$ cd linux-2.6.21
$ make ARCH=avr32 CROSS\_COMPILE=avr32-linux- defconfig
$ make ARCH=avr32 CROSS\_COMPILE=avr32-linux-
```

A.2 2.6.21-avr-hrt-dynticks

```
$ wget
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.tar.bz2
$ tar xfj linux-2.6.21.tar.bz2
$ tar hrt-dynticks-patches.tar.bz2
$ cd linux-2.6.21
$ for patch in ‘cat ../patches/series‘; do
>patch -p1 < ../patches/$patch;
>done
$ make ARCH=avr32 CROSS\_COMPILE=avr32-linux- defconfig
$ make ARCH=avr32 CROSS\_COMPILE=avr32-linux-
```

A.3 2.6.21-avr-rt

```
$ wget
http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.21.tar.bz2
$ tar xfj linux-2.6.21.tar.bz2
$ tar rt-patches.tar.bz2
$ wget
http://people.redhat.com/mingo/realtime-preempt/older/patch-2.6.21-rt8
```

```
$ cd linux-2.6.21
$ patch -p1 < ../patch-2.6.21-rt8
$ for patch in ‘cat ../patches/series‘; do
>patch -p1 < ../patches/$patch;
>done
$ make ARCH=avr32 CROSS\_COMPILE=avr32-linux- defconfig
$ make ARCH=avr32 CROSS\_COMPILE=avr32-linux-
```

Appendix B

Benchmark results

Table B.1: Test case 3 with no load.

Kernel	Minimum	Maximum	Average
2.6.21-avr	690	2717	1694
2.6.21-avr-hrt1	30	129	64
2.6.16	24	4043	1989
2.6.16-hrt5	12	94	20

Table B.2: Test case 3 with load.

Kernel	Minimum	Maximum	Average
2.6.21-avr	2919	5065	3961
2.6.21-avr-hrt1	24	256	57
2.6.16	55	4280	2198
2.6.16-hrt5	11	458	55

Table B.3: Test case 4 with no load.

Kernel	Minimum	Maximum	Average
2.6.21-avr-hrt1	19	292	31
2.6.16-hrt5	5	108	24

Table B.4: Test case 4 with no load.

Kernel	**Minimum**	**Maximum**	**Average**
2.6.21-avr-hrt1	20	268	35
2.6.16-hrt5	9	684	56

Appendix C

CD-ROM Content

- ./avr32-hres-dynticks
  - avr32-prepare-for-dyntick.patch
  - avr32-use-gtod-persistent-clock-support.patch
  - clockevents-driver-for-avr32.patch
  - gtod-clocksource-avr32.patch
  - gtod-clocksource-avr32.patch.patch
  - hrt-dynticks-patches.tar.bz2
  - hrtimer-hres-avr32.patch
  - remove-useless-code-in-time-c.patch
  - version.patch
- ./avr32-rt-preempt
  - avr32-realtime-preempt.patch
  - rt-patches.tar.bz2
- ./benchmark
  - .config
  - /cyclictest
    - Makefile
    - cyclictest.c