CHEMNITZ UNIVERSITY OF TECHNOLOGY

Faculty of Computer Science

Chair for Real-Time Systems

Diploma Thesis

Lightweight RTAI for IA-32

Michael Heimpold

Chemnitz, June 21, 2007

Supervisor: Dr.-Ing. Robert Baumgartl

Heimpold, Michael
Lightweight RTAI for IA-32
Diploma Thesis, Faculty of Computer Science, Chair for Real-Time Systems
Chemnitz University of Technology, June 21, 2007

Abstract

Using the IA-32 platform for time critical tasks allows highest computing performance at reasonable hardware costs. However, systems with restricted resources were limited to proprietary real-time operating systems. As many of available open source real-time distributions base on the Linux kernel the relatively high requirements of Linux also apply to these systems. In this work a porting of the open source RTAI distribution to a bare machine is presented which eliminates Linux' core subsystems. Thus the image size and the memory footprint are optimized for small systems. Henceforth, a free available, efficient and well supported real-time operating system can be used on IA-32 with restricted resources.

Acknowledgements

I would like to use the chance at this point to thank everybody who supported me during preparation of this work. At first I thank my parents who made my studies possible at all. Furthermore, I want to thank my family, especially my wife, and friends for the patience they had with me in the last months and for their moral support. A special thanks goes to Jörg Rödel for proofreading this work. Last, I want to thank my supervisor Dr.-Ing. Robert Baumgartl for supporting this work and for his understanding of the problems I had with it.

Aufgabenstellung

Das Realtime Application Interface (RTAI) ist ein Weg, die Echtzeitfähigkeit von Linux zu erreichen. Nachteilig ist jedoch der verhältnismäßig große Speicherplatzbedarf eines RTAI-Systems sowie die fehlende Separierung zwischen Echtzeit-Applikationen und Linux-Kernel. Ziel der Arbeit ist es daher, für die PC-Architektur IA-32 das Linux-Subsystem aus einem RTAI-System zu eliminieren.

Für eine DSP-Architektur ist dies bereits erfolgreich durchgeführt worden (DA Jens Kretzschmar), die Intel-Architektur bietet jedoch einige Herausforderungen, da vielfältige Beziehungen zwischen RTAI und dem Linux-Subsystem bestehen. Unter anderem muss der Bootvorgang und der Start des Systems neu konzipiert werden, der traditionelle insmod/rmmod-Mechanismus steht nicht mehr zur Verfügung.

Das resultierende System ist ein sehr kleiner, übersichtlicher und effizienter Echtzeitkernel, der die RTAI-API bietet und für Systeme mit stark eingeschränkten Ressourcen prädestiniert ist.

Bearbeiter:	Michael Heimpold geb. am 15.05.1981 in Werdau
Betreuender Hochschullehrer:	Dr.-Ing. Robert Baumgartl Juniorprofessur Echtzeit-Systeme
Ausgabedatum:	21.09.2006
Abgabedatum:	21.06.2007
Tag der Abgabe:	

Contents

List of Figures

Chapter 1

Introduction

In the recent past there was an increasing demand for applications that run with strict timing constraints. The fields of such applications spread over processing audio and/or video data i. e. for telecommunications to dealing with complex calculations in motor control units. All this applications have in common that they need to fulfill their work in a time bounded fashion to avoid service degeneration or damage of the system they belong to.

Usually the development of such applications does not start from scratch for the desired target environment (that means the , the system board etc.) but an existing operating system is used which provides a more or less extensive set of functionality. Nowadays there is a wide range of various commercial and/or proprietary real-time operation systems () and open-source variants. Using the later ones does not only result in cost saving but also in having a "white box" system which guarantees full control over it. This is true also for the progress of the development as needed modifications could be integrated upstream. So using open source additionally provides investment security.

At the time of writing Linux as the most commonly known open source operating system is still a general-purpose operating system () and does not yet deserve being called RTOS. However, using Linux as a starting point for a real-time environment does make sense because of the code maturity and support for the most common hardware.

Basically there exist two strategies to enhance the Linux kernel with real-time capabilities:

- Modifying the Linux kernel itself to provide the wanted determinism and predictability. For historical reasons this approach was developed in parallel to the kernel as some of the original design goals were oppositional to what would be needed by a real-time operating system. But today

there is an ongoing development to integrate native real-time support into the Linux as shown in the following.

- Using an additional real-time kernel which encloses/bases the standard Linux kernel. While this concept seems to be very similar of using a micro or nano kernel driven approach it differs in so far that the added real-time part could be and usually is interwoven closely with the standard kernel. Also, in a micro or nano kernel based system only the core components run in *kernel mode* whereas the non-essential services, in this domain then called *servers*, resist in user-space.

When the development on Linux started it was solely designed to run on Intel's 386 or 486 CPUs. And it tried to use all available features of the CPU, particularly paging and the 32-bit protected mode. The reasons for this becomes clear when remembering the historic situation of these days. Linux was intended to be a replacement for the Minix system which was mostly used as an educational operating system. Linux itself, however, was (and still is) strictly user orientated. That means that during the evolution in the following time only features get integrated which were actively used and needed. The two main fields of application were desktop and server systems for a long time. Therefore and as hardware, particularly , becomes more and more available at reasonable costs, other goals dominated the development and compromises on using resources miserly or economically were needed. By the example of RAM, the result is that a minimal Linux kernel requires at least 2 MiB of installed RAM in the target system.

is an enhancement for the Linux kernel to provided real-time support. The minimal requirements of the resulting system in respect to e. g. the memory size do not change. On the contrary, the extension tries to manage with the resources provided by the host system. Therefore to reduce the total requirements, the Linux side of the requirements has to be trimmed or the Linux subsystem has to be eliminated at all.

In this work such a trimmed version of a RTAI enabled kernel is presented forming a lightweight implementation *LRTAI* of the RTAI . Thereby only the traditional Intel architecture IA-32 is focused, similar work for e. g. a architecture was performed by [1] and [2]. Also a lower bound of the required memory for the newly developed system will be discussed and estimated to simplify the decision whether the use of for a real-time system should be considered or if a traditional Linux/RTAI system would be more appropriate.

Chapter 2

State of the art

2.1 Linux kernel's native real-time support

2.1.1 Historical non-preemptibility

As already mentioned the Linux kernel was originally not designed for real-time awareness. One indication for this is that all interrupts are treated as equal. For a possible prioritization the kernel beliefs in the hardware to care for this. Also it was not a preemptive kernel at all. It was assumed that, when entering the kernel from a trap or a system call, the current user-space process will not change unexpectedly.

In kernel version 2.0 a global kernel lock was inserted to ease the introduction of symmetric multiprocessing () support. This was needed to protect some critical sections and to serialize the access to important data structures. If a process wanted to enter the kernel, it had to acquire this so called "Big Kernel Lock" (). The kernel itself could therefore be seen as a big critical section which was not preemptible. However, it was possible that a process calls the scheduler voluntary.

A kernel with such properties is not suitable for real-time systems. If a hardware interrupt occurs which e. g. could be triggered from an active device and a process is currently running in kernel mode, the interrupt handler is delayed until the current process finishes its work. One sees that the interrupt response time is subject to large fluctuations which is not usable in real-time operating systems.

During the further kernel development on version 2.2 and 2.4 the big kernel lock was more and more replaced with finer-grained localized locks of particular critical sections. This improved the response times slightly but the kernel

could still not being regarded as real-time aware.

The lack of real-time support has some simple reasons: Linus Torvalds, the original author and maintainer of the Linux kernel, defied adding real-time relevant patches for a long time following the idea of high stability and throughput of traditional UNIX kernels.

But the current situation was not satisfying for many Linux users. So two independent works started to resolve the problem. The results of these works were some patches which could be applied to the kernel's sources tree. As they were not included in the kernel's upstream both were maintained outside the kernel tree for a long time. Both patches which actually are patch sets are shortly presented in the following two subsections.

2.1.2 The Preemption Patches

The embedded Linux vendor MontaVista Software, Inc [5] was one of the activists. After publishing in the year 2000, their work was quickly picked up by the Linux community and consists of the actual preemption patch and a real-time scheduler. Usually this work is referred to as the "preemption patches" which are maintained by Robert Love in the meantime and can be obtained from his kernel.org space [6].

The work tries to minimize the time lag between an incoming event e. g. an interrupt and the next invocation of the scheduler. To achieve this some locking primitives are modified together with some slight adaptation of the interrupt handlers. In the result the scheduler is called more often and can faster react to a pending rescheduling request. In [7] some measurements are done which confirm the improved scheduler latency.

Even though the patches are relatively small and good to maintain, the inclusion into the Linux kernel's upstream was delayed until version 2.5.4 in 2002.

2.1.3 The Low-Latency Patches

The second approach which was presented by Ingo Molnár in the year 2001 tries to attack the response time problem from another direction. He examined the existing source code looking for long time running uninterruptible code blocks. The goal was to break this blocks into smaller chunks by explicit calls to the scheduler after some amount of work is done. The original task will be resumed at the interrupted position when the scheduler comes back later.

```
1  void prune_dcache(int count)
2  {
3
4
5    spin_lock(&dcache_lock);
6    for (;;) {
7
8
9
10
11
12
13
14
15
16     /* prolonged work */
17   }
18   spin_unlock(&dcache_lock);
19 }
```

```
void prune_dcache(int count)
{
  DEFINE_RESCHED_COUNT;
redo:
  spin_lock(&dcache_lock);
  for (;;) {
    if (TEST_RESCHED_COUNT(100)) {
      RESET_RESCHED_COUNT();
      if (conditional_schedule_needed())
      {
        spin_unlock(&dcache_lock);
        unconditional_schedule();
        goto redo;
      }
    }
    /* prolonged work */
  }
  spin_unlock(&dcache_lock);
}
```

Figure 2.1: Introduction of an explicit preemption point (in *fs/dcache.c*).

The most popular example of such an explicit preemption point is shown in Figure 2.1. One may notice that the test for the rescheduling condition is enveloped by a test of the iteration progress of the loop. This is necessary to avoid so called "live locks" which may occur on heavy system load. Then the rescheduling condition would be almost true resulting in a loop which is solely interrupted but does not any work. With the test of the iteration count at least a minimal progress of the intended work is assured.

Compared to the former one, this patch seems to achieve better results as shown in [7], too. Even though, the concept of the low-latency patch is simple, the main work has to be done during implementation. Firstly, potentially long term running code blocks have to be identified and then it has to be examined if an explicit preemption point could be safely inserted.

The official inclusion of this patch set was delayed for a long time, too. Finally they got integrated into kernel version 2.6.11.

2.1.4 Final approach: full-preemptibility

The two patch sets which were presented above improve both the kernel response times when applied independently. It was also possible to apply both patches in common which is actually true for kernel versions greater than 2.6.11.

However, the kernel can not yet be called a RTOS kernel as some important RTOS specific features (e. g. priority inheritance) are still missing. Also the kernel is still not fully-preemptible as interrupt service routines were not yet preemptible.

It was Ingo Molnár again who presented a solution. This new patch set is called "preempt-rt" standing for "real-time preemption patches". A subset of this patch set was already integrated into kernel version 2.6.18. However, it is still under development.

The final goal of this work is a full-preemptible kernel and therefore a fully deterministic scheduling behavior. This shows that major attributes and features of a RTOS are slightly integrated into the Linux kernel's upstream. So it could be expected that the vanilla kernels[1] will play an increasing role in the real-time market.

2.2 The Real Time Application Interface

2.2.1 Principles

The Real-Time Application Interface (RTAI) is not a standalone RTOS. It is rather one of the efforts to enhance the Linux kernel with hard real-time capabilities.

RTAI was developed in Italy in the Dipartimento di Ingegneria Aerospaziale at Politecnico di Milano. The project evolved from former real-time experiences of the group around professor Paolo Mantegazza. Their previous work was called "PCDOS-DIAPM-RTOS" standing for a RTOS running in 16-bit real mode of Intel compatible standard PCs. For the transition to 32-bit protected mode the group evaluated several approaches and systems which should provide a new basis for the 32-bit code base.

Approximately at this time a patch for the Linux kernel was presented which added elementary support for real-time tasks. This RTLinux patch was developed at the New Mexico Institute of Mining and Technology by Victor Yodaiken and Michael Barabanov [9].

Using this patch, Mantegazza and his team discovered that an own implementation was needed due to bad performance. As a reason for this bad performance the so called "one shot mode" was identified. In middle of April 1999

[1] "Vanilla" kernels are the official Linux kernels released from [8].

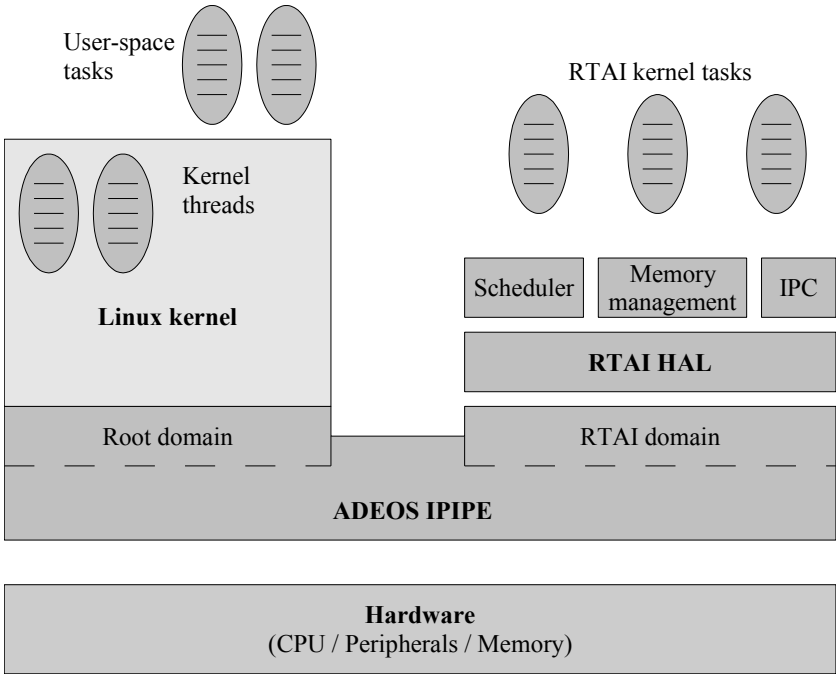

Figure 2.2: Stacked layers in a Linux/RTAI system.

the first version under the acronym *RTAI* was finally released. In the mean while the project is not solely a research work but a distributed open-source community project and a widely used real-time distribution. More of the history of the project can be found in [10] and an overview of the various Linux based real-time distributions and their mutual influences is given in [11].

As already mentioned the RTAI concept bases on the "kernel dualism" approach. For this a hardware abstraction layer () is used. This layer is inserted between the real hardware and the Linux kernel and covers in substance only the interrupt system. Figure 2.2 tries to illustrate the stacked layers in a Linux/RTAI system. The idea behind such an abstraction layer is that determinism of a scheduling algorithm can only be achieved when full control over the interrupt system is available. This is explained in the following subsection.

Furthermore RTAI comes with a separate scheduling algorithm which enables the resulting system to run tasks which are completely independent from the Linux kernel. The RTOS base is completed by a real-time aware memory

management and some inter-process communication () tools. These concepts are also introduced in the following.

2.2.2 The interrupt pipeline

The interrupt pipeline, in short , is the core of the two-kernel strategy. It is depended from the used definition what is called *kernel*. From my point of view the resulting combination does not form to independently executable instances, apart from that the original Linux kernel will do without the IPIPE system. However, the (Adaptive Domain Environment for Operating Systems) project from which the IPIPE system originate refers to the "ADEOS nanokernel beneath the Linux kernel" [12]. In fact, it neither manages other resources than the interrupts nor does it provide any inter-domain communication facilities.

The basic idea of the ADEOS interrupt management consists of virtualizing all available interrupt sources and introducing some additional "virtual" interrupts. This means that all hardware interrupts are caught by the ADEOS layer and translated into interrupt events. Therefore ADEOS groups the hardware interrupt access into so called *domains* of different and fixed priority.

The code which is actually inserted in the Linux code base modifies Linux' original interrupt management to handle these new interrupts events. The Linux system is therefore migrated into the *root domain* which is able to open up further ADEOS domains. This is used by RTAI's hardware adaption layer module *rtai_hal* which spans a second domain during initialization. RTAI domain's priority is higher than that of the root domain.

The priority of the domains are used when dispatching a hardware interrupt source. As every domain is allowed to register an interrupt handler within the ADEOS system for every available interrupt source it is not guaranteed that it will be called at all. When an interrupt event occurs the domain list is processed in order of decreasing priority which means that a registered interrupt handler of a high-priority domain is called first. Depended from its return value and the corresponding domains configuration it is decided whether the "lower" interrupt handlers are called or if the event is not further dispatched. Hence the name interrupt pipeline.

With RTAI's higher priority it has full control over all available interrupt resources and can decide which interrupts are passed to Linux. It is also possible to stop the pipeline. Then no interrupts are passed at all to lower domains. The

handlers of the stalled interrupts are called soonest when the current domain restarts the pipeline.

As already remarked both the original Linux domain and RTAI's domain run in the same security level of the CPU i. e. ring level zero which is called "kernel mode". In this mode all instructions of the CPU are available and the code executed in this mode has full access to the hardware. That means that Linux kernel code could easily circumvent the restrictions imposed upon to it by directly interfering in the interrupt handling e. g. by clearing or setting the global interrupt flag of the CPU. The cleaner and therefore compatible way of dealing with hardware interrupts is to use the provide macros in the source code. These are subjects to be changed by the ADEOS patch and will be replaced with safer code which maps the desired functionality to the hardware abstraction layer. So the Linux sources well be turned into a well behaving teammate in the system.

2.2.3 Scheduling

When introducing new real-time tasks there must be a possibility to manage these new tasks. The Linux kernel would already provides some data structures, matured task management code and a scheduler which operates on these data structures. However, the scheduler is/was not suitable for real-time tasks. Additionally, it is a very complex system so any modification would have been hard to maintain. Thus RTAI implemented the needed data structures and code parts itself to being not restricted to the limited capabilities of the Linux scheduler.

Nowadays in recent RTAI distributions, the scheduler is available in two "incarnations": the first one is built as a modules called *rtai_sched*, the second one can be found in *rtai_lxrt*. The implementation of both modules is almost the same, only the type of objects which can be handled is different. The later scheduler is provided for user-space based real-time applications. These application are mainly used while developing and ease this process as user-space debugging can be used. The first module is only capable of scheduling RTAI's own lightweight tasks which use the kernel mode only. This module will be used in LRTAI.

The scheduler's implementation is almost platform-independent, only a few parts are done in assembler e. g. the task switch. The both mentioned incar-

nations arise when toggling a flag at compile time. The standard build system compiles both modules by default. The generated kernel modules contains additionally almost all functions of the RTAI API. Only the memory management and inter-process communication facilities are provided by their own modules as describe in the following subsections.

2.2.4 Memory management

To fulfill the hard timing constraints of a real-time system, RTAI comes also with its own memory management module. This is necessary as Linux, being a general purpose operating system for desktop and server system, implements oppositional design goals, at least partly. For example Linux tries hard to fulfill a memory request for an application. Therefore it can be configured to swap memory to disks. Another option was to reduce some buffers which have been grown when free memory had been available. In the result the process requesting memory could be sent to sleep until the allocation could be successfully terminated.

Such a behavior is not suitable for hard real-time systems. The request should be processed in a time bounded manner so that the timing constraints are not broken. For this RTAI implements its own memory allocator which bases on the algorithm presented in [13].

However, the allocator could not simply distribute memory. As the RTAI subsystem is usually loaded after the Linux system is already up and running, the Linux memory management has already grabbed all available memory. Therefore RTAI must firstly request some chunks of free memory via Linux' API. This is done for example when the *rtai_malloc* module is loaded. It request a configurable amount of memory which forms a global heap and can be used later in a time bounded fashion as described.

To avoid any further negative influence of Linux' management, the allocated pages are usually locked which means that they are not considered to being swapped out to disk. This applies at least for the user-space parts of a real-time application. The kernel mode real-time tasks uses either the *kmalloc* and/or *vmalloc* kernel functions to preacquire some memory which is later converted into a real-time aware memory heap or they use the memory API of RTAI directly. In the second case the global heap is used actually. But in both cases Linux kernel memory is used, so there is no need to lock these pages to prevent Linux from interfering as kernel memory is never swapped to disk. However,

the pages are still marked as reserved.

In the current RTAI distribution, the *rtai_malloc* module is of minor importance only. The reason is that a second implementation for an global heap is available in the *rtai_shm* module. This module was originally intended to only provide shared memory capabilities but by sharing the memory segments with unique identifier the behaviour of a global heap could easily emulated. For new applications the documentation suggests to solely use the methods provided by the shared memory module. This is because of the symmetrical API which became available for user-space tasks during the transition to support hard real-time aware user-space applications.

However, as the shared memory module uses the kernels page (re)mapping infrastructure, the functions "are better assumed as not affording real time performance" as stated in [14]. Therefore for LRTAI the *rtai_malloc* will be preferred for providing memory services.

2.2.5 Additional features

The already presented modules only provides elementary functionality for real-time applications. Beside the functions which make the virtualized interrupt system available, the RTAI distribution consists additionally of a few further modules which expand the described base system by some inter-process communication facilities. The modules which are be used can be chosen at compile time.

In detail this modules are (non-completive list):

- **rtai_bits**
  With this module it is possible to use compound synchronization facilities which base on logical AND/OR operations executed on 32-bit variables. This functionality is often referred as flag or event handling. The difference to semaphores is that signaling depends not only on a single value but on a bit combination.

- **rtai_mbx**
  In this module some mailbox related functions are implemented. This could be used to pass messages of arbitrarily size between two processes.

- **rtai_msg**
  The module implements real-time aware message handling functions.

- **rtai_sem**
  This module contains RTAI's semaphore and spinlock implementations. The RTAI user manual [14] explains:

  > A semaphore can be used for communication and synchronization among real-time tasks. [...] A spinlock is an active wait synchronization mechanism useful for multiprocessors very short synchronization, when it is more efficient to wait at a meeting point instead of being suspended and then reactivated, as by using semaphores, to acquire ownership of any object.

2.3 The Linux kernel build system

The Linux kernel is distributed as a so called "tarball" which can be downloaded from The Linux Kernel Archives [8]. After unpacking, the build system of the kernel will assist to turn the source packages into something useful i. e. to compile the kernel.

Therefore the build system has to address several points. First, it is the "face" of the kernel sources which is seen by a user. As an user a person is considered when it solely wants to compile a recent kernel for its Linux distribution but not actually spend a look into the kernel sources themselves. Such a person needs an interface to choose between the many subsystem modules, drivers and other features. This interface should be reasonable easy to use i. e. it should guide the user and prevent incorrect use.

The second target group are the developers who actively work with the kernel sources e. g. by maintaining a subsystem or device driver. For these people the build system should provide rich functionality to ease their work.

The process of building a kernel can be divided into two parts. The first part consists of the configuration step. As the kernel comes with lots of drivers and as the subsystems are highly modularized, it can be freely chosen which of the features should go into the static kernel image, which ones are built as kernel modules for dynamical loading at runtime and which features are not needed at all. After this first step, the second one is executed which finally compiles all source files and links the object files together.

2.3.1 Configuration

The process of kernel configuration is usually initiated by running one of the "make {,menu,x,g,old,rand,def,allmod,allyes,allno}config" commands where the first four targets will give a different (graphical) user interface and the later ones differently preset the configurable items.

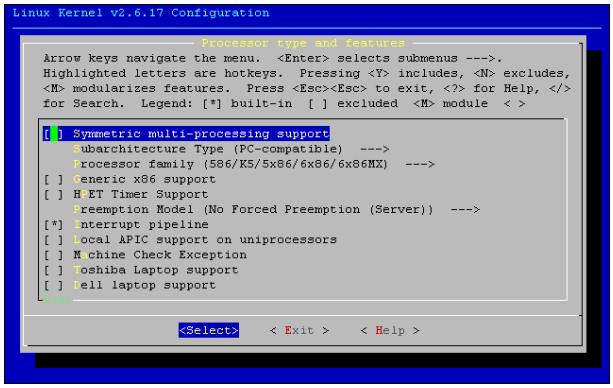

Figure 2.3: A "make menuconfig" provides a dialog based kernel configuration for choosing between various features and/or for tuning parameters.

Each configuration item has a data type which defines which values the item can hold. The most important types are "int", "bool" and "tristate" where the last mentioned will accept the values "n", "y" or "m". This is usually used to decide whether an item should be included statically in the kernel image ("y") or if it should be built as a loadable module ("m"). A value of "n" indicates that the feature behind the item is not used.

The kernel build system knows which configuration items are available by inspecting the various *Kconfig* files spread over the source tree. To keep the overview, the items are grouped into menus at different levels. Since some modules (this is true for the statically included ones too) require the presence of services provided by other kernel parts, a dependency information can be associated with each item. The build system evaluates these dependencies and adopts the configuration dynamically if changes are needed.

The result of the kernel configuration is finally saved in *$(KERNELOUTPUT)/.config*. In the further build process this file is transformed in a header file which can be included in the C source files and it is split into multiple small include files which serves for easier dependency tracking.

2.3.2 Makefiles

The kernel build system relies heavily on 's *make* tool which can usually found on every Linux installation. This tool assists larger projects by providing build dependencies so that it can determine automatically which pieces of the sources need to be recompiled if something changed. The needed information is given in so called *makefiles*. It is also possible to define macros and variables which are evaluated at runtime.

To support kernel developers and to keep the complexity of maintenance as low as possible, the kernel build system uses such makefiles to implement a powerful framework. Thus it can be achieved that when adding new features only a few lines must be added to the corresponding makefile.

Also the kernel configuration is included and can be referred. So it is possible to selectively compile the wanted features and to not spend time on compiling stuff which will not be used. This speeds up the kernel build time.

Important to know is that the build system uses the makefiles in an inconsistent fashion. As the kernel sources are split over multiple directories there are potential problems with resolving build dependencies, see [15] for details. A solution for this problem is to generate a big virtual makefile which includes the makefiles of lower level rather than to descend into lower level with new makefile processes. Linux' build system mostly uses the virtual makefile approach, however, some targets are defined explicitly and launch a new process.

These internals are hidden from a user who initiates the whole process by a simple/single *make zImage* or *make bzImage* call.

2.4 RTAI's build system

RTAI's original build system is tricky. It successfully combines the most powerful features of two worlds, namely the intuitive user-interface and dependency system of the kernel build system and the powerful host tool chain which is provided by GNU's *autoconf/automake/libtool* system. This complexity was necessary to fulfill the multiple requirements e. g. a based configuration system, cross compilation support and a reliable host feature detection. Furthermore, RTAI supports the C++ programming language for its applications, whereas the original kernel code is restricted to be written in C and assembler.

Building the RTAI distribution is very similar to building a kernel. First, the

vanilla kernel sources has to be patched to include the ADEOS IPIPE. This patch is included in the original RTAI tarball and has to be applied before configuring the kernel. The modified and added parts integrates smoothly in the remaining system. No special care must be token by users so that thereafter the kernel could be compiled as usual i. e. first the configuration step, followed by the actual build.

When the kernel build finished, for RTAI a similar configuration step can be run. In this stage a configuration *.rtai_config* is generated which is RTAI's counterpart of Linux' *.config*. But RTAI's build system passes this file to autoconf which finally creates *rtai_config.h*, a header which contains the whole configuration and is therefore included in nearly every source file. Also the makefiles for the RTAI modules are generated during this stage.

2.5 Patches

Among many other things, the word "patch" has the sense of being a form of source code modification. Usually intended to correct small error in software systems, the patch utility is somewhat "misused" in open-source communities. Here patches are also used to distribute significant changes in the software as already shown in 2.1.2 and 2.1.3.

A patch file can have various layouts. The most useful layout is where the actual modifications are enveloped by some lines of the original context. Another layout carries only the line number as a hint, beside the information what should be replaced and whereby.

Using a patch instead of distributing modified copies of the original work has some advantages. First, though the patch may carry significant changes, the modifications to the source code are usually small. So resources are saved during transmission and while archiving. Secondly, patches tend to be more easily to maintain. For example when reducing a function to a stub by inserting preprocessor directives, only the header and the tail of the implementation has to be modified, the actual implementation does not matter. A patch which consists only of the modification and a few context lines would also apply to a function where the inner implementation has changed completely. Thus a patch can often be used with recent software releases, too. An adaption is only necessary where changes directly conflicts or when the original changed significantly.

2.6 Other related work

As already mentioned, RTAI was only one approach of enhancing Linux with real-time capabilities. The most competitive solution was RTLinux which was also mentioned above. Originally developed at the university of Socorro, New Mexico (USA), the development was soon migrated to a newly launched company. As a result, the code was available as a proprietary commercial product and a free open-source variant. The later is also known as RTLinux Free respectively as RTLinux and is a community supported project.

It was already mentioned that RTAI was influenced by the work which is also the base for project. So it does not surprise that the same kernel dualism approach is used here, too. However, both projects were developed independently[2] in the mean while which resulted in drifting apart. This can be verified by the different application programming interfaces of both projects.

In year 2003 a work similar to this thesis was presented for the RTLinux GPL project. It was published by the group around Alfons Crespo at the Universidad Politcnica de Valencia, Spain [16]. The work presented a porting of RTLinux[3] to a bare machine, called Stand-Alone RTLinux-GPL (SA-RTL). Here too, the intention was to create a real-time kernel which is suitable for systems with low resources.

The work was done by an incremental code migration from the RTLinux tree to the code base. Though many code was directly transferred, the original Linux identity, however, was completely eliminated from the new system. The final system provides solely the RTLinux API. Additionally, a new multi level memory protection scheme was introduced. Therewith it shall be possible to protect not only the core RTLinux executive but also the mutual real-time tasks. The implementation of this protection scheme is achieved with very low overhead.

In autumn 2006 the successor of the previous work was presented by almost the same group. The new project was called “Embedded RTLinux” [17]. As in the meanwhile some drawbacks of the original SA-RTL implementation occurred, this new approach tried to solve these problems.

The problems identified were:

[2] If this is/was possible at all as both projects knows of each other.

[3] Actually, RTLinux GPL. The word “RTLinux” refers always to RTLinux GPL in this work.

- The compatibility of SA-RTL is limited to source level. This means that the source code of the desired RTLinux application has to be available and it must be recompiled for use in SA-RTL. Binary only applications will not run in the system.
- The maintainability of SA-RTL's code base. As already mentioned, SA-RTL was created by copying code from a specific version of RTLinux-GPL. This approach did not allowed to stay always synchronized with RTLinux' upstream. The result was complicated support and increased porting efforts with every new release of RTLinux.

The new Embedded RTLinux tries to solve these problems. Instead of completely eliminating the Linux system, it is just replaced with a thin software layer. Thus it is possible to run an unmodified version of RTLinux on top of this thin layer, achieving the desired binary compatibility.

Chapter 3

Design

The goal of this work is to create a new real-time operating system kernel which is suitable for embedded systems with low resources. This kernel has to provide the RTAI API as an interface between applications and the system. In the task formulation, some further development goals are stated:

- Small size of the resulting kernel image.

  Embedded devices often do without large hard disks. The reasons for this are manifold. Usually normal hard disks are not robust enough to provide continued service under the environmental conditions the embedded system is installed in. Otherwise there is often simply no need for large data memories so using hard disks for bootstrapping only would be a waste and increases the price of the whole system. For this reasons flash memory is the most commonly used solution today. Being mechanically resilient it also offers a good trade-off in cost-benefit considerations: larger flash memory is available relatively inexpensive in the mean while.

  Regardless of this more memory always implies a higher power consumption. So the overall embedded system profits when the operating system is already designed economically.

- Clear and efficient design.

  This goal is hardly to achieve as there is no explicit metric given. For example, it depends usually on the subjects previous knowledge to distinguish between “clear” and “unclean” design. However, implicit metrics are used for verifying the results. Since the Linux kernel is open-source, its complexity can be studied by everyone, so it serves as a base. The

result's complexity should not exceed this upper bound while being less complex is acceptable.

A metric for the efficiency is given by the comparison of the scheduling latencies. The values of the new RTOS should not be worse than these of an established Linux/RTAI system. Therefore measurings should be done to confirm this thesis.

Broadly, there are two possible approaches of dealing with this task. The first one is to take the existing RTAI code as a base and looking through the sources which (global) variables and data structures respectively which functions are used. It has to be examined if these are relevant for functionality on the given platform. In this case these have to be reimplemented later and pushed underneath the RTAI system. Otherwise they can replaced with a simple stub function or they could be simply dropped. Then the referring parts of the existing code need probably to be adopted to fit the new base system. While this approach gives the possibility to completely ignore the previous base system and therefore allows an entire redesign of the new code, this is at the same time the main disadvantage: the whole new code has to be written from scratch. This is not a problem by itself, however it could negatively influence the acceptance of the project by the industry which prefers matured and well-tested code to save cost-intensive test cases.

The second way takes an established Linux/RTAI system as a base and then cuts the unneeded parts away. The existing code has to be reviewed also to detect dependencies on functions and global variables. But this differs from the first approach in so far as it tries to reuse as much as possible of the existing code and base system. Particularly, the design of the taken over (sub)systems is left unchanged or only minimal adopted. In this respect this approach should guarantee or improve the acceptance in comparison to the first one. At first sight it looks easy to discover which parts of the code are needed and which ones can be dropped since the RTAI code is very modular and well-structured. The core RTAI modules mutually depends only weakly and mainly in a linear fashion i. e. one module uses only services of the other one and not vice versa.

However, the Linux part has to be analysed as well. And this code is not as well structured as it could and should be: if deselecting e. g. the sysfs feature via the kernel config system at compile time, there are still many code references to it and several data structures are included which are unused in the end.

This is because the kernel is not fully designed for systems with low resources, and at a normal system a few bytes of code and data more or less do not matter in the memory footprint. For this reason, it is necessary to review the whole Linux sources, moreover, there are many features and subsystems which cannot be deselected by the original configuration system at compile time. These code parts have to be dropped manually and the remaining references has to be deleted or replaced with stubs.

In this work the second approach was chosen for the following reasons. Firstly, the task formulation suggests this way indirectly by "eliminate the Linux subsystem". Another reason is already given above: the results of this work should not remain a proof-of-concept implementation but be already usable by real world applications. In addition it is guessed that maintaining a complete new operating system kernel, which was the result of the first approach, would require much more time and known-how as the maintenance of some patches for the established Linux system. And finally there are many concepts already implemented which seem to be optimal for the resulting system and a new OS kernel from scratch would be like reinventing the wheel. These concepts are briefly discussed in the following sections.

3.1 Token over concepts

3.1.1 Binary image layout

Usually, when creating software the processor instructions are not put in binary form into a file but a compiler collection i. e. a preprocessor, a compiler, an assembler and finally a linker are used. This tool chain translates the human readable source files to a binary representation which is executed on the target CPU. For the Linux kernel, most of the source code is written in the C programming language which is highly portable, only for some hardware-dependent low-level functions (or for performance reasons) assembler is used. To preserve readability the sources are split over multiple files, so that all parts have to be merged together finally. This step is done by the linker. The linker output does usually not only consist of native processing instructions but is rather a meta format containing additional information which is needed on the target OS to load the application. For the Linux kernel, the Executable and

Linking Format () is used as it is the default binary format for applications on most UNIX based operation systems and on Linux itself.

An ELF file can consist of several sections, mainly a text (code) section, a data and a so called (Block Started by Symbol) section. Additional sections may be included e. g. symbols for debugging or comments. The linker creates a ELF file upon the data found in the used linker script, which describes how the sections of the input files should be mapped into the output file and how the memory layout of the application looks like. Since the Linux kernel is not a common application, this meta information is not needed and so discarded in the further build process. However, the layout of the sections is kept in the final binary.

To run the Linux kernel on the target machine, the image has to be loaded into RAM and then control needs to be passed. On a architecture, the boot process has many historical "vices", the important ones are described later in detail. In this context, it is sufficient to mention, that the Linux kernel was traditionally prepended by a small boot sector and a loader which puts the remaining parts of the kernel into memory.

In Figure 3.1 a simplified overview is given of the various files generated during the build process and theirs binary layout for the IA-32 architecture.

The image layout mainly affects how bootloaders deal with the image. Nowadays, there exist lots of bootloaders for various environments. Many of them have built-in support for the Linux kernel. This is necessary because the bootloader is able to pass some additional information to the kernel during the boot process. As this is also a wanted feature of the new LRTAI system (cf. section 3.1.3), the "backup strategy" implemented in nearly every bootloader could not be used. This strategy of handling unknown "boot objects" is loading the whole image into memory and jumping to the first address where the image was loaded. Booting this way, no information could be passed to the kernel. Actually the kernel would not boot at all, since the included boot sector prints only a notice that booting without a loader is not supported anymore.

For reaching the mentioned design goal, the bootloaders ought to be expanded to support the new LRTAI kernels. Even though, many of the potentially used bootloaders are open source and could be adopted, it would be easier to use the existing support for Linux kernel and the way how parameters are passed. This results in keeping the major image layout for LRTAI so that bootloaders would detect a standard Linux kernel. This way, all familiar bootloaders would be able to bootstrap the LRTAI system as they can use their

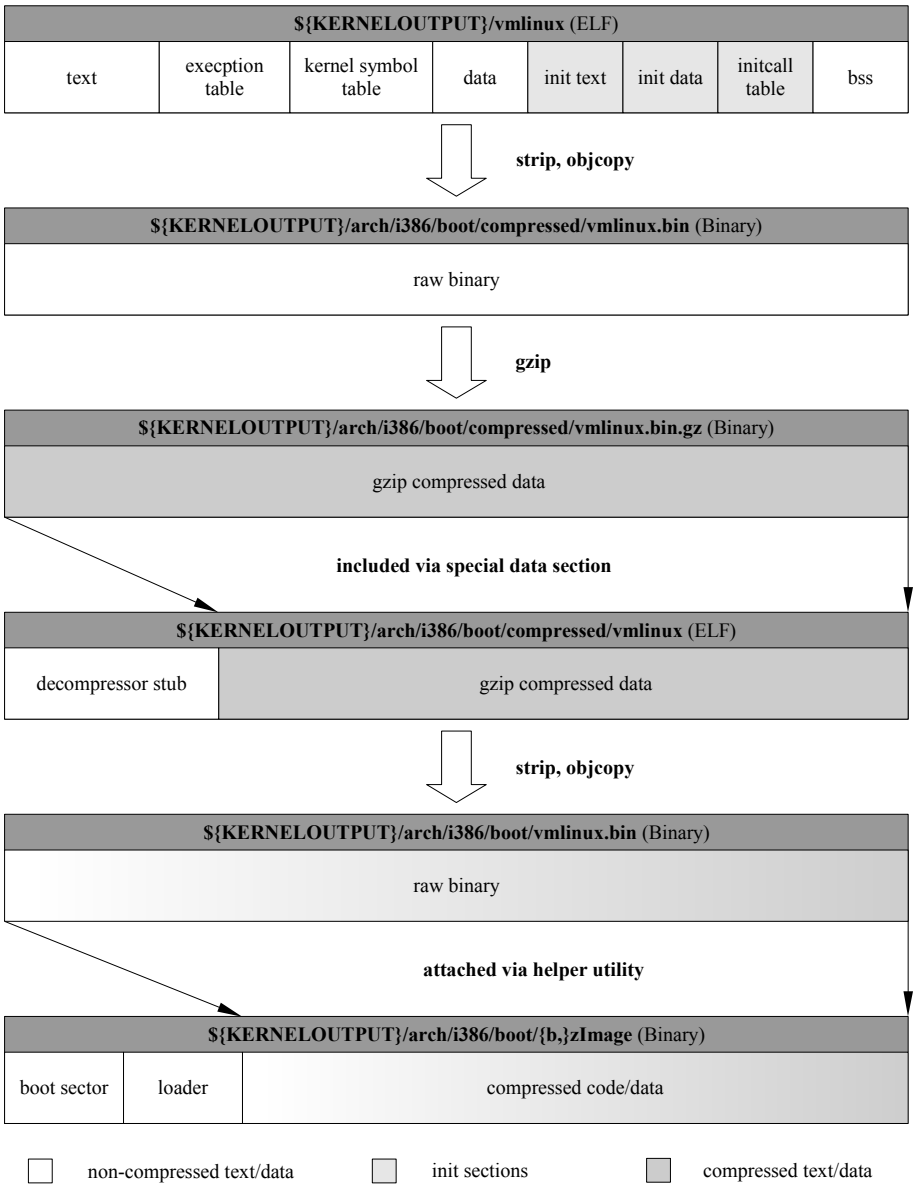

Figure 3.1: Simplified overview of vital file layouts on IA-32.

existing knowledge to determine e. g. the entry point.

3.1.2 Image compression

As the image size of the Linux kernel increased continuous over time by added features, some size constraints, notably those of IA-32, hit and limited the static core kernel. Compression of the image was chosen as the way out. The used compression algorithm is zlib [18] by Jean-loup Gailly and Mark Adler which was originally intended for compressing pictures.

Using image compression the kernel should be able to decompress itself which requires a small decompression stub in the final image. Of course the compression can only be justified when then added overhead plus the compressed part are smaller than the uncompressed piece of code.

As the zLib implementation is not an in-place algorithm the target system must have enough memory to hold both the compressed and the uncompressed images. On systems with extreme low resources this could be a problem as described later in section 4.3. This could be a potential argument against using compression at all, however, it is to the developer to make this decision and to choose the preferred or necessary variant.

Today there are already some bootloaders which natively support the kernel's compression. Then the bootloader itself decompresses the kernel image to its final location, usually by reading the data from a flash memory directly, so that it is not necessary to have both images in RAM.

Additionally when using compression, the booting time of the system is affected. While a smaller image results in loading less blocks e. g. from a disk and should speedup therefore the boot process, a low-performance CPU could spend more time in decompressing the code. On modern CPUs the decompression should be a negligible factor, however the time needed for booting the system can carry weight for an embedded system. Though this time depends on various conditions, it is still possible to measure it for a specific system as it should remain constant.

For LRTAI, support for image compression is desirable since it allows the user to burn the image into a small flash chip instead of using large hard disks. Even though larger flash memory chips becomes available, a small footprint of the base system leaves a margin for more user-defined data or functionality.

Another option could be to include both a "live" and a "rescue" system in the same flash memory. This could be used to provide the user software updates

which overwrite the live area while keeping a fail-safe variant in another flash segment. If an update fails or the new system is flawy the user could easily activate the backup system e. g. by pushing a reset button while powering on the device.

3.1.3 Kernel command line

As already mentioned above, the original Linux kernel has the ability to receive information during boot process from the bootloader and/or thereby from the user itself indirectly. This is mostly used to pass (semi-)dynamic configuration information, preventing the user from compiling the whole kernel a new when some minor changes are needed.

This interface is represented as a simple command line string, which can filled with several tokens and/or key-value pairs. These arguments are usually preconfigured in the bootloaders configuration. Additionally many bootloaders permit the user to modify this string. Disclosing it finally to the kernel is done by writing the memory address of the string to a well-known memory address inside the loaded kernel image. Then while booting the kernel iterates over the elements of the command line, invoking a callback function which was registered at compile time for each possible element. Many callbacks simply sets an internal kernel variable to a new value, but also complex functions are possible.

Since this concept is quite simple and anyhow powerful, it is kept in LRTAI. Additionally, as it is widely supported by bootloaders, it guarantees the user highest flexibility.

3.1.4 Initcalls

During kernel configuration at compile time, the user has for nearly all features the possibility to choose between building it as dynamically loadable kernel module or linking it statically into the kernel. Mostly, a feature encapsulates a specific subsystem e. g. a hardware driver. This often needs an explicit initialization procedure e. g. resetting the corresponding hardware device.

When the feature is inserted into kernel as module at runtime, the kernel uses a well-known interface to invoke the module's initialization function if one is defined. The author of the module/feature can use predefined macros for this

in the source code, providing a kind of abstraction layer so that the real kernel implementation could change.

Actually, this source level abstraction for kernel modules is also used when linking the module statically into to kernel. Then the mentioned macros are redefined so that every occurrence adds an entry to a so called *initcall table*. This initcall table is simply a list of pointers to initialization functions. These are grouped by functionality or precedence into separate subsections. Currently, there are seven predefined subsections, namely *core_initcall*, *postcore_initcall*, *arch_initcall*, *subsys_initcall*, *fs_initcall*, *device_initcall* and *late_initcall*. During boot process, the initcall table is processed in order, starting with the core initcalls. Ordering inside the subsections is determined by link order i. e. a module which is referred to later in the makefiles—and therefore linked in later—is also initialized later.

Even though, many original subsystems of Linux are dropped for LRTAI and hence the remaining ones could be initialized by hand, this concept is overtoken to LRTAI. This is to achieve source code compatibility to existing RTAI modules which make usually use of the described macros. Since the module support is dropped for this first LRTAI version, the macros expands to the initcall table implementation as described. More exactly the usually used *module_init* translates to a *device_initcall*. That will be later from interest when the memory allocations are discussed.

3.1.5 Initialization memory freeing

In Figure 3.1 an area in *vmlinux* is highlighted. In this area the linker places initialization data and functions which are explicitly marked by the authors in the source code. The motive is that such marked functions or data are solely used during the boot process. This could be e. g. the initialization function of a driver. After execution of the code, it is not needed anymore, could be dropped from memory and the resulting free space could be given to the memory management.

Especially on system with memory constraints, such a feature is very useful. Since LRTAI is already stripped down to a minimum, there are only few functions remaining which can be freed after usage. Nevertheless, this feature is implemented. This is to support the users when they needs or expect such behavior and to optimize the memory footprint of LRTAI.

3.1.6 Memory management

Every operation system needs to manage the available memory. Usually, this is done with multiple layers or levels for simplicity. On a standard Linux/RTAI system, there are up to three memory subsystems involved before a real-time application receives its requested memory block.

3.1.6.1 Bootmem memory allocator

The closest level to the hardware is the memory page management where whole physical pages are marked as used or free. Since a page is typically 4 KiB on IA-32, it would be a waste when an application tries to allocate a few hundreds of bytes and is given a whole page. So an additional level of memory allocation is introduced, the SLAB (or especially as replacement for embedded systems: SLOB) allocator. This second stage is described in the following subsection.

Since the original page management of Linux is quite complex and strongly interwoven with the remaining memory management and therefore the file and file system handling code, the decision was to replace and simplify the whole system.

Fortunately the original Linux comes with an alternate page management system which was designed to be used solely during booting: the bootmem allocator. This is a small bitmap based allocator, which will completely satisfy our requirements for lowest-level allocations. Originally the code is marked to be freed after initialization completes (see above), so it requires some changes to fit for the new LRTAI system.

Also the code assumes that is running solely on one CPU as SMP support is normally not yet enabled at the stage when the bootmem allocator is used. To prepare LRTAI for SMP—in the first implementation SMP support will not yet be included—the allocator is also enveloped with spinlocks to prevent concurrent modifications of the internal data structures by multiple threads/tasks. When compiling for uniprocessor systems, the kernel's macro magic will optimise the calls to simple interrupt preventing implementations, so there will be no overhead introduced.

3.1.6.2 The Nano SLOB allocator

The second level of memory management is normally done by the SLAB allocator, originally developed by Jeff Bonwick for the SunOS 5.4 kernel [20].

Since this allocator is quite complex, a replacement exists. This replacement is the so called SLOB system, which can be chosen at compile time. It was designed to be used on embedded systems which do not need the full power of the SLAB system. As it provides SLAB's interface, it replaces the traditional SLAB system silently.

The SLOB allocator serves as a base. Since it requires the original page management by default, it has to be adopted to use the bootmem allocator which will manage the physical free pages not solely while booting. Additionally some functions concerning cache management are dropped since these are mainly needed by device drivers and other kernel subsystems which will be dropped from LRTAI.

The remaining SLOB allocator solely supplies the *kmalloc*, *ksize* and *kfree* functions. It is only used during the boot process where various functions of the original Linux kernel requires dynamically allocated memory. Another use case is when a RTAI module desires its own real-time heap. In a Linux/RTAI system it would allocate a chunk of memory via *kmalloc* and passes this block to a RTAI function which converts the block into a real-time aware heap.

So the existence of the SLOB layer is mainly for retaining compatibility with the described application case. The alternative choice would have been to map the three above mentioned functions to the RTAI's counterparts. But then major changes to the RTAI level would have been necessary. Also the special use case above would result to a "RTAI heap in RTAI heap" scenario, which could potentially lead to confusion.

3.1.6.3 RTAI's own memory management

RTAI comes with an additional layer of memory management as already became evident. This is necessary to fulfill hard real-time constraints. Since the original Linux system tries always to serve memory requests e. g. by freeing unused caches, it can suspend the execution of the requesting task. However, using this mechanisms could break the timing of a real-time application.

The RTAI module *rtai_malloc* bypasses these problems by preallocating memory blocks from the Linux memory manager and by distributing this space via its own interface to RTAI applications. To fulfill the timing constraints, it can be forced to operate in a time bounded fashion. Thereto the used memory heap, from which the request should be served, needs to be non-extendable. Otherwise, on a standard Linux/RTAI system when the heap is marked extend-

able, the allocation request can also trigger a dynamic enlargement losing hard real-time. Although this functionality is (still) documented in the manuals, there was no code found in the sources which provides this feature. Support for dynamic expansion seems to be dropped silently from RTAI's upstream. It could be reimplemented easily with a few lines of code. But since in LRTAI only hard real-time tasks exists which usually should not use this feature, this is not necessary.

While configuring the RTAI sources at compile time, the user can choose whether the RTAI module should use Linux' *kmalloc* or *vmalloc*. The latter one does not give any advantage on LRTAI, actually, the whole memory in LRTAI will be linearly mapped, so that there is no difference between both variants. For LRTAI this configuration option is therefore preselected to *kmalloc*.

Another choice is the amount of memory which should be preallocated from Linux. Here the user presets the size of the global heap which is provided and used by RTAI applications if these do not allocate their own heap. It is still possible to change this value at runtime passing a module parameter when loading the module. By the way, for compiled-in modules such parameters can also be set via the kernel command line. Then the parameter has to be prepended by the modules name and a dot e. g. the parameter *rtai_global_heap_size* becomes *rtai_malloc.rtai_global_heap_size*. This way the user could still set the memory which RTAI should use.

Actually, the user has to preset this value in the bootloader if he/she had not compiled in the right value. If the compiled-in value is larger than the available memory, the initialization routine exits with an error an the global heap is not initialized at all. On the other hand, if the value is to small, there might be much memory which can not be used for RTAI's global heap. So this needs some configuration overhead at runtime and makes this method unfavourable.

Also this initialization procedure has a further drawback. It requests the memory from the Linux memory management in blocks of *extentsize* bytes. As the default *kmalloc* implementation poses an upper limit of 128 KiB for each request, RTAI uses this limit which is neither configurable at compile time (without direct modification of the sources of course) nor at runtime. As by definition all extents are of the same size, for small systems this extentsize could be to large and could result in memory blocks which remain unusable.

For LRTAI this initialization code has to be rewritten to eliminate this two drawbacks or one or more new initialization strategy should be implemented. The later approach was chosen to provide the user the highest compatibility but

also to fit optimally to the new environment. The new implementation lets the user to choose at compile which allocation strategies are compiled in, and at runtime which one is actually token.

Actually two new algorithms are added. The first is called “greedy” as it tries to allocate as much memory as possible from the bootmem allocator. Like the original approach this is done in smaller or larger extents but the extent size could also be given dynamically. The method has still the drawback that memory requests which are larger than one extent can not be satisfied as RTAI’s allocation function can not split the request over multiple extents.

A further algorithm should be provided to support such large allocations. It is called “largest” as it simply allocates the largest available free block in one step. It is not divided into smaller extents but forms only one extent. The special case where multiple large blocks of the same size are available is not handled, even though, these could be treated as several extents. It could be implemented easily if such functionality is needed.

As mentioned above, the user can choose between these two new algorithms and the “traditional” one using the kernel command line. This is also used to pass the parameters e.g the extentsize to the greedy algorithm. As a default the greedy implementation is chosen for LRTAI with an extent size of 128 KiB.

3.1.6.4 Private heaps of real-time applications

RTAI provides applications the possibility to use not only RTAI’s global heap but to request their own memory block which is “converted” into a private heap. This is useful e. g. when an application exactly knows about its memory consumption and does not want to interfere with other applications.

The normal process of registering such a private heap is to allocate a memory block of the desired size from Linux’ SLAB/SLOB system which is later passed to RTAI. With respect of the initialization order of the embedded modules and when using a modified global heap initialization algorithm as described above, the situation can occur that the requested blocksize is not available anymore when the applications memory request is executed.

Without modifications to the application’s sources this problem can only be solved by changing the global heap allocation algorithm as described or at least restricting its “greediness”. During LRTAI’s configuration a margin could be given which is not allocated and therefore can be obtained from *kmalloc*. Other solutions implies at least slight changes in the application sources. One solu-

tion could be to use the global heap instead of spanning up its own. However, this could be not wanted. Another way could be to allocate a big memory block at a whole from the global heap. This results in the heap-in-heap phenomenon. The third variant could be to split up the memory allocation from the remaining initialization code. To that the order of the initcalls can be used. As described in section 3.1.4 the *module_init* macro expands to *device_initcall* which is executed lately. Assuming that *arch_initcall* and *subsys_initcall* should not be "misused", the remaining candidate is *fs_initcall*. This choice seems reasonable as LRTAI comes without filesystem support.

It has to be mentioned, that a traditional kernel module should only have one initialization macro referred when compiling as a loadable module. So to achieve source highest compatibility, the second call should be hidden with some kind of preprocessor magic. An extract of a possible implementation is given in Appendix A.

3.1.7 Console output via printk

Normally, a computer system has the possibility to display errors or other messages to the user. Combined with the option for receiving user input, Linux defines the concept of a *console* like many other UNIX-like operating systems do. This console is simply a data structure with pointers to functions that will input or output the given data. The default console usually uses the build-in video graphics card for text output to a monitor and a directly attached keyboard for data input. But it can alternatively use a serial port, a line printer etc.

For LRTAI the full power of this concept is not used. The main purpose is to get messages out to an attached display. The possibility to input data via a keyboard has to be reimplemented later as a real-time aware device driver.

For embedded devices it is not unusual to have no display attached as there is often no need for it. For debugging purposes a serial console is often used instead. But using a serial port is difficult on real-time systems: if using a polling mode the system has to actively wait for all status changes. The other, interrupt driven approach requires a working interrupt system of course, so output would not be possible if the interrupt system is not yet started.

As default LRTAI will assume that a VGA compatible video card is installed. Such a card has a video memory which is mapped into the I/O address range. In text mode writing to the screen is simply copying the data from a buffer to the

video memory, no need for dispatching interrupts or other complex stuff. This is used by the early vga console, which is included in the default Linux system. But it is not activated by default, so the LRTAI version has to be modified.

On regular systems the early console is dropped as soon as the remaining kernel drivers are initialized, so that these can overtake the management of the console. Since there is only limited driver support for various hardware on LRTAI and the functionality of the early vga console is sufficient, support for replacing it during boot process is not needed and omitted.

3.1.8 Tracking time with jiffies

Keeping track of time is one of the most critical tasks in every operating system. The Linux kernel therefore uses a global variable *jiffies* which is a simple 64-bit counter. It gets incremented with every tick of the timer. The time lag between two timer ticks depends on the target platform and a configuration value set at compile time. On IA-32 it is traditionally 10 ms, in newer kernel versions and/or on newer systems 4 ms or even 1 ms are not unusual.

RTAI uses this global variable in some places when it was configured to use the Intel 8254 as timer chip. Since support for s is not yet included in LRTAI, the preconfiguration has to choose the 8254 code paths. Therefore the jiffies counter needs to be included, too. The original Linux timer interrupt handler is reduced to update this variable solely. This interrupt handler is the only one which remains from the original Linux part, all other s are managed in the real-time domain.

3.2 Spurned features

As a general purpose operation system, Linux has a lot of features. Many of them would be useful if included in LRTAI, some feature implementations are actually required by RTAI while sharing the existing code (e. g. spinlocks). However, including many features implies are large footprint in the end. Reducing the system to fit on small embedded systems means to omit code and functionality.

Many parts of the original Linux are not strictly necessary for an RTOS. Especially, since the new LRTAI should only provide the original RTAI API,

some features can simply dropped as there is no corresponding or equivalent functionality.

It is for this reason that for instance the filesystem layer can be completely omitted. The original RTAI does not care about the representation of data on disk or other media. It relies on the Linux subsystem to manage these tasks. For instance while bootstrapping new processes from loadable RTAI modules: these modules are usually stored as a file in an filesystem. When a module should be loaded into the kernel, it is handled like a usual Linux kernel module which means that is is loaded by the *insmod* command line tool. This tools opens the file and copies the content to a memory location in user-space. After that, a system call is invoked to insert the module in kernel space and to start execution of the new code if the module provides a defined entry point. While running this initialization code the module can fork of new real-time processes which are subsequently managed by the RTAI domain. Thus it appears that the RTAI API starts at process level, there are no API functions which deal with loading or storing data on disks etc. Thus in LRTAI is no need for such functionality too and the code concerned can be dropped from the project.

With dropping filesystem support, the ability to dynamically insert modules at runtime is also lost since there is no source anymore which could hold the modules object code. A possible solution would be to feed a module via serial port to the system. Since a real-time aware device driver for the serial port exists in RTAI, a serial protocol could be established between a host system and the embedded device to transfer object code to the targets RAM. After the transfer, the existing routines of RTAI could be invoked to initialize the new code and to start new processes as requested by the modules initialization procedures.

As the main focus of this work to get a standalone RTAI system running, the ability to dynamically load modules is considered as a feature what would be nice to have but not strictly required. It is further assumed that the code which is intended to run on an embedded target system is infrequently changed and therefore can be included in the kernel image at compile time. It results that the sources of the included code have to be available and only source level capability to existing RTAI projects can be achieved. Another reason for omitting a loader is that tests of new modules could be done on normal PC systems where dynamic loading is available. When the development process of the module is finished, it can be ported easily to LRTAI as only minor changes in the core implementation are done. Thus the behaviour is nearly identical to the developer

machine when the same release of the kernel and RTAI is used.

Another topic is the device driver support in LRTAI. Since all parts of the original Linux kernel run at the same CPU privilege level (on x86 all code in kernel mode runs at the so called "ring 0"), all code has invariably access to the hardware. So no further differentiation between core kernel parts, network subsystem or in particular drivers can be made. This opens the possibility to device drivers to disable the interrupt system completely on the CPU. If this is used by a driver implementation it would break the whole hard real-time environment. To prevent this, all drivers has to be implemented with these facts in mind. RTAI therefore comes with its own implementation of a serial port device driver which is real-time aware. All other original Linux drivers have to be audited before they could be used. In LRTAI many parts of the infrastructure, which original device drivers are used to found, are dropped. So they could not be token over directly and a code review is needed. But since LRTAI should only provide a framework which can be adapted to special needs by its future users, such driver examinations are assumed to be done by the users. Only the drivers for the interrupt/timer circuit are included by LRTAI.

With omitting the remaining device driver and since the filesystem support is dropped, the infrastructure and the need for the block and character devices is lost too. These special filesystem nodes enabled user-space applications and system tools to communicate with drivers i. e. to load and store data and/or setting configuration values.

As can be seen from the task formulation, the "Linux subsystem" should be "eliminated" from the project. However, an exact definition of the term was not given so the most logical one is assumed: it is supposed that the Linux subsystem covers all kernel parts which are unconditionally needed for spanning the user-space. That means in particular that the Linux system calls (*syscalls*) are not longer required since they are solely used by user-space applications. Since the kernel subsystems are located in the same address space they can call the requested functions directly and does not need to use the indirection of syscalls. The implementation of the original Linux' syscalls is usually composed of at least two functions for each syscall. One function encapsulates the main functionality and can be considered as the "worker code", the other one is used as the kernel-mode entry function, which is indirectly called from the user-space. This wrapper usually checks the given arguments, calls the worker and returns the result or error code to the user-space. For LRTAI all wrapper

functions can be dropped since the user-space is not used anymore. However, the remaining code has to examined if it still requires some worker functions.

The Linux subsystem consists additionally of all code which handles the tasks in user-space. This is mainly the scheduler but also the signal management or the capability subsystem for example. In kernel mode there should not remain any thread or activity which is not covered by the RTAI execution domain. That means that all kernel threads and real-time applications has to managed by RTAIs own scheduler. This included all interrupt service routines with one exception: the original timer interrupt is not migrated to RTAI and will be executed in the IPIPE's Linux domain.

At last, the Linux memory management is dropped. Being one of the most significant struts which characterizes an operating system the code does not fit into the LRTAI project. The reason for this is that the code is quite complex and provides a lot of functionality which is not required anymore. For example with eliminating the user-space functions which care about correct user-space to kernel mode transition and back again are not needed anymore. The same applies to the ability to memory-map files. Additionally, the memory management is strongly interwoven with the filesystem layer which is going to be dropped. These dependencies result from performance demands which are put on a general purpose operating system like Linux is. However, for a tight and small real-time kernel this over-fulfill the requirements.

Since a working memory management is fundamental for an OS, Linux tries to setup it as soon as possible while booting. Before this setup process finishes, it uses a boot time configuration which has less functionality and is limited to a total of 8 MiB RAM. Less functionality hereby means, that the page table are statically initialized with a linear mapping between the available memory and the existing address space. However, this configuration is for LRTAI sufficient so it is not replaced with an other memory management system but is kept after the boot process.

Some code parts which relay on the mapping or unmapping of memory ranges has to be dropped in consequence. Since this affects mainly functions which tries to work-around some hardware flaws, this is of no further interest. For example there was a bug in the early Intel Pentium models, which can be prevented with a special memory mapping.[1] It is assumed that LRTAI will not be used on such hardware e. g. that hardware is operating correctly.

[1] The bug is known as the F00F-Bug. See [19] for details.

3.3 Merged build systems

As already describe, the original Linux build system and RTAI's one uses the same configuration system before compiling. So it seems to be the best solution to integrate RTAI's configuration in the existing one of Linux. But not only the configuration stage is used, but also the complete original build system for the following reasons.

Since there will be no user-space anymore, all real-time applications are built-in into the kernel so the need for the user-space tool chain support has gone. Also cross compiling support is given by Linux' tree. The Linux kernel itself comes with a lot of configurable features, changes to system would be hard to maintain. In comparison, after stripping down RTAI to its core elements, only a few features will remain. Since most configuration choices come with a sensible preset and some items are hardly prechosen for LRTAI, the need of the configuration would be only to select additional services e. g. for inter-process communications. However, as some new configuration items were added to be able to choose between the global heap allocation methods, a modified version of RTAI's configuration should be merged into the kernel's one.

The configuration items which will be newly present are now managed with the kernel's dynamic include system. The *rtai_config.h* which is normally the central include file for RTAI applications and generated after configuration, degenerates to a static file which holds some not yet migrated or legacy definitions and refers for the rest to Linux' includes.

Actually, the LRTAI will not yet be well-prepared for the most configurable options on Linux side. For example choosing LAPIC support would be desirable, however, support in not yet implemented in LRTAI. So it ships with a sensible preset of the *.config* file.

For compiling the RTAI source files within the Linux tree, the required C files are simply copied into a new sub directory. The existing Linux top makefile has to be adopted to descend into this new directory. As the original RTAI makefiles depends heavily on the auto-generated makefiles which are produced by the mentioned user-space tool chain system, they can not be simply copied but have to be replaced. Because the kernel build system brings along a lot of functionality, these replacements are quite simply and often only consist of a few lines. So it will be easy to transfer additional RTAI features

which are not yet included in LRTAI.

Except the mentioned adaption of the Linux top makefile and the replaced ones for the RTAI subtree, there should be no changes necessary. This will keep the efforts for maintenance low e. g. when/if the system is ported to a new kernel version.

Chapter 4

Implementation details

4.1 The boot process

The process of booting a Linux/RTAI system consists of a number of stages. When the system is powered on or reset, the CPU instruction pointer register is set to a hard-wired, well-known value and thus executing code at a well-known location. In standard PCs this code is located in the systems BIOS, stored in a small flash memory on the motherboard. Usually, a modern BIOS is very flexible in the further boot process i. e. the user can choose between a wide range of boot media e. g. floppy disks, hard disks, () flash memory or even the network. In case of booting from a disk and the decision which device is used (if multiple ones are present in the system), the tries to load the first sector from this device, usually 512 byte, into memory and executes this code by jumping to the first address of the loaded sector. Since a normal[1] bootloader does not fit into the maximum available size of 446 bytes (see Figure 4.1), this first sector contains the first stage of the bootloader and the code's job is solely to locate a further stage of the loader. This is done traditionally by scanning through the partition table for an active flagged partition and loading a predefined number of additional sectors from this partition. Modern bootloaders (e. g.) does not rely on this partition flag but the first stage code is configured statically with the sector addresses of the following code. Adhering to the example of GRUB, this is called "stage 1.5" and implements a tiny filesystem driver for the target partition so it varies between the used filesystems. Having loaded that code, the bootloader has gained the ability to find its further stage(s) and finally the kernel image etc. in the filesystem which provides a larger flexibility and stops

[1] Menu-based or even graphical GUI

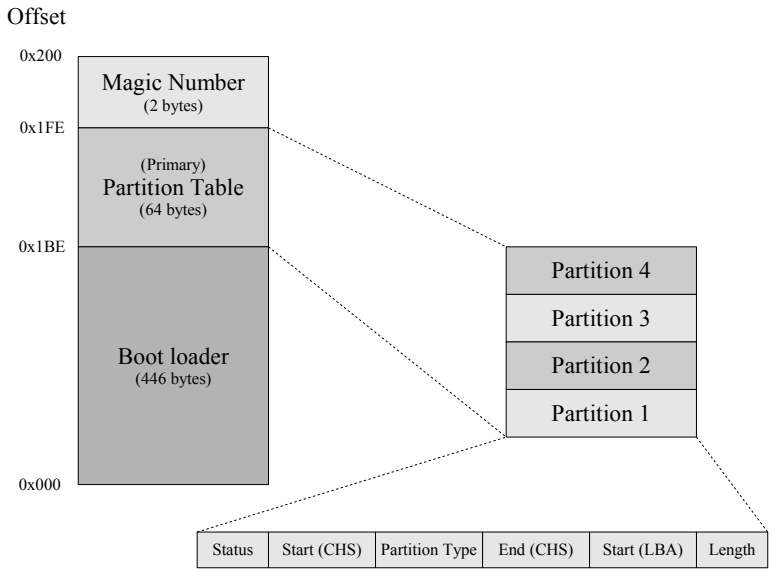

Figure 4.1: Traditional boot sector layout on a PC architecture.

the need for updating the master boot sector if the position of the kernel image on disk changes.

Usually this is also used to give the user the possibility to choose between multiple kernel images. After this selection, the kernel image is loaded into RAM and the bootloader passes control to the kernel by jumping to a well-known start address in the kernel image. The processor is still running in real-mode at this stage, therefore the kernel entry has to be also 16-bit code. This entry point is the *startup* function (located in *arch/i386/boot/setup.S*) which does some elementary initialization e. g. tries to determine how much memory is installed. Later, the protected mode part, which is still compressed, is moved down in memory. After that preparations, the CPU is finally switched into protected mode and a *startup_32* (*arch/i386/boot/compressed/head.S*) function is called. This routine sets up a basic environment e. g. a tiny stack and clears the Block Started by Symbol (BSS) area. Subsequently the underlying kernel is decompressed through a call to a *C* function *decompress_kernel* (*arch/-*

i386/boot/compressed/misc.c). This code is located in a small non-compressed stub which heads the compressed part (cf. Figure 3.1). The remaining non-compressed code just places the image at the memory location which was chosen at configuration time. This will be discussed in detail in section 4.3. Finally, yet another *startup_32* (located in *arch/i386/kernel/head.S*) function is called which initializes the page tables, detects the CPU type and the and starts the paging. Then it passes control to *start_kernel* (*init/main.c*) which runs the non-architecture specific boot routines. This can be regarded as the kernel's *main* function in comparison to a normal *C* program. Figure 4.2 illustrates an example of a boot process using GRUB from a hard disk until the call of *startup_32* in *arch/i386/kernel/head.S*.

As already mentioned, *start_kernel* is a high-level, not architecture dependent initialization function. However, its first step is to call a further high-level, but architecture variable procedure, by name *setup_arch*. In this function, a data structure containing various information about the CPU is filled. This structure is used at multiple places all over the code as it holds, among others, the list of features supported by the CPU. Further on, *setup_arch* prints the memory map provided by the BIOS, preprocesses early command line parameters and finally setups the main memory. At this stage the bootmem allocator is also setup.

Back in *start_kernel*, the remaining part of the command line is processed and the trap and interrupt system is setup. After initializing the time subsystem, which solely increments the *jiffies* counter, the IPIPE root domain is opened and the interrupt subsystem is enabled.

The *start_kernel* function is marked as an *__init* function (cf. section 3.1.5). This means that the code would be freed after the initialization completes. So its last call is to invoke a non-*__init* function, namely *rest_init*, which finalizes the kernel startup. Here the already mentioned *init_calls* are processed and the *__init* memory section is freed. After that, the CPU enters the idle loop.

The RTAI domain's initialization is done via the initcall mechanism. As the RTAI code was originally implemented as kernel module, the sources use the *module_init* macro to mark the module's entry functions. While linking a kernel module statically into the kernel, this macro translates to the initcall mechanism. While in a standard Linux/RTAI system the modules's initialization order is determined by the order of loading the modules, the initcalls are invoked in order of linking the module sources into to main binary. This order can be controlled by the arrangement in the makefiles. To keep the or-

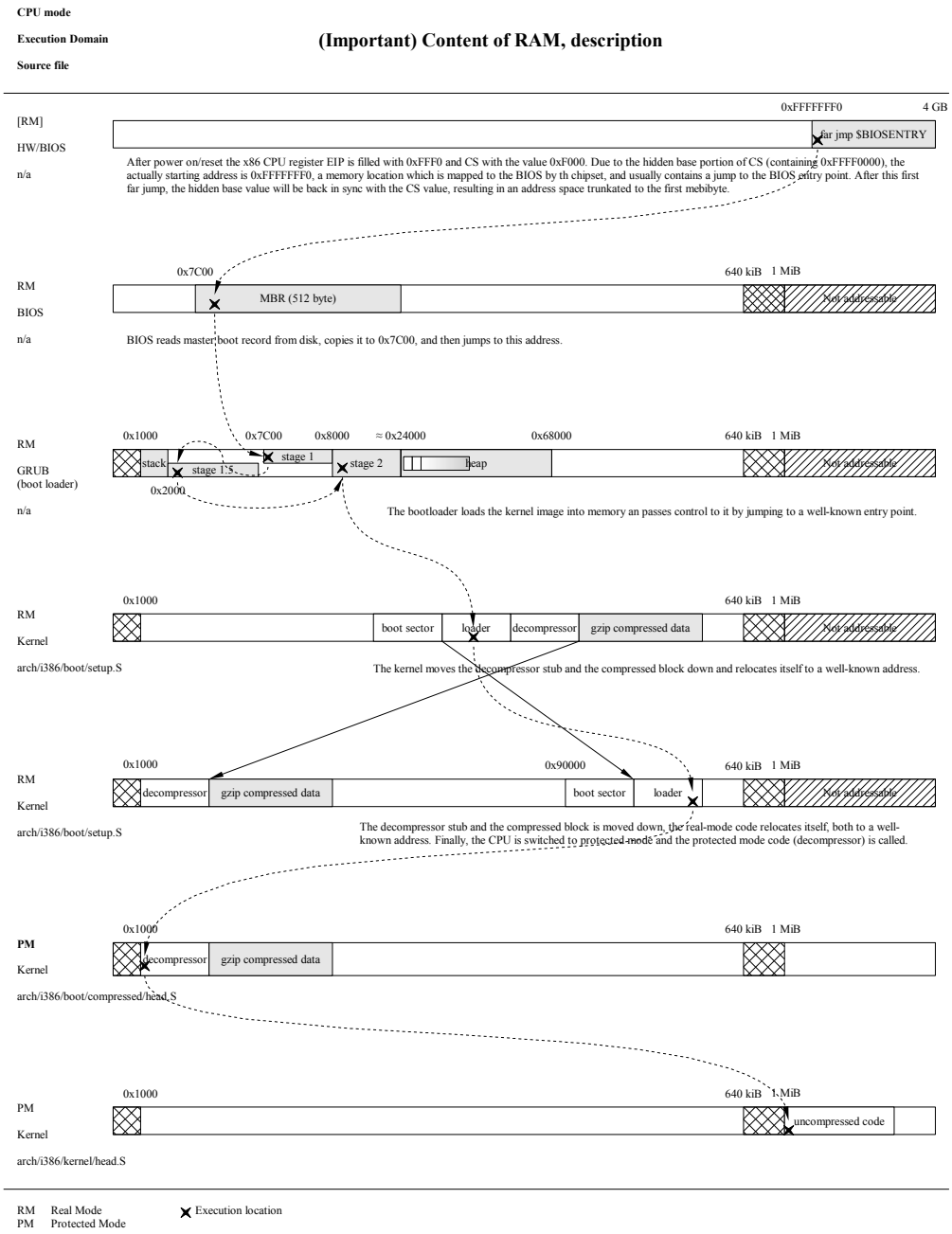

Figure 4.2: Simplified exemplary boot process on IA-32.

der in comparison to a standard Linux/RTAI system, the relevant makefile was written with this in mind. In the end, the (original) *rtai_hal* module is initialized first, which registers the RTAI domain in the IPIPE system and setups the basic interrupt system for RTAI. After this, RTAI's own memory allocator is installed (module *rtai_malloc*). This modules initializes a global heap which can be used by RTAI applications. Finally, the RTAI scheduler (*rtai_sched*) is setup and the timer interrupt system is enabled. After this, an elementary RTAI system is up and running. In the LRTAI sources a trivial sample RTAI application is included which is initialized after the above steps have finished.

When a LRTAI user adds his/her own applications to the source tree, he/she has to take care of this initialization order. A special sub-directory *rtai/apps* is provided where the custom applications and the corresponding makefile should be placed. The higher level makefiles ensure that this directory is linked in after the RTAI core components. This way, the user has to track only his/her own module dependencies and it is guaranteed that the RTAI subsystem is already available when the custom applications start.

Note, that at least one application has to trigger the RTAI domain execution by calling *start_rt_timer*. This is not done by default as the period to use should be specified by the user to fit best the application's requirements.

4.2 The nano SLOB memory allocator

As already state above the memory management is the most critical task of an operating system. Especially when the target system has restricted resources it is essential to maximize the utilization and efficiency. While in a standard desktop system an economic usage of the available RAM is a minor issue, nowadays a desktop has plenty of RAM installed, and other factors (e. g. interactive response time) emerge, an embedded system's OS should not allow to leave resources lying dormant.

In a Linux system most memory requests are handled by the SLAB or SLOB system. As already mentioned in section 3.1.6.2 the SLOB system was chosen for LRTAI because it is much simpler than the SLAB system but it provides the same API. In general these allocator systems were primarily developed to prevent memory fragmentation as a result of frequently and small memory allocations. Small means in this context that the requested memory size is usually less than the physical page size. However also larger requests are handled by

Requested blocksize	Required pages	SLOB allocation Order	SLOB allocation Pages	Worst-case waste
1 KiB + 1 B	2	1	2	0
2 KiB + 1 B	3	2	4	1
16 KiB + 1 B	5	3	8	3
32 KiB + 1 B	9	4	16	7
64 KiB + 1 B	17	5	32	15
128 KiB + 1 B	33	6	64	31
256 KiB + 1 B	65	7	128	63
512 KiB + 1 B	129	8	256	127
1 MiB + 1 B	257	9	512	255
2 MiB + 1 B	513	10	1024	511
4 MiB + 1 B	1025	11	2048	1023
8 MiB + 1 B	2049	12	4096	2047

Table 4.1: Exemplary worst-case blocksizes of memory requests, which maximizes the wasted page count of the allocated block for a particular order if Linux' default SLOB allocator would be used.

these systems.

If the requested memory size is greater than the size of a page the SLOB allocator calculates the so called *order* of the request. This order is the logarithmic size of the group of contiguous pages which will be requested from the memory manager below the SLOB system:

$$order = \left\lceil \log_2 \frac{requested\_bs + \text{PAGE\_SIZE} - 1}{\text{PAGE\_SIZE}} \right\rceil$$

The assigned block size arises out of:

$$assigned\_bs = \text{PAGE\_SIZE} \cdot 2^{order}$$

If an application does not requests (accidental) the calculated and assigned block size the so assigned block is always larger than the actually requested one and a trailing part of the block will be unused. This space can not be reclaimed by further memory requests unless the original application wishes to resize the block. If the new size still fits into the already assigned block this

can be easily performed, a major benefit of this approach. It depends largely on the used applications if this resizing support is really necessary.

The back of this procedure is that the larger the request is the larger is the assigned block. The problem is here that the finally assigned block size has to be available as free memory, not only the requested size. This increases the probability of larger memory allocations to be rejected. An example: a requested blocksize of 524289 Byte is assumed. To satisfy the allocation a minimum of 129 pages must be free. The SLOB allocator calculates an order of eight which would result in an actually memory allocation of 256 pages. If the count of free pages is between 129 and 255, the allocation would fail though it could be successfully performed. If further a successful allocation is assumed and that the application does not try to resize the block later 127 pages are wasted. Of course this is one of the worst-case scenarios. Table 4.1 lists for each order an exemplary worst-case blocksize resulting in a maximum of wasted pages of the allocated block.

In a LRTAI system, there are only a few memory allocations which accesses the SLOB system directly. The usual case is that applications acquire their memory by calling the corresponding RTAI functions which then again call the SLOB system. However, as the RTAI system acts as a proxy, it usually requests blocks from the SLOB allocator whose sizes are a multiples of a page size.

For this reason the support for resizing an already assigned memory area is only a minor issue. There against the potential waste of memory should be minimized. For this the SLOB allocator was modified to drop the concept of acquiring blocks by their orders. Actually the order is simply redefined to the minimal count of pages which are necessary to satisfy the requested block:

$$order = \left\lfloor \frac{requested\_bs + \text{PAGE\_SIZE} - 1}{\text{PAGE\_SIZE}} \right\rfloor$$

While the functionality of resizing a block is still available, it is now degenerated so that a resize request could increase the block size at most by *PAGE_SIZE - 1* bytes. However this is also the worst-case value of wasted memory space for each allocation request to the modified SLOB system.

With replacing this allocation mechanism, a bug in the SLOB system of the used kernel version silently disappears. As pointed out above, the order of the requested block size is determined by the allocator. Interestingly enough

this calculation is faulty in the original kernel version for blocksizes which are close to a wrap to the next higher order. The consequence of this wrong computation is that the returned memory block is to small. However, the requesting application does not known this. If it fully utilizes the memory area, there is a high probability that is overwrites code or data from other applications. This would be even more badly as there is no memory protection between the core kernel and the RTAI applications. Unexpected crashed or core dumps would have been a typical reaction.

The reason for calling the modified SLOB system "nano SLOB allocator" was already mentioned in section 3.1.6.2. All functions which deal with the cache memory management (originating from SLAB, but also build on top of the SLOB system) were dropped. This concerns concretely the functions which name starts with *kmem_cache_...*. As the RTAI subsystem does not use these memory caches and all other modules which usually uses these caches are also already dropped, these functions were no longer necessary.

4.3 Memory layout

On systems with low memory it is not only essential to known how much memory is available but also how it is organized and used i. e. which constraints are given by the underlying operating system and/or the hardware architecture itself. On a PC system for example there is a memory "hole" which can not be used as normal RAM but is used for accessing the bus and the system's BIOS. This hole starts at address *0x000A0000* (640 KiB) and ends at *0x00100000* (1 MiB), a compatible video adapter's video memory could be addressed starting from *0x000B8000* for example.

When a system has more than 1 MiB of RAM installed this memory hole therefore divides the RAM into two memory areas. While this is not a problem per se, this placing in the middle of the available address space can be suboptimal under certain conditions e. g. when searching a contiguous free memory block. However, this can not be changed as it is an architectural (and historical) limitation. The remaining option is to exploit this restriction. This is what Linux tries to do in its original implementation. As the kernel image has to be placed somewhere in memory, the default configuration is to place it starting at *0x00100000*. As a result the kernel image and the memory hole forms a contiguous reserved memory block so the kernel image's code section, which

will usually resist statically in memory, does not make the situation worse by opening a second hole.

Because of this, an original Linux system requires at least 2 MiB of physical RAM installed or booting is denied before decompressing the kernel image into memory. This limitation makes only sense if the finally location of the kernel image is kept. To potentially allow embedded systems with less than 2 MiB of RAM the memory size check is disabled, relying on the developer's involvement. This is to carefully choose the kernel configuration value *CONFIG_PHYSICAL_START* which contains the physical starting address of the uncompressed kernel image. Leaving this value on its default value *0x100000* locates the image starting at the first mebibyte. Of course the target system should have sufficient RAM installed to house the uncompressed image. If the target device has only one mebibyte or even less of RAM, the value must be decreased. But the new starting address has to be chosen carefully: as the image is compressed by default, both the uncompressed image and the compressed one must fit into the available memory as the decompression mechanism does not support in-place decompression. On systems with an extreme low count of RAM, the abandonment of the image compression should be considered (cf. section 3.1.2).

Both approaches, using compression respectively the use of a non-compressed image, have in common that the physical start address is a hard coded value in the kernel image. So the kernel image requires to be placed at the address or a recompilation with a changed configuration value is needed. This will be at least true until kernel version 2.6.22 is released.[2] In this version experimental support for relocating the kernel at runtime will be introduced by keeping the necessary relocation information in the final image. So a recompilation will not be needed anymore but a tiny relocation function adopts the offset addresses at runtime. However, the already available documentation states that this functionality increases the final image size by around 10 percent. So it depends again on the developer choice and/or the target system resources if the use of this new feature is favorable.

Obviously, the actual memory layout depends on multiple factors. The most important one is the actual installed amount of RAM which is physically present in the system. As pointed out above, the information if more than 2 MiB are available should be known already at compile time. At runtime, the

[2] The current LRTAI implementation bases on kernel version 2.6.17.

```
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009fc00 (usable)
 BIOS-e820: 000000000009fc00 - 00000000000a0000 (reserved)
 BIOS-e820: 00000000000e0000 - 0000000000100000 (reserved)
 BIOS-e820: 0000000000100000 - 00000000003f0000 (usable)
 BIOS-e820: 00000000003f0000 - 00000000003ff000 (ACPI data)
 BIOS-e820: 00000000003ff000 - 0000000000400000 (ACPI NVS)
 BIOS-e820: 00000000fffc0000 - 0000000100000000 (reserved)
```

Figure 4.3: Example of a memory map provided by the BIOS for a system with 4 MiB RAM installed.

major factor is the BIOS which provides information about the current systems configuration.

This is used by the kernel while requesting a memory map from the BIOS before switching the CPU into protected mode. This map is a simple list of usable and/or reserved memory areas in the system. An example of such a map is shown in Figure 4.3. Important memory points and the therefore resulting memory layout of the real-mode kernel part should be evident from Figure 4.2. Beside the flow of execution it shows where the kernel image respectively parts of are located in memory. A standard boot process (i. e. via a bootloader) on a system with more than 1 MiB of RAM was token as a basis.

After switching into protected mode and remaining at its final location, the kernel is able to generate a list of usable RAM areas, founding on the received memory map and its built-in knowledge of its own location. The last important step in the memory management is to free the areas which are marked as *__init* sections. This is done after the major initializations routines have finished and the code and data is not needed anymore. By simply marking the relevant pages as free, they can be utilized by the bootmem allocator for subsequent memory requests.

4.4 Problems

Even though the chosen approach of this work is straightforward, while implementing some problems appeared which had to be solved.

As described earlier the resulting LRTAI system does not require many of the features, drivers or subsystems which come with Linux by default. To drop

them from the project, there were two possibilities. The first naive option is to physically delete the concerned files and directories from the filesystem. But this approach failed due to the static references in the makefiles which were not updated. It resulted in a source tree which did not even build an original Linux kernel at all which was not considered as a good starting point.

A further variant is to solely drop the addressed references in the makefiles. This would result in reviewing every single makefile, around 500 which are potential candidates for the IA-32 architecture. It should be remembered that one of the goals was to minimize the needed changes to the original Linux source tree to retain maintainability and to ease the porting to a newer kernel version if needed. So this variant was rejected in the main too and only for some hand-picked makefiles chosen as pointed out later.

Since the most features can be individually selected or deselected in the kernels configuration step, this approach was the preferred one as it is already built-in and well-supported. So a minimal configuration was created which excludes almost everything which could be deselected. Together with a clean source tree a kernel image was built which has had a size at about 450 KiB.

However it was detected that many code parts were compiled into the kernel which were thought to have been unselected deliberately. By examining the sources the major cause could be determined: starting with Linux kernel version 2.6, the kernel is equipped with the *kobject* infrastructure. As this system maintains a close relationship with the sysfs filesystem it was originated that it would be disabled by not using the sysfs filesystem. Many source files enclose the relevant code parts with preprocessing directives which ignore the code if sysfs is not going to be used. However, there are many source files which does not do this. For LRTAI this files were patched with simple preprocessing instructions to statically exclude these lines of code. As this is a straightforward solution it should be reverted later. The goal should be to supply a clean patch for the Linux kernel which could be included in the kernel's upstream.

As the core kernel subsystems can not be deselected during kernel configuration a few makefiles have to be touched. But simply dropping the files from the makefiles would result also in a lot of undefined references as there (temporarily) may still exist code parts during development which uses the source module's functions and/or variables. A systematic, iterative method was chosen to solve this problem. References to files which could be directly identified as not being used or necessary are dropped immediately. All other files were reviewed by hand and the exported functions were replaced with a stub. So

it was possible to compile the project at every time. The goal was to restore the files later with the original code. When all references to a file were eliminated then it could be finally dropped from the makefile. For files which export needed variables or functions it has to be determined if preprocessing instructions were used to exclude unused code or to extract the used code to a new file. The later approach is not covered by the development's goals but used as a temporary solution. The final result should be patch for a specific kernel version which excludes all unused code segments. Such a patch could be applied easily to newer kernel source trees.

Another problem occurred while porting RTAI's configuration system. Usually, the kernel build system differentiates cleanly between flags being defined or not and integer values. Not so RTAI. Many source files uses the directives directly in statements which resulted in some compile errors after integrating the config system. The reason for this was that Linux' build system undefines non-selected items whereas RTAI's system defines the items with a zero value. So for some items a workaround was implemented in the *rtai_config.h* which redefines the flags accordingly. However, for a cleaner implementation this should be fixed in RTAI's upstream.

Chapter 5

Measurements

5.1 Test system

As a test system a somewhat ancient PC got into action. As LRTAI has (still) limited peripheral support and no access to filesystems is available, a method had to be found to transfer the measured data from LRTAI to a host system. As a working solution the serial console was used. It has to be activated via kernel command line parameter e. g. *earlyprintk=ttyS0*. On the host system the input was redirected into a file from where the data was extracted later. Therefore the measured raw data was prefixed with a short tag to allow easier distinguishing from normal kernel messages. Due to the lack of a present serial port on newer systems, the mentioned "ancient" PC system was used. However, in the measurements the absolute values are of minor interest, the relative ones have to be interpreted.

The used system consists of the following parameters:

- AMD® AMD-K6® CPU, running at 200 MHz
- 128 MiB RAM installed, but only 8 MiB used by LRTAI
- Linux kernel version 2.6.17
- ADEOS IPIPE version 1.3-08
- RTAI version 3.4
- Linux timer interrupt is running at 100 Hz

These parameters are kept constant for all measurements on a LRTAI system and a normal Linux/RTAI installation. For the later, a Debian Sarge system was chosen as a base where the kernel was replaced with an accordingly patched kernel.

5.2 Scheduling latencies

To prove the thesis of chapter 3 that the scheduling latencies of the new LRTAI system should not be worse than that of an established Linux/RTAI system, some measurements are done. Therefore the latency measurement module found in RTAI's tarball was slightly adopted to fit for both environments. This was necessary a it is split into two parts, a kernel mode task and a user-space process. While the kernel module calculates the actual data, the user-space counterpart reads the measured data from a FIFO. This concept could not be used as in LRTAI the user-space has gone. Therefore, the data is simply output by the kernel module via *rt_printk*.

The mentioned module sets up a periodic task which calculates the difference between the expected and the actual activation time. The default period of 100 μs was used. In each measurement 250,000 values are collected. On the Linux/RTAI system the system was stressed with I/O and CPU load. This could be achieved by running several flood pings and intensive hard disk access. For CPU utilization the tool cpuburn was used. On the LRTAI system only CPU load could be generated. For this the initialization routine finalizes with an endless loop and not with a call to *cpu_idle*. The results are plotted in Figures 5.1–5.4.

Figure 5.1 shows the measured latencies on the Linux/RTAI system using the oneshot timer mode. While the major part of the values is around 20000 ns, the maximum measured latency is with 44419 ns more as twice as long. In Figure 5.2 which illustrates the same measurement on a LRTAI system it can be noticed that the average latency has minimal improved in comparison to the Linux/RTAI system. This is due to the fact that on the LRTAI system no further interrupts need to be processed as in the Linux/RTAI system. Also the values show a lower jitter.

The remaining Figures 5.3–5.4 shows the measured values using the periodic mode of the schedulers. Here too, the latency of LRTAI is minimal better than the one of the Linux/RTAI system. The jitter could also be improved. The

negative times prove that the scheduler do some kind of calibration trying to minimize the jitter of real-time tasks. However, in some cases the time delays are less than expected and the real-time task is activated before the requested time. This behavior can be optimized by presetting the calibration manually via kernel command line or by direct compiling in the target systems values.

The result of these measurements are that the LRTAI system behaves nearly like a traditional Linux/RTAI system. This means that its efficiency is quite as good and the system can therefore be used as an alternative solution.

5.3 Image size and memory footprint

The values listed in this section are only for completeness. A comparison between a Linux/RTAI and a LRTAI would not be meaningful because of the oppositional design goals.

The kernel image size of the generated *zImage* is about 122 KiB. The corresponding memory footprint is about 370 KiB thus it is possible to run the kernel on systems with less than 1 MiB of RAM installed. With continuing the work it should be possible to lower this bound further.

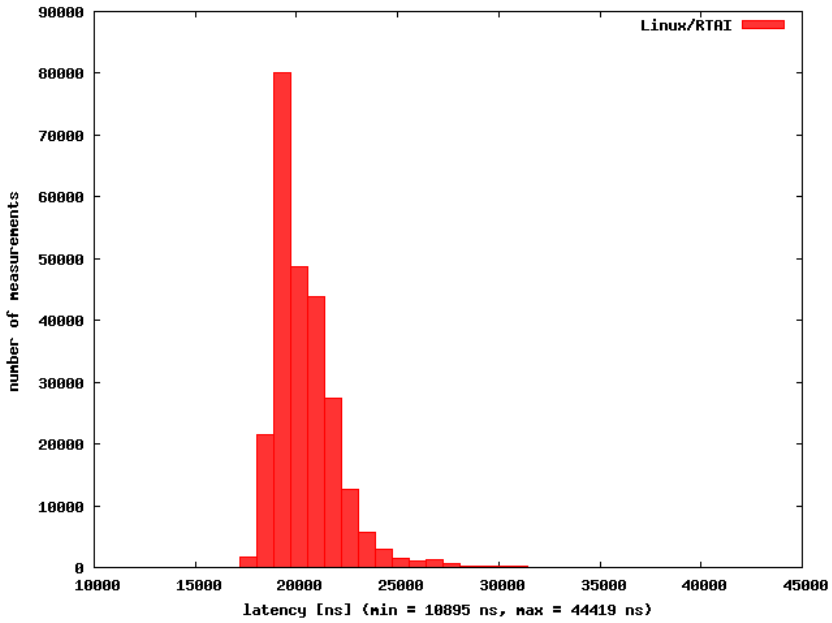

Figure 5.1: Latency of Linux/RTAI in oneshot mode.

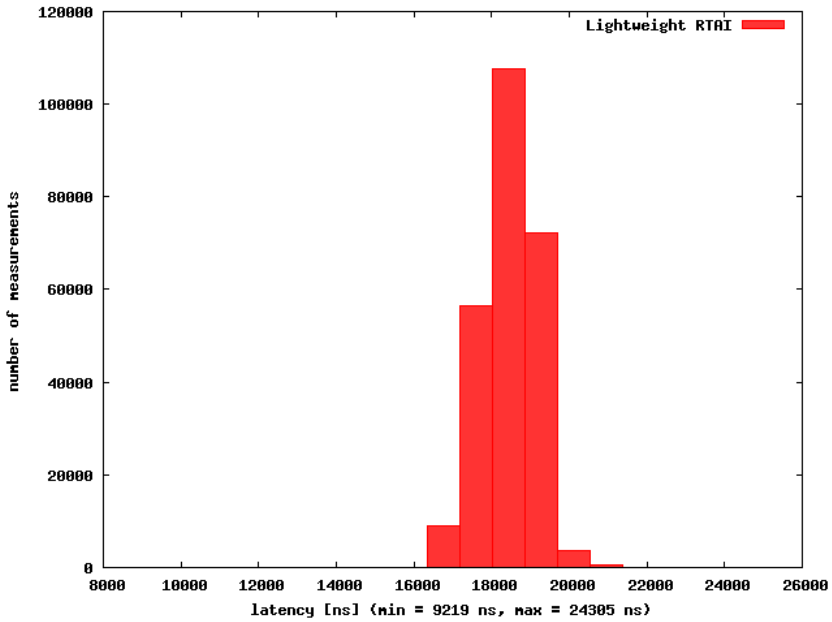

Figure 5.2: Latency of Lightweight RTAI in oneshot mode.

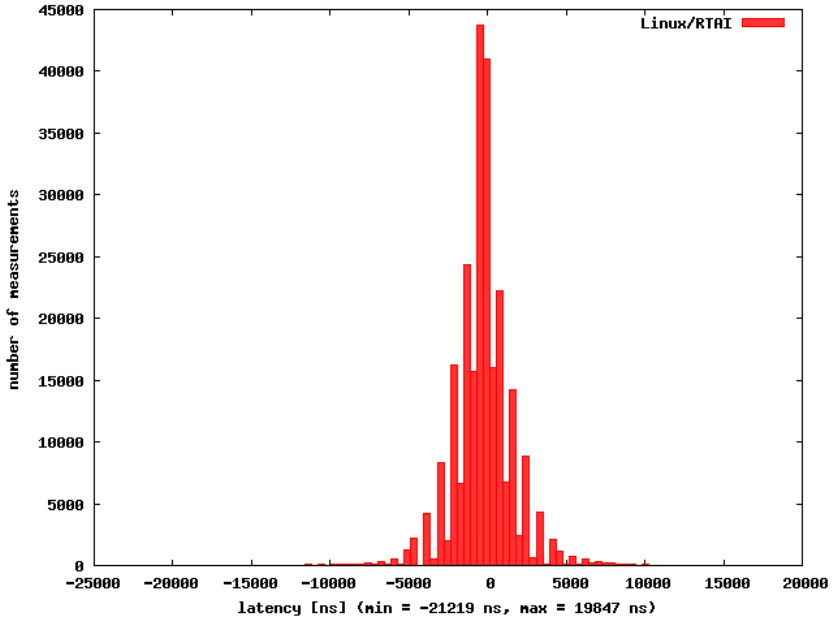

Figure 5.3: Latency of Linux/RTAI in periodic mode.

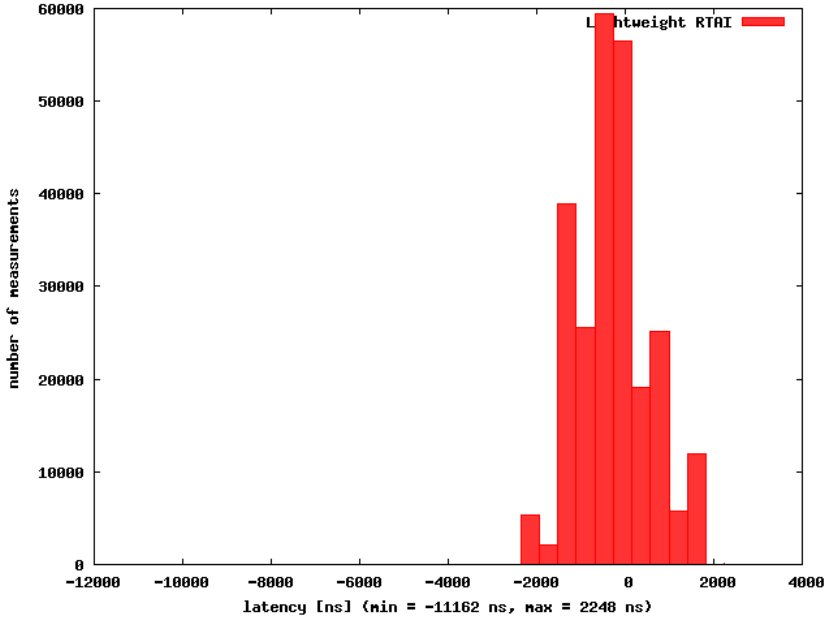

Figure 5.4: Latency of Lightweight RTAI in periodic mode.

Chapter 6

Conclusions

In this thesis a lightweight porting of the Real Time Application Interface API to a bare machine was presented. The work started from an established Linux/RTAI system and reduced the system to a minimal set of Linux' core functions which are strictly required by the original RTAI implementation. Therefore the Linux/RTAI symbiosis was analyzed both at compile time and at runtime. Important dependencies were pointed out. With the compiled knowledge and some design goals the actual implementation was started. The core modules and some additional IPC modules of the RTAI distribution were statically merged with the original Linux kernel. On the other hand, subsystems which are usually included in Linux' default distribution but now are unused or unwanted could be excluded successfully. Finally some measurements are done to prove that the newly created system is an equal replacement of traditional Linux/RTAI compositions. With its reduced memory footprint LRTAI is predestined for systems with restricted resources.

The resulting LRTAI system is not yet optimal. There are still some dependencies of RTAI's scheduler implementation which still requires access to some Linux code paths. This is because the scheduler is capable of not only handling RTAI's proper tasks but it is also able to "steal" and return tasks to Linux' scheduler. Therefore Linux' internal implementations are called directly to achieve the desired functionality. This could be improved by a continued work.

In spite of the code which is still required by the scheduler and thus still enlarges the whole system, the already achieved image size and the size of the memory footprint proves the feasibility of the project and suggests its continuing. Also a continued work could further reduce the changes introduced in the build system. Likewise new features could be implemented. For example

support for APICs could be introduced which would be the base for symmetric multiprocessing.

Appendix A

Sample implementation for using a private heap

Listing A.1: properheap.c

```
1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <rtai.h>
4  #include <rtai_config.h>
5  #include <rtai_malloc.h>
6
7  MODULE_DESCRIPTION("Sample␣application␣which␣allocates␣a␣private␣heap");
8  MODULE_AUTHOR("Michael␣Heimpold␣<michael.heimpold@s2000.tu-chemnitz.de>");
9  MODULE_LICENSE("GPLv2");
10
11 #define HEAP_SIZE (512 * 1024)
12
13 rtheap_t heap;
14 void *heapaddr = NULL;
15
16 static int __init heap_init(void)
17 {
18         if (!(heapaddr = kmalloc(HEAP_SIZE, GFP_KERNEL))) {
19                 printk("myapp:␣kmalloc␣failed.\n");
20                 return 1;
21         }
22         printk("myapp:␣heap␣inited.\n");
23         return 0;
24 }
25
26 #ifdef CONFIG_LRTAI
27 fs_initcall(heap_init);
28 #endif
29
30 static int __exit heap_exit(void)
31 {
32         /* Either use rtheap_destroy or kfree; never both! */
33         if (heapaddr)
34                 kfree(heapaddr);
35         return 0;
```

```
36 }
37
38 /*
39  Rest of module implementation
40  */
41
42 static int __init myapp_init(void)
43 {
44         /*
45            ...
46           */
47         int rv;
48 #ifndef CONFIG_LRTAI
49         if (!heap_init()) {
50                 printk("myapp:␣could␣not␣init␣heap.\n");
51                 return 1;
52         }
53 #endif
54         if (heapaddr
55             && (rv = rtheap_init(&heap, heapaddr, HEAP_SIZE, PAGE_SIZE))) {
56                 printk("myapp:␣rtheap_init␣failed␣with␣%d.\n", rv);
57                 kfree(heapaddr);
58                 return 1;
59         }
60         /*
61            Rest of initialization
62           */
63         printk("myapp:␣loaded.\n");
64         return 0;
65 }
66
67 static void __exit myapp_exit(void)
68 {
69         /*
70            Rest of destructor
71           */
72         if (!heap_exit()) {
73                 printk("myapp:␣could␣not␣destroy␣heap.\n");
74         }
75         printk("myapp:␣unloaded.\n");
76 }
77
78 module_init(myapp_init);
79 module_exit(myapp_exit);
```

Appendix B

Building the LRTAI kernel image

The tarball with the source code of this work is stored on the attached data medium or can be obtained via Internet at [21].

To build the LRTAI kernel image, extract the tarball to the */usr/src* directory on your Linux workstation. If you use this suggested location, you do not need to update the included makefiles, otherwise you have to adopt the variables *KERNELSRC* and/or *KERNELOUTPUT* in the build directory's top-level *Makefile*. After this, just changing into the build directory (*/usr/src/lrtai-build* per recommendation) and running *make* builds the image.

If changes to the LRTAI system are needed, a *make menuconfig* step gives a graphical user interface for configuration. Changes in the Linux kernel originated part are not yet recommended, in RTAI's part some presets can be changed. Also IPC modules can be chosen.

It is assumed that the build is done as user *root*. It was not tested but it should be possible to run this as a unprivileged user. Use appropriated tools like *fakeroot* if necessary.

```
# extract tarball to /usr/src
buildsys:~# tar -C /usr/src -xvjf ~/lrtai-0.1.tar.bz2

# configure LRTAI
# (neither necessary nor recommended for the initial release)
buildsys:~# cd /usr/src/lrtai-0.1/build && make menuconfig

# build the zImage
buildsys:~# cd /usr/src/lrtai-0.1/build && make
```

Figure B.1: Transcript of building the LRTAI kernel image.

Appendix C

GnuPG signature of the LRTAI tarball

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBGeXU9KGO9ZzVRhqoRAt3GAJ9r3BhOkFt5Wj+d1oUoZG80KbZ2ggCZATV1
OJrWAkldt40AQZpL7qlMyc=
=wbIg
-----END PGP SIGNATURE-----
```

Appendix D

Copyright notice

For files of the original Linux kernel or of the RTAI distribution the licenses apply which were distributed with the particular package or file. Modifications of such files are usually covered by the same license, see the included license documents for details.

For all other files which were created by this work and does not contain an explicit copyright notice and/or license term the following applies:

Nomenclature

ADEOS ..... Adaptive Domain Environment for Operating Systems

API ..... Application Programming Interface

APIC ..... Advanced Programmable Interrupt Controller

BIOS ..... Basic Input/Output System

BKL ..... Big Kernel Lock

BSS ..... Block Started by Symbol

CPU ..... Central Processing Unit

DSP ..... Digital Signal Processor

ELF ..... Executable and Linking Format

FPU ..... Floating point unit

GNU ..... GNU is Not Unix

GPL ..... GNU General Public License

GPOS ..... Gerneral-Purpose Operating System

GRUB ..... GRand Unified Bootloader

GUI ..... Graphical User Interface

HAL ..... Hardware Abstraction Layer

I/O ..... Input/Output

IPC ..... Inter-Process Communication

IPIPE ..... Interrupt PIPEline

IRQ ..... Interrupt request

ISA ..... Industry Standard Architecture

LRTAI ..... Lightweight RTAI

PC ..... Personal Computer

PIT ..... Programmable Interrupt Timer

RAM ..... Random Access Memory

RTAI ..... Real-Time Application Interface

RTOS ..... Real-Time Operating System

SA-RTL ..... Stand-Alone RTLinux

SMP ..... Symmetric Multiprocessing

USB ..... Universal Serial Bus

VGA ..... Video Graphics Array

References

[1] Jens Kretzschmar. *Implementing RTAI on a DSP without Linux*. Diploma thesis, Chemnitz University of Technology, 2005.
`http://rtg.informatik.tu-chemnitz.de/docs/da-sa-txt/da-krej.pdf`

[2] Michael Luft. *Completing and Testing Lightweight RTAI/C6x*. Seminar paper, Chemnitz University of Technology, June 2006.
`http://rtg.informatik.tu-chemnitz.de/docs/da-sa-txt/sa-luft.pdf`

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 2005. ISBN: 0-59600-565-2

[4] Robert Love. *Linux Kernel Development, Second Edition*. Novell Press, 2005. ISBN: 0-67232-720-1

[5] Homepage of MontaVista Software, Inc.
`http://www.mvista.com/`
(June 21, 2007)

[6] The Preemption Patches at Robert Love's kernel.org space.
`http://www.kernel.org/pub/linux/kernel/people/rml/preempt-kernel/`
(June 21, 2007)

[7] Clark Williams. *Linux Scheduler Latency*. Red Hat, Inc, March 2002.
`http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf`
(June 21, 2007)

[8] Linux Kernel Organization. *The Linux Kernel Archives*.
`http://www.kernel.org/`
(June 21, 2007)

[9] Victor Yodaiken and Michael Barabanov. *A Real-Time Linux*. New Mexico Institute of Technology, 1996/1997.
`ftp://luz.cs.nmt.edu/pub/rtlinux/papers/usenix.ps.gz` (Offline)
`http://citeseer.ist.psu.edu/6239.html`
(June 21, 2007)

[10] Paolo Mantegazza. *DIAPM RTAI for Linux: WHYs, WHATs and HOWs*. Real Time Linux Workshop at Vienna University of Technology, December 1999.
`https://www.rtai.org/index.php?module=documents&JAS_DocumentManager_op=downloadFile&JAS_File_id=31`
(June 21, 2007)

[11] Markus Franke. *A Quantitative Comparison of Realtime Linux Solutions*. Seminar paper, Chemnitz University of Technology, March 5, 2007.
`http://rtg.informatik.tu-chemnitz.de/docs/da-sa-txt/sa-franm.pdf`

[12] The ADEOS Project.
`http://home.gna.org/adeos/`
(June 21, 2007)

[13] Marshall K. McKusick and Michael J. Karels. *Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel*. In: Proceedings of the San Francisco USENIX Conference, pp. 295–303, June 1998.
`http://docs.FreeBSD.org/44doc/papers/kernmalloc.pdf`

[14] Giovanni Racciu and Paolo Mantegazza. *RTAI 3.4 User Manual rev 0.3*.
`https://www.rtai.org/index.php?module=documents&JAS_DocumentManager_op=viewDocument&JAS_Document_id=44`
(June 21, 2007)

[15] Peter Miller. *Recursive Make Considered Harmful*. AUUGN Journal of AUUG Inc, 19(1), pp. 14–25.
`http://aegis.sourceforge.net/auug97.pdf`

[16] Vicente Esteve, Ismael Ripoll and Alfons Crespo. *Stand-Alone RTLinux-GPL*. Universidad Politcnica de Valencia, October 20, 2003.
`http://www.rtlinux-gpl.org/~vesteve/docs/ws2003.pdf`

[17] Miguel Masmano, Apolinar González, Ismael Ripoll and Alfons Crespo. *Embedded RTLinux: A New Stand-Alone RTLinux Approach*. Eighth Real-Time Linux Workshop at Lanzhou University - SISE, China, October 2006.
`ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2006/paper_07.pdf`

[18] Jean-loup Gailly and Mark Adler. *zlib Home Site*.
`http://www.zlib.net/`
(June 21, 2007)

[19] Intel Corporation. *Intel Pentium Processor - Invalid Instruction Erratum Overview*.
`http://support.intel.com/support/processors/pentium/ppiie/index.htm`
(June 21, 2007)

[20] Jeff Bonwick. *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, 1994.
`http://www.usenix.org/publications/library/proceedings/bos94/full_papers/bonwick.ps`
(June 21, 2007)

[21] Homepage of Michael Heimpold.
`http://www.heimpold.de/`
(June 21, 2007)

TECHNISCHE UNIVERSITÄT CHEMNITZ

Zentrales Prüfungsamt

Eidesstattliche Erklärung*

Name: **Heimpold** Vorname: **Michael** geb. am: **15.05.1981** Matr.-Nr.: **24902**	**Bitte Ausfüllhinweise beachten:** 1. Nur Block- oder Maschinenschrift verwenden.

Ich erkläre an Eides statt, gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende **Diplomarbeit** selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht.

Datum: **21.06.2007**

Unterschrift: Michael Heimpold
Antragsteller

* Diese Erklärung ist der eigenständig erstellten Arbeit als Anhang beizufügen. Arbeiten ohne diese Erklärung werden nicht angenommen. Auf die strafrechtliche Relevanz einer falschen Eidesstattlichen Erklärung wird hiermit hingewiesen.