Magisterarbeit

A Partitioning Operating System based on RTAI-LXRT Linux

Ausgeführt am Institut für
Technische Informatik
der Technischen Universität Wien

unter der Anleitung von

O. Univ. Prof. Dr. phil. Hermann Kopetz
und
Univ. Ass. Dr. techn. Roman Obermaisser
als verantwortlich mitwirkendem Universitätsassistenten

durch

Bernhard M. J. Leiner, Bakk. techn.
Geblergasse 24-26/1/10, A-1170 Wien

Wien, im Dezember 2006 .....

Abstract

Dependable embedded systems are an indispensable technology in today's information society. Especially since more and more safety-critical applications are depending on such embedded systems. There are various reasons to replace conventional, mechanical systems: embedded systems offer more flexibility and are often needed to make new applications possible at all.

One of the main challanges of the wide spread usage of such systems is the increasing complexity which makes it very difficult to reach the necessary dependability. Furthermore, production and maintenance costs play an increasing role as soon as the deployment reaches large numbers.

Integrated architectures, building execution platforms from a set of prevalidated hardware and software components, promise a solution for the stated problems. The DECOS (Dependable Embedded Components and Systems) project, funded by the European Union, aims at developing basic technologies for the introduction of such integrated architectures.

One of the core technologies are partitioning operating systems which are able to execute multiple software components on a single processor while guaranteeing that these components cannot influence each other. This holds for both, the temporal and the spatial domain.

This thesis shows a possible implementation of such a partitioning operating system on top of a RTAI (Real Time Application Interface) real-time Linux system. An existing prototype implementation, done for the DECOS project, is evaluated regarding partitioning capabilities. To collect information about the temporal behaviour, a monitoring framework that periodically sends timestamps via a network to a monitoring server, is implemented. Additionally, checksums calculated over multiple memory areas are used to observe violations of spatial partitioning.

Running multiple test cases identifies some holes in the partitioning, mainly resulting from the availability of to much functionality in the interface provided to the partition tasks. A kernel module is developed that cuts down this functionality. When loaded, partition tasks are not able to use Linux system calls any more and are restricted to a small subset of RTAI functions. This subset is provided to them in form of a new interface that also introduces a mandatory signature parameter in each function call to provide basic authentication of jobs claiming ownership of a resource.

Rerunning the test cases that causes a violation of partitioning in the original setup shows the improvements with reference to partitioning in the adapted execution environment.

Zusammenfassung

Zuverlässige elektronische Systeme sind aus der heutigen Gesellschaft nicht mehr weg zu denken. Insbesondere, da sich auch immer mehr sicherheitskritische Anwendungen auf die Funktionalität von eingebauten Mikroprozessoren — so genannten *embedded systems* — verlassen. Die Gründe für die Abkehr von konventionellen und bewährten mechanischen Systemen sind vielfältig: Computergesteuerte Prozesse bieten mehr Flexibilität und machen viele neue Anwendungen erst möglich.

Eine der größten Herausforderungen bei dieser immer weiter gehenden Verbreitung ist die zunehmende Komplexität solcher Lösungen, die es schwierig macht, die benötigte Zuverlässigkeit zu erreichen. Des weiteren spielen Kosten, sowohl in der Fertigung als auch für die Wartung, mit zunehmenden Stückzahlen eine immer größere Rolle.

Integrierte Systemarchitekturen, die eine Plattform zusammengesetzt aus validierten Hardware und Softwarekomponenten bieten, gelten als eine mögliche Lösung für die oben genannten Herausforderungen. Das DECOS (Dependable Embedded Components and Systems) Projekt, gesponsert von der europäischen Union, zielt darauf ab, grundlegende Technologien für die Einführung solcher integrierter Architekturen zu entwickeln.

Eine solche Technologie sind partitionierende Betriebssysteme, die die Ausführung von mehreren Softwarekomponenten auf einem Prozessor ermöglichen und dabei garantieren, dass keine gegenseitige Beeinflussung zwischen diesen Komponenten stattfindet. Dies gilt sowohl für die zeitliche, als auch für die räumliche Dimension.

Diese Arbeit zeigt eine mögliche Implementierung eines solchen partitionierenden Betriebssystem auf Basis eines RTAI (Real Time Application Interface) Echtzeit Linux Systems. Eine existierende Prototyp-Implementierung, durchgeführt im Rahmen des DECOS Projektes, wird auf Schwächen hinsichtlich Partitionierung untersucht. Um Informationen über das zeitliche Verhalten zu erhalten wird ein Kontrollsystem entwickelt, dass periodisch Zeitstempel über ein Netzwerk an einen Kontrollserver sendet. Zusätzlich werden Prüfsummen über verschiedene Speicherbereiche berechnet um eine Verletzung der räumlichen Partitionierung festzustellen.

Das Ausführeren mehrerer Testfälle zeigt einige Lücken in der Partitionierung auf. Größtenteils werden diese durch die zu großzügige Verfügbarkeit von Funktionalität in der Schnittstelle für die Partitionsprozesse verursacht. Um diese Funktionalität einzuschränken wird ein Linux-Kernel Modul entwickelt. Sobald dieses geladen ist, haben Partitionsprozesse

keinen Zugang zu Linux Systemaufrufen mehr und sind außerdem auf eine kleine Teilmenge der RTAI Funktionen beschränkt. Diese Teilmenge wird ihnen durch eine neue Schnittstelle zur Verfügung gestellt die weiters einen zusätzlichen Signatur-Parameter für alle Funktionsaufrufe verlangt. Dieser wird verwendet um eine einfache Authentifizierung von Prozessen durchzuführen die auf eine Ressource zugreifen.

Eine Wiederholung der Testfälle, die in der ursprünglichen Konfiguration eine Verletzung der Partitionierung verursacht haben, zeigt die Fortschritte die bezüglich Partitionierung in der adaptierten Ausführungsumgebung erzielt wurden.

Contents

List of Figures

List of Tables

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing (1912–1954)

Chapter 1.

Introduction

DEPENDABLE EMBEDDED SYSTEMS are an indispensable technology in todays information society. Especially since more and more safety-critical applications are depending on such embedded systems. There are various reasons to replace conventional, mechanical systems: Embedded systems offer more flexibility and are often needed to make new applications possible at all.

One of the main problems of the wide spread use of such systems is the increasing complexity which makes it very difficult to reach sufficient dependability and maintainability at feasible costs. The DECOS (Dependable Embedded Components and Systems) project is an integrated project, funded by the European Union under framework programme 6, that aims at developing an architecture for dependable systems while reducing development, production and maintenance costs at the same time.

The way to achieve this goal is to switch from traditional *federated systems*, that use dedicated hardware components and communication networks for each function, to a *component-based, integrated system* [KOPS04]. In such an architecture different functions share hardware resources and communicate with each other via a homogeneous network. A lower overall system *complexity* along with reduced *maintenance costs*, while at the same time providing an increased level of *dependability*, are the expected outcome of this solution.

Integrating multiple independent software components of mixed criticality levels on a distributed system poses a couple of challenges. *Computational* and *communication resources* have to be separated among the components in such a way that they cannot influence each other. For the communication resources, a time triggered communication approach like used in the DECOS project is able to ensure such a separation[OPK05]. For sharing computational resources on a single micro controller, an operating system, that is able to guarantee, that each of the components is executed without being influenced by the other components, is needed. The hardware resources are virtually divided into *partitions* and each software component shall use only resources within its partition, both in the *temporal* and in the *spatial* domain. Consequently, an operating system achieving this is called a *partitioning operating*

system.

For the DECOS project, a prototype cluster has been built, that features an implementation of such a partitioning operating system based on a real-time Linux kernel. The RTAI real-time Linux variant was chosen because its LXRT extension allows to implement hard real-time tasks in user space. On hardware platforms including a memory management unit, user space Linux processes have separate virtual address spaces and therefore a spatial partitioning of the main memory exists. Temporal partitioning is done by using a time-triggered dispatcher, running at highest priority, which enforces a static scheduling of the partitions.

This thesis evaluates the partitioning capabilities of the existing implementation. It is shown that, although memory protection provides basic spatial partitioning and the static scheduling approach aims at temporal partitioning, a couple of problems exists. Those problems can be broken down into three areas: (1) RTAI offers a naming service to share object references among different tasks. This naming service does not enforce authentication. (2) Partitions have full access to all LXRT calls. (3) Partitions can use all Linux system calls.

As part of this thesis, a kernel module has been implemented that deals with these problems and provides an improved API to the partitions. This API only contains function calls which are known to have no adverse influence on partitioning and whose parameters are strictly checked to ensure that no partition has access to private resources of another partition. All other LXRT functions or Linux system calls are disabled by this module.

1.1. Structure of the Thesis

After this introduction, Chapter 2 introduces basic concepts and related work. Among those explained concepts are *distributed real-time systems* and *real-time communication* as *dependability* as a core feature of a *safety-critical* system. *Real-time operating systems* with a focus on implementations based on the Linux kernel are next, followed by a short overview about more related work discussing *partitioning* and *virtualization*.

Chapter 3 introduces the *DECOS integrated architecture*. Besides the hardware structure, the software setup of a DECOS node is specified as well. An important part of the DECOS software stack is the *core operating system*, a *partitioning OS*, with properties as discussed in Chapter 4.

A prototype implementation of the core OS, based on RTAI-LXRT Linux, is described in Chapter 5. The rest of this chapter concentrates on having a close look at the partitioning capabilities of such an implementation. This includes an analysis of RTAI-LXRT resulting in several *hypotheses* about this issue. Those hypotheses form the theoretical basis for the implementation of several *test cases*. A *monitoring framework* to gather informations about the task execution via LAN is described in

the chapter too. The end of this chapter shows how the problems that have been identified during the testing stage, can be solved by *restricting the available task API*.

The conclusion in Chapter 7 looks at the achieved results and presents some issues for *future work* which have not been addressed in this thesis.

I love deadlines. I like the whooshing sound they make as they fly by.

Douglas Adams (1952–2001)

Chapter 2.

Basic Concepts and Related Work

THIS CHAPTER introduces the basic concepts that are necessary for understanding this thesis. It will introduce the terms *real-time* and *safety critical* systems and summarize the most important concepts in the field of *dependability*. Besides that, it will give an overview about *real-time operating systems* and further *related work*, such as real-time Linux variants, *partitioning operating systems* and *virtualization*.

2.1. Safety-Critical Real-Time Systems

> A *real-time computer system* is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced. [Kop97, p. 2]

People new to the field of real-time computing frequently make the mistake to misinterpret this definition. They tend to belief, that a fast and powerful computer system is needed in order to achieve real-time capability. But in fact, *real-time computing is not fast computing* [Sta88]. It is only important to be fast enough to meet all deadlines even under worst-case conditions like peak loads. A "slow" embedded 8 bit microcomputer can be part of a real-time system whereas a fast and powerful workstation, running a common desktop operating system, is generally not suited for real-time tasks since important tasks might be preempted by background processes or interrupts.

In real-time systems the focus is on guaranteed meeting of deadlines, and not on pure performance (throughput, bandwidth, minimal execution time, ...). In other words, there is the need for *guaranteed response times* even in the case of peak load in contrast to *best effort* systems.

2.1.1. Time

Obviously the concept of *time* plays an important role in the field of real-time computing. Important enough to justify a closer look on it.

The purpose of a clock is keeping track of time. This goal cannot be perfectly met by a physical clock since it is always restricted in several ways. First of all, physical digital clocks have a finite resolution, the interval between two *clock ticks*. The second problem is that the frequency at which the quarz oscillator, which is responsible for generating the clock ticks, runs, is in general not perfect. Thus clocks are running either to slow or to fast compared to the theoretical *perfect clock*. This phenomen is called *clock drifting*.

For a single computer and a single clock, an inaccurate time is usually not a big problem since it's at least internally consistent [TvS02, p. 243]. For a distributed systems the situation gets more complicated. If the different nodes rely on a global time, a clock synchronization mechanism is needed. This synchronization can be done via an *internal distributed algorithm*, an *external time source* like GPS (Global Positioning System) or a combination of this approaches [Kop97, p. 59 – 67].

Hard vs. Soft Real-Time

For real-time systems one can distinguish between *soft* and *hard* real-time systems. Both of them share to basic principle that results must be available at certain *deadlines*. If a result that missed its deadline is still useful to a certain extend, the deadline is called *soft*. Otherwise its a *firm* deadline. Following this definitions, a soft real-time system is allowed to miss its deadlines infrequently. For hard real-time systems on the other side, a catastrophe can occur if a deadline is missed. Therefore *hard real-time systems* are also called *safety-critical real-time systems*.

Real-Time Scheduling

Real-time scheduling deals with the allocation of resources to tasks in such a way that all timing requirements are met. The basic requirement is that all tasks meet their deadline. This field is most probably the most widely researched topic within real-time systems and literally thousands of research papers have been written [Kop97, p. 227].

Most safety-critical real-time systems rely on static scheduling algorithms like *cyclic scheduling* or *rate monotonic scheduling*. The main advantage of this *off-line* scheduling approach is that it allows a complete analysis of the task set. An important number for this form of analysis is the Worst Case Execution Time (WCET) of a task. If it's possible to generate a feasible off-line schedule where all tasks block the resources for their WCET, there should be no deadline misses during operation.

The main disadvantage of the described static scheduling approach is its inflexibility since it does not handle unexpected tasks. Therefore, for systems that cannot eliminate

such sporadic tasks, a *dynamic scheduling algorithm* (also called *on-line* scheduling algorithm) is necessary. Earliest Deadline First (EDF) is such an algorithm and has some very good properties, like simplicity and even optimality under certain conditions [S+98, chap. 3.1].

Nevertheless, in *general* it is not possible to deal with an arbitrary number of sporadic tasks and the resulting overload. As a matter of fact, the prototype implementation of the DECOS architecture that will be introduced in chapter 5 will use a simple and static cyclic scheduling strategy that is generated off-line.

2.1.2. Distributed Real-Time System

Chapter 2 of [Kop97] states as answer to the question, why a distributed solution is a good idea for building safety-critical real-time systems, three important attributes that fit to a distributed solution:

(i) *Composability*

(ii) *Scalability*

(iii) *Dependability*

Composability is an attribute of an architecture that makes it possible to build a bigger system from predefined parts (components). Properties (like timeliness or testability) already established at subsystem level will hold also at system level [Kop97, p. 34]. This makes not only sense for distributed systems but also in general, e.g. for software development. A lot of research had been done in this area, to develop design paradigms[1] and architecture description languages[2], making compositional design of software possible. In fact, compositional design is a necessity to make it possible for the human mind to control increasing complexity [Sim96, chap. 8].

Scalability is also favored by a distributed system since it's usually easier to add additional resources to such an architecture. And combined with the composability attribute, the complexity of the extended system does not increase significantly.

The issue of dependability will be handled in detail later on in Section 2.2. The basic argument is that a distributed approach makes it possible to create well defined fault containment regions and can achieve fault-tolerance by redundant nodes.

Linking Interfaces

Building a distributed system raises the challenge of linking nodes of this system together in a way to preserve the attribute of composability. This raises the question

[1] especially Object Oriented (OO) programming
[2] like the Architecture Analysis and Design Language (AADL)

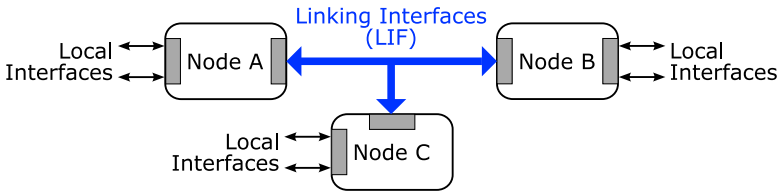


Figure 2.1.: A composition of three components A,B,C [KS03]

of how to link subsystems (or components) together (Figure 2.1) such that properties that have been established at subsystems level will hold at the system level [Kop97, p. 34]. This is done via *Linking Interfaces* (LIFs). They ensure encapsulation of the components, hiding all internal details and local interfaces.

Crucial is, that the LIFs are specified both in the *value* and in the *time domain*. This makes *composition driven strictly by LIF level interactions* achievable. A LIF is characterized by [KS03]:

(a) Its data properties, i.e., the structure and semantics of the data items crossing the interface.

(b) Its temporal properties, i.e., the temporal conditions that have to be satisfied by the interface for both control and data delivery validity.

Real-Time Communication

When linking together several components like shown in Figure 2.1 there is obviously the need for a *communication system*. This communication system has to fulfill some requirements to be suitable for a real-time system. First, the *protocol latency* — the time interval between sending and receiving a message — has to be short enough. A detailed analysis of this timinig requirement can be found in section 2.1.2. Additionally, for safety-critical applications, the communication system has to support *error detection* and if possible *error correction*. Furthermore *end-to-end acknowledgment* is essential to provide a dependable service.

There are several approaches to meet these and some more requirements which results in different protocols. One possibility to distinguish between those different protocols, is their access method on multiple-access buses[3]. For predictable hard real-time, *time-division multiple-access* (TDMA) based protocols [TTT02] and token based approaches [MZ94] are used. If event-triggered, asynchronous messages are required, a *carrier-sense multiple-access* (CSMA) soft real-time protocol like CAN [Bos91] is suitable [MN99].

[3]For wireless communication several other access methods are possible. For example *frequency-division multiplex* (FDMA) or *code-division multiplex access* (CDMA).

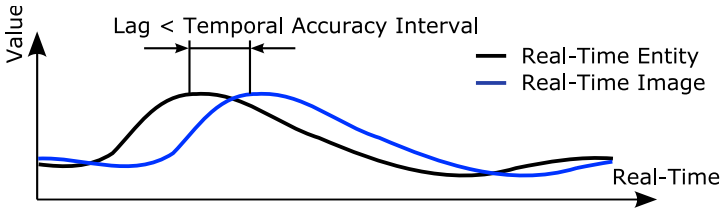


Figure 2.2.: Time lag between real-time entity and real-time image [Kop97, p. 103]

The DECOS architecture that is introduced in chapter 3 is based on a time-triggered core architecture that uses a TDMA based protocol. This core architecture is explained in Section 3.2.1.

Real-Time Entities and Images

A *real-time entity* is a relevant state variable somewhere in the environment or the computer system. A *real-time image* is a temporally accurate picture of a state variable at an instant t [Kop97, chap. 5].

In order to announce the value of this state variable and update its corresponding image, messages containing observation informations have to be sent. An observation is an atomic data structure containing the name of the real-time entity, an observation value and the time of observation t_{obs} [KB03]:

$$\text{Observation } = < \text{Real-Time Entity Name, Observation Value, } t_{obs} >$$

It's now possible to identify two timing characteristics. First, there is a time interval starting from the observation event until the real-time image gets updated. This time interval includes computation time on the sender side, the transmission time of the message over the communication system and computation time for receiving and updating the image on the receiver side. This time delay is also called *lag*.

The second timing aspect is the so called *temporal accuracy interval* d_{acc}. This concept is needed since a real-time image is only useful and valid for a certain time, which is an application-specific parameter of the given real-time entity. It takes into account the dynamic properties of the value of the real-time entity $v(t)$, and the needed precision on the image side. The maximum error (value difference between image and entity) is given by:

$$\text{maximum error} = \max_{\forall t}\left(\frac{dv(t)}{dt} d_{acc}\right)$$

If a real-time image is not updated during d_{acc} it's outdated and not useful any more since the error might already be bigger than expected. So it's crucial that the lag is at most d_{acc} (see also Figure 2.2).

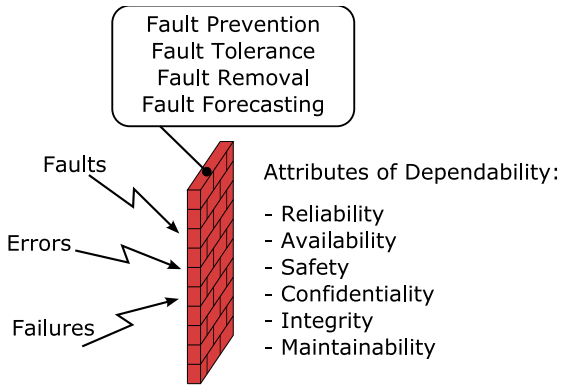


Figure 2.3.: Different means to attain the attributes of dependability under the occurrence of threats.

2.2. Dependability

So far only real-time systems working exactly along its specification have been considered. Real-world systems have to deal with environmental influences which can cause a failure of the system. Additional to this *external* causes for a system failure there can be *internal* ones too. Since — at least for safety critical systems — human life can depend on an operational system, such failures have to be prevented or made very unlikely. *Dependability of a computing system is the ability to deliver service that can justifiably be trusted* [ALR01]. Figure 2.3 illustrates which different techniques are used to protect the attributes of dependability from various threats.

2.2.1. Faults, Errors, and Failures

Faults, errors and failures are the main threats to dependability. Figure 2.4 shows that those terms are not independent from each other but form a fault-error-failure chain.

A *fault* is the cause of an error and can be classified in several ways. When looking at the system boundaries, a fault can be either an external or an internal fault. In its persistence it can be permanent or transient. The fault can happen in hardware or in software and so on. [ALR01] explains all this elementary fault classes and shows how to gather them into three major fault classes for which defenses need to be devised: *design faults*, *physical faults* and *interaction faults*.

A fault can lead to an incorrect internal state of the (sub)system. Such a incorrect system state is called *error*. Like faults, errors can be transient or permanent depending whether an explicit repair action is required to remove the error.

Errors in one part of a system can propagate and cause further internal errors. As soon as such an incorrect state reaches a service interface and cause a deviation from

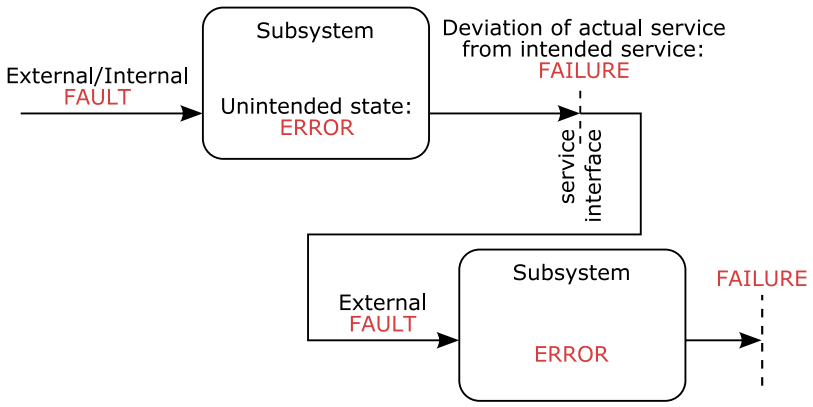


Figure 2.4.: Relationship between faults, errors and failures.

the specification of its intended service, a system *fails*. A *failure* is therefore an *event* that indicated the shift from correct service to incorrect service. The failure of one subsystem can be seen as an external fault for any other subsystem that depends on it, or as an internal fault for a bigger system containing the failing subsystem.

2.2.2. Attributes of Dependability

The attribute of dependability is based on the following basic attributes (see also Figure 2.3): [ALR01]

- **reliability**: Continuity of correct service. The reliability $R(t)$ is the probability that a system will work as expected for the next time duration t presupposed that the system is operational at the moment.

  $$R(t) = e^{-\lambda t}$$

  This equation is correct if the *failure rate* λ (in *failures/hour*) is constant. The inverse of λ is also called Mean-Time-To-Failure (MTTF). MTTF and the according Mean-Time-To-Repair (MTTR) and Mean-Time-Between-Failure (MTBF) are depicted in Figure 2.5.

- **availability**: Readiness for correct service. It is measured as the fraction of time the system is providing its service.

  $$\text{availability A} = \frac{\text{MTTF}}{\text{MMTF} + \text{MTTR}} = \frac{\text{MTTF}}{\text{MTBF}}$$

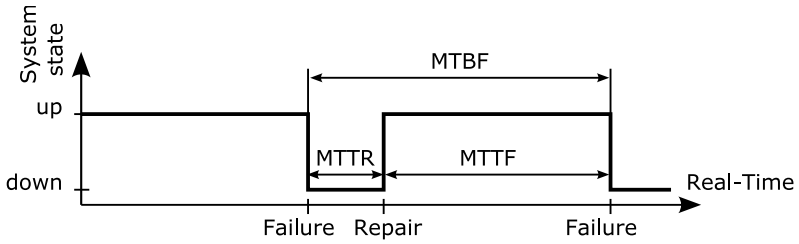


Figure 2.5.: Relationship between MTTF, MTBF and MTTR. [Kop97, p. 11]

A common demand for high-availability systems in the telecommunications industry is an uptime of 99.999 %. This means that the system is allowed to be out of service for only about 5 minutes per year (see also [HHL03]).

- **safety**: Absence of catastrophic consequences on the user(s) and the environment.
- **confidentiality**: Absence of unauthorized disclosure of information.
- **integrity**: Absence of improper system state alterations. Together with the two attributes above, safety and confidentiality, one can introduce the attribute *security*.
- **maintainability**: Ability to undergo repairs and modifications.

Each system has its own dependability requirements and therefore each of these attributes can be more or less important. An example showing this might be a online store of some company versus a plane engine. The webserver running the online store has to have a very high *availability* since every time it's unreachable the company looses money. So, if an uptime of 99.99 % is required, it's ok for the web store to be unreachable for about 10 seconds per day. This means that the system is, in theory, allowed to fail for a second every two or three hours. For a plane engine, on the other side, it's completely unacceptable to fail even for short time during operation. Here a *ultrahigh reliability* and therefore a failure rate λ of about 10^{-9} failures/h or lower is required. The regular maintenance after each flight, necessary to reach this high reliability, obviously prevents a high availability.

2.2.3. Techniques used to Build Dependable Systems

The threats and the attributes of dependability have already been introduced. Now it is time to look a techniques that are used to achieve the attributes of dependability despite the threats like shown in Figure 2.3. [ALR01] describes four means to attain dependability:

(i) **Fault prevention** is mainly quality control during the whole development process of hard- and software. It also includes all kinds of shielding, hardening, rigorous maintenance procedures and defenses to prevent malicious faults.

(ii) **Fault Tolerance** is a concept aiming at providing a service even in the presence of faults. This is generally done by *error detection* and subsequent *system recovery*.

Error detection can be done based on *a priori* knowledge about the system (in the time or value domain) or through *redundant* computation [Kop97, p. 126].

(iii) **Fault Removal**. The two key words here are *validation* and *verification*.

Validation is checking if the specification really describes the real-world problem in a correct and complete way[4]. Verification is done on the actual implementation of the system and checks if this implementation fulfills the specification. Verifying a formal representation of the implementation is called *static verification* and uses techniques like model checking and theorem proving. Verifying the actual system by supplying inputs to the system is called *dynamic verification* or usually just *testing*.

(iv) **Fault Forecasting** is done by performing an evaluation of the system behavior with respect to fault occurrence or activation. This can be done in a *qualitative* or a *quantitative* way.

The qualitative approach tries to identify event combinations that would lead to system failures. Quantitative fault forecasting evaluates the extend to which some of the attributes of dependability are satisfied in terms of probability.

2.2.4. Fault Hypothesis

The fault hypothesis is a statement that describes the faults that a specific fault-tolerant computer system is supposed to handle. It covers the types and the frequency of faults [Kop97, p. 73].

Fault and Error Containment Regions

For all faults stated in the fault hypothesis it should be clear which parts of the system are affected. This is important since faults and the resulting errors might propagate throughout several subsystems like shown in Figure 2.4.

A fault-containment region (FCR) is defined as the set of subsystems that share one or more common resources and may be affected by a single fault [Kop03]. Such shared

[4] A common problem in specifications is that the informal part is written in an ambiguous way. To prevent this it's useful to define the exact meaning of key words and phrases in advance (see also [Bra97]).

resources might be: the computing hardware, power supply, timing source, physical space, etc. Fault containment is achieved by architectural decisions concerning shared resources.

When an error occurs in one FCR it can propagate through message failures (in time or value domain) to another FCR. All FCRs that might become erroneous by such messages belong into one *error-containment region*.

Partitioning

The purpose of partitioning is error containment: a failure in one partition must not propagate to cause failure in another partition [Rus99]. This thesis concentrates on software faults, so a partition is a error-containment region for a software fault. The goal of a partitioning software system can also be expressed in a simpler and more direct way:

> The behavior and performance of software in one partition must be unaffected by the software in other partitions [Rus99].

In considering the design of partitioned systems, it is useful to distinguish two dimensions of partitioning, *spatial* and *temporal*. Spatial partitioning is focused on protecting memory and private devices from other partitions whereas temporal partitions deals with quality of service of shared resources.

A more detailed view on providing multiple partitions on a single processor computer is given in Chapter 4 and an implementation is shown in Chapter 5.

2.3. Real-Time Operating Systems

Basically, a real-time Operating System (OS) has to provide the usual OS services (such as process and memory management, interprocess communication and I/O) with an emphasis on timely execution of tasks [YPW97].

2.3.1. Features

Like common operating systems, real-time OSs have four main functional areas:

(i) *Process* or *task management*

(ii) *Memory management*

(iii) *Interprocess communication*

(iv) *Input/Output* (I/O)

As an additional feature for real-time OSs, not found in general purpose OSs, one might add *time-management* [Kop97, p. 218]. This point means that the OS offers special services, like *clock synchronization* or support for *time stamping*.

Regarding to task management, the basic problem is the real-time scheduling problem already discussed in section 2.1.1. The resource access that has to be scheduled is the *CPU access time* for each process/task.

For the other three functional areas the real-time scheduling may arrise as well (e.g. for shared I/O resources). Additionally, each function provided by the OS has a bounded execution time.

Safety-Critical, Hard Real-Time

The focus of this thesis is in the field of safety-critical real-time systems. Therefore the support for such systems is also a required feature for the OS. The main difficulty regarding this, is the tradeoff between functionality and simplicity. Usually, OSs that are offering a wealthy set of features, are complex pieces of software. As an example, a current Linux kernel[5] consists of approximately 5.6 million lines of code. This huge size mainly results from device drivers (more than 50% of the code), multiple network protocols and different filesystem implementations.

Those complexity makes it difficult or even impossible to use such an OS for safety-critical hard real-time systems since it's not possible any more to verify the correctness of the software.

2.3.2. Available Real-Time Operating Systems (RTOS)

Real-time operating systems are an integral part of real-time systems [RS94]. This paper introduces three major categories of RTOSs that will be — slightly modified[6] — used here too: small, specialized kernels, real-time extensions to existing operating systems and research operating systems.

Small, Fast, Specialized Kernels

Those kernels, either homegrown or commercial, are often used for embedded systems. They are typically rather small and offer fast, low overhead execution and high predictability. Quite often these kernels are highly application- and platform-dependent (especially the homegrown ones).

Some examples for this category, either commercial or open source, are[7]:

[5] Linux 2.6.9 — the kernel version also used for the implementation of the DECOS partitioning operating system described in Chapter 5.

[6] [RS94] doesn't mention open source RTOS variants which became increasingly popular in the last years.

[7] more examples can be found on `http://www.faqs.org/faqs/realtime-computing/list/`

- *DeltaOS* from CoreTek Systems[8]
- *vxWorks* from Windriver[9]
- *eCos* from Red Hat, eCos group[10]
- *LynxOS* from LynuxWorks[11]

Real-Time Extensions to General-Purpose Operating Systems

The term *general-purpose* OS is used for OSs that have not been designed with real-time capabilities in mind. The reason to extend such OSs with real-time features is mainly their functionality. They might already support the target platform, offer device drivers, are compatible with convenient and familiar software development tools or offer a standardized API (like POSIX [Ope04]).

The main problem with this approach is that the general-purpose OSs are usually designed for high average performance and flexibility and not for predictability. Furthermore, due to their relatively huge size, it's in general not possible to verify them for example according to the highest criticality levels defined in [RTC92] (e.g. class A).

Due to these problems this category of RTOSs is in general not suited for safety-critical real-time systems, but can be used for soft-real time systems or for prototyping hard-real systems (as shown in Chapter 5).

Section 2.3.3 discusses this approach a bit further with Linux as the underlying OS.

Research Operating Systems

An overview over recent trends and research projects in real-time computing can be found in [But06]. Interesting points are e.g. *energy-aware scheduling* for low-power devices, *portability* by using standardized APIs, high level *modeling* of system resources, a trend towards *component-based real-time OSs* and *resource partitioning*.

2.3.3. Real-Time Linux

The increased usage of Linux in embedded systems [LD05] both by the industry and in academic research, has lead to multiple different implementations of real-time services within Linux systems, as an extension to the base Linux kernel like described above in Section 2.3.2. The website[12] of the Real Time Linux Foundation, Inc. lists 11 different

[8] `http://www.coretek.com.cn`
[9] `http://www.windriver.com`
[10] `http://ecoscentric.com`
[11] `http://www.lynuxworks.com`
[12] `http://www.realtimelinuxfoundation.org/`

non-commercial real-time Linux variants. Those different implementations can be grouped into two alternative approaches[13] which will be referred to as *preemption improvement* and *interrupt abstraction*.

In the preemption improvement approach, the Linux kernel is modified to reduce the amount of time that the kernel spends in non-preemptible sections of code [Bir00]. The principle goal is to minimize the length of the longest non-preemptible code. This minimizes interrupt latency and the shortest scheduling latency that can be *guaranteed* for a hard real-time system running on Linux. The most well known effort to patch the Linux kernel in such a way [DW05], is driven by MontaVista[14].

The second basic approach, used by FSMLabs[15] in their RTLinux and by the RTAI Linux project, is to run the Linux kernel as the lowest priority process of an underlying micro- or nanokernel. This additional kernel can also be seen as hardware abstraction layer and intercepts hardware interrupts. The work presented in Chapter 5 is based on RTAI Linux and, as a matter of fact, uses this approach. The strengths are excellent scheduling latencies, hard real-time support in some configurations and reasonable fault isolation. On the down side, applications have to deal with different APIs of two OS instances and explicit communication as well as performance and scalability penalties [McK05].

2.4. Related Work

Switching from a federated to an *integrated architecture*, like done in the DECOS (Dependable Embedded Components and Systems, [POT+05]) architecture, is a common goal nowadays. For the field of avionics, the Integrated Modular Avionics (IMA) approach [Mor91] describes a dependable integrated architecture. Top-level goals of the IMA architecture are the reduction of overall cost of ownership. It lowers acquisition costs, reduces spares requirements, reduces equipment removal rate, and reduces weight and volume both in wiring and avionics. At the same time, this architecture improves the system performance, testability and maintainability. An improved version of IMA is used by the NASA for the next generation of space avionics [BF04].

The switch from a federated to an integrated architecture is currently also an ongoing effort in the automotive industry. Major car manufactures created the AUTOSAR standard [AUT05] for this purpose.

[13] In a recent and very long post to the Linux kernel mailing list [McK05], even seven different approaches are discussed.
[14] MontaVista; `www.mvista.com`
[15] FSMLabs; `www.fsmlabs.com`

2.4.1. Available Partitioning Operating Systems

The ARINC 653-1 specification [Air03] defines an APplication EXecutive (APEX) software interface for partitioning operating systems. Example for available commercial operating systems which already fulfill this standard are LynxOS-178 RTOS from LynuxWorks[16] and VxWORKS AE653 from windriver[17].

2.4.2. Virtualization Techniques

Virtualization is a growing trend in the computer industry especially since low cost, commodity CPUs start to provide hardware support for virtualization [vD06]. Mostly this trend is driven by the demand for increased overall system security and reliability by concurrent execution of multiple operating systems on *virtual machines. Those virtual machines must be isolated from one another: it is not acceptable for the execution of one to adversely affect the performance of another* [BDF+03].

This concept of partitioning is a bit different than in a partitioning OS since, instead of application tasks, the partitions contain full blown operating systems. Virtualization is provided by a thin layer between those OSs and the hardware, called *hypervisor*.Examples for such hypervisor software are the commercial VMware[18] and the open-source XEN[19] solution.

The target market for the two examples above is in the field of database and web servers. Nevertheless there are already ongoing projects to bring the concept of hardware supported virtualization into the area of embedded systems [IIK+06]. Considering the rapid development of CPUs in the last centuries one can expect such features in embedded systems at some point in the future, just like memory management units can already be found in ECUs.

[16] `http://www.lynuxworks.com/solutions/milaero/arinc-653.php`
[17] `http://www.windriver.com/products/product-overviews/Platform-Safety-Critical.pdf`
[18] `http://www.vmware.com`
[19] `http://www.xensource.org`

Simplicity is prerequisite for reliability.

Edsger W. Dijkstra (1930–2002)

Chapter 3.

The DECOS Integrated Architecture

EMBEDDED COMPUTER SYSTEMS are already an indispensable technology in todays information society and will likely become even more important. A crucial fact for the seamless integration of those embedded systems into our daily live is their *dependability*.

The DECOS (Dependable Embedded Components and Systems) project[1] aims at developing an architecture that produces dependable systems while reducing development, production and maintenance costs at the same time.

3.1. Introduction and Motivation

If one looks at current automotive or avionics systems a steady increase of used electronic devices can be observed. This is due to the fact that requirements are getting bigger and bigger. Cars for example *are no longer simple means of transport but rather need to convince customers with respect to design, performance, driving behavior, safety, infotainment, comfort, maintenance and cost* [POT+05]. This trend has gone for a while now — today's cars may have around 4 kilometers of wiring compared to about 45 meters in 1955 — and it is estimated that more than 80 percent of future innovations in cars will result from advanced electronics [LH02]. Similar points are also arising in the avionics domain and cause an overall increasing system *complexity*.

Going along with an increasing system complexity is the number of built in Electronic Control Units (ECUs) since the traditional design is to use one dedicated ECU for one function. Distributed systems like this are called *federated systems*. In such an architecture, each electronic system uses an own communication network and its own choice of telecommunication protocol (e.g. Controller Area Network (CAN) [Bos91] or Local Interconnect Network (LIN) [LIN03]). The downsides of this approach are high costs due to the high number of ECUs, a considerable amount of weight resulting from the wiring, huge system complexity and maintenance costs because of multiple com-

[1] `http://www.decos.at`

munication protocols. On the other hand, the big advantage and also the reason for the rise of these federated systems is that partitioning of various systems is inherent.

The DECOS architecture aims at developing an *integrated system* where different functions share hardware resources and communicate with each other via a homogeneous network. This architecture should help to decrease *complexity* and *maintenance costs* while providing an increased level of *dependability*.

3.2. System Structure

The DECOS architecture offers a framework for the development of distributed embedded real-time systems which is based on a time-triggered core architecture. It is designed to integrate multiple applications with different levels of criticality (safety-critical and non safety-critical) that share a common communication network. Standardized interfaces and a set of provided high-level services ensure strong encapsulation of the different subsystems.

3.2.1. Functional Structure

Figure 3.1 shows an overview of the DECOS Integrated System Architecture (from [KOPS04]). In essence the DECOS integrated architecture consists of four layers. Distributed Application Subsystems (DASs) consisting of multiple jobs are using the provided high-level services. Those are built on top of a minimal set of core services which are provided by the core architecture.

The Time Triggered Architecture

The time-triggered architecture (TTA) [KB03] provides a computing infrastructure for the design and implementation of large distributed embedded real-time systems in high-dependability environments. A fault-tolerant global time base is the fundament of this architecture and helps to provide a simple communication protocol (based on time division multiple access (TDMA)), perform error detection and to guarantee timeliness of real-time applications.

Lets first elaborate the most important points of the architectural model of the TTA:

(i) The TTA uses a *Sparse Time Base* like shown in Figure 3.2. This time base is introduced to cope with the problem of finite precision of a global time base in a distributed system in particular and digitalization of a dense timeline in general. No matter how precise the global time is, it is not possible to order events in a consistent way based on their global timestamp. The TTA solves this problem

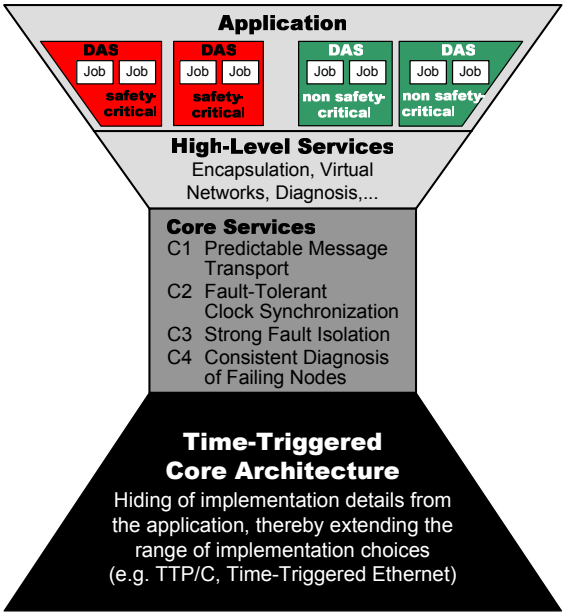


Figure 3.1.: Functional structure of the DECOS Integrated Architecture. Multiple jobs use the high-level services of the communications network to form a Distributed Application Subsystem (DAS). Those high-level services are build on top of core services provided by the time-triggered core architecture.

with a sparse time base which considers all events that occur during the same duration of activity to be happened at the same time.

Events within the architecture (like sending messages) are always appearing during such an activity period per design. Events from outside, occurring during the interval of silence, are mapped to an activity period by an agreement protocol.

Due to this sparse time base all nodes of the system have a consistent concept of "*at the same time*", "*in the past*" and "*in the future*". The duration of silence can be seen as dividing line between the past and the future.

(ii) The TTA uses the concept of *state information* instead of *event information* to update its real-time images (like described in Section 2.1.2). This means that a property of a real-time entity is observed at a given point of time and this

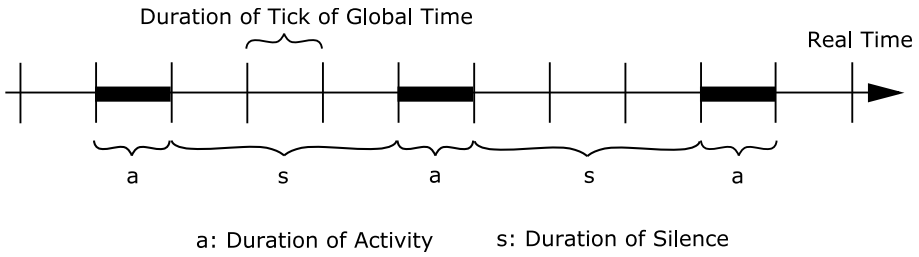


Figure 3.2.: Sparse time base

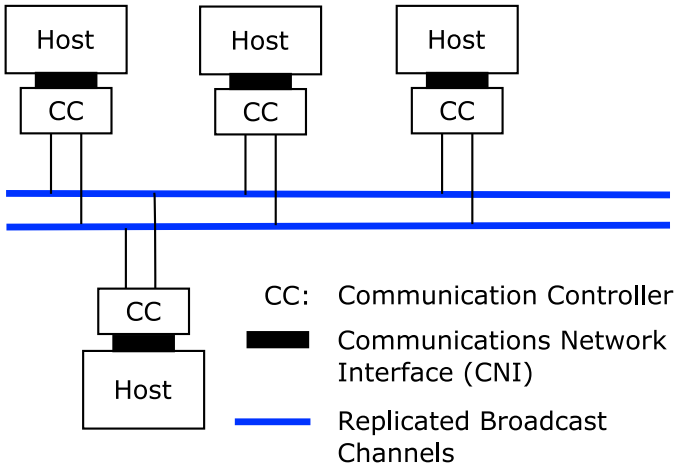


Figure 3.3.: Structure of a TTA cluster

observation is encoded into the atomic triple

$$< \text{Name of variable, value, time of observation } t_{obs} >$$

These state observations are taken periodically and result in a series of snapshots with constant time difference. From this sequence of snapshots it is possible to *reconstruct events* that raised in between two snapshot instants.

The principle structure of a distributed system based on the TTA can be seen in Figure 3.3. Basically it consists of a fault-tolerant broadcast channel and a set of *nodes* connected to this channel and using it for exchanging messages. In order to do so each node has a *TT communication controller* (CC) responsible for accessing the communication channels. The *host computers* are connected to the CC via a CNI (Communication Network Interface).

As already stated, the TTA uses a TDMA scheme for the bus. Based on the global time and a static *message scheduling table* (the Message Descriptor List (MEDL)) the

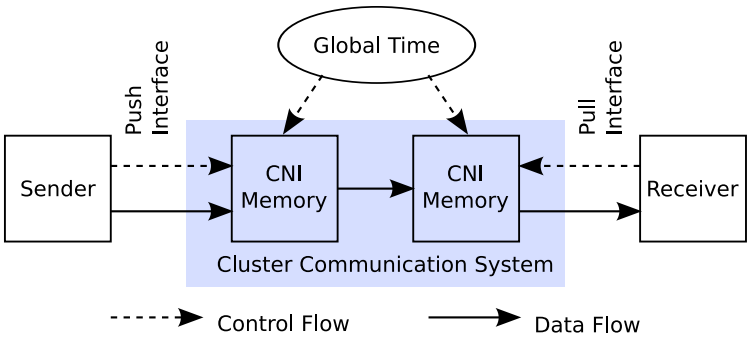


Figure 3.4.: Data and control flow at a TTA interface.

CCs of the different hosts access the communication system periodically at *a priori* known points of time. The transmission of a (state) message works as follows: The sending host writes a state message to its CNI. The corresponding CC will *fetch* the message at an *a priori* known instant of time and sends it to all other CNIs in the cluster at an *a priori* known delivery instant. This new state message overwrites the old saved state message from the sending host on all receiving hosts[2]. All receiving host are now able to read their local copy of the state message if they are interested in it. Figure 3.4 illustrates the control and data flow. This communication scheme with all its predictable messages (in the time domain) makes the TTA very suitable for distributed hard real-time systems.

On the dependability side there is still something missing in Figure 3.3. It does not show the replicated bus guardians. Their job is to use the information of the MEDL to supervise the bus and check if each CC respects its time slot defined in the MEDL. If a CC breaches its time slot, the bus guardians are able to cut this obviously faulty node from the bus. More details about that can be found in [KB03].

Core and High Level Services

Based on the TTA it is possible to build a minimal set of *core services* (Figure 3.1).

(1) *Predictable message transport*

(2) *Fault-tolerant clock synchronization*

(3) *Strong fault isolation*

(4) *Consistent diagnosis of failing nodes through a membership service*

[2]Recall that if no event, observable by the sending host, happened, the previous version is exactly the same as the new one. It will be overwritten nevertheless.

In fact, the underlying core architecture does not have to be the TTA. Each architecture providing these four core services is suitable for the DECOS integrated architecture.

Based on the minimal set of core services, DECOS allows an extensive number of *high-level services*. Examples would be virtual networks [OPK05], encapsulation [HPOS05] and diagnosis services [PO06] . The need for this high-level services depends on the demands of the DASs. For example it might be necessary to build a virtual CAN network for a legacy application.

Applications

A Distributed Application Subsystem (DAS) is a part of the systems overall functionality that is typically distributed among several physical ECUs to provide fault tolerance by redundancy. Examples for DASs in the automotive domain could be a brake-by-wire DAS, a steering-by-wire DAS, a passive safety DAS, a multimedia DAS and so on. Although all of those DASs can have a different level of criticality, they share a common core communication infrastructure.

DASs themselves can be decomposed into multiple *jobs* which are the most basic units of the distributed system. They can be located on different physical component and use virtual networks [OPK05] (provided as high-level DECOS service) to communicate with each other. The access point of a job to a virtual network is called *port*. Those virtual networks can either belong to the event-triggered or time-triggered paradigm. Important is the strong encapsulation against other virtual networks belonging to other DASs.

3.2.2. Physical Structure

The DECOS architecture is a distributed system consisting of several *components* (Figure 3.5) interconnected by a *physical network*. Since this component is a node of the underlying TTA cluster (compare Figure 3.3) there is the need for a *Communication Controller* (CC) which is accessed by the rest of the component via the CNI. On top of this interface is the *Basic Connector Unit* (BCU) which provides interfaces for the safety-critical and the non safety-critical subsystem of the component. Between the application computers, executing the various jobs, and the BCU are *Secondary Connector Units* (SCUs) which are called *safety-critical connector units* in the *safety-critical subsystem* and *complex connector units* in the *non safety-critical subsystem*. Those SCUs provide the DECOS high-level services to the jobs running in the application computers. Each component must have at least two independent processor cores to grant encapsulation for the safety-critical and non safety-critical subsystem.

Such a component will usually be implemented as a System-On-a-Chip (SOC) but

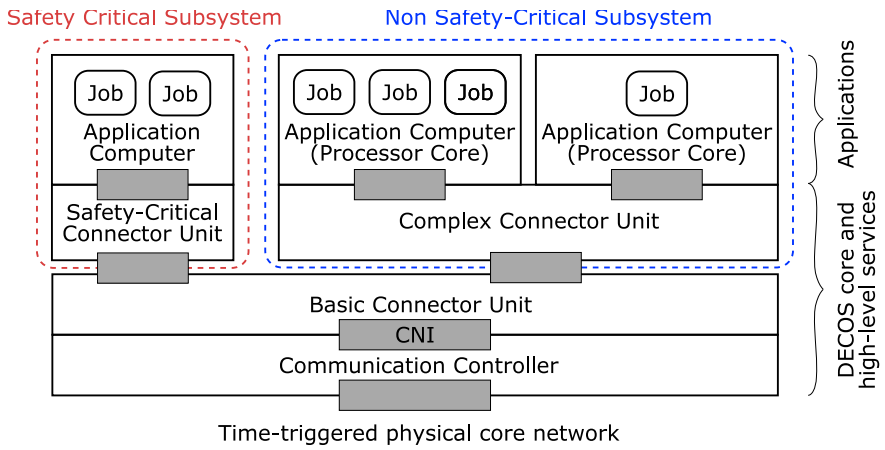


Figure 3.5.: Physical structure of a DECOS component with three multiprocessors.

can also be distributed among several ECUs implemented as a single board computer[3].

3.2.3. Fault Hypothesis

The fault hypothesis for the DECOS architecture is described in detail in [KOPS04] and distinguishes between *hardware* and *software faults*.

For hardware faults, the FCR is one component, since it is typically implemented as a System-on-a-Chip (SOC) and contains shared resources (e.g. processor, memory, power supply, oscillator) which can be affected by a single fault. The failure mode of a hardware FCR is assumed to be arbitrary. Furthermore it is assumed that only one hardware FCR is faulty at the same time.

The FCR for a software fault is a job. The failure mode of a job is the violation of its port specification either in the time or in the value domain.

[3]The DECOS prototype implementation that will be introduced in 5.1 has its own independent hardware components for the BCUs and for the SCUs which communicate via Time-Triggered Ethernet (TTE).

I don't need to waste my time with a computer just because I am a computer scientist.

Edsgar Dijkstra (1930 - 2002)

Chapter 4.

Partitioning Operating System for DECOS Nodes

In the previous chapter the functional and physical structure of the DECOS system was introduced (Figure 3.5). It was stated that a physical DECOS component consists of several hardware elements like communication controllers, connector units and application computers. Those application computers can execute multiple jobs belonging to different DASs (see Figure 4.1). In order to do this, it is necessary to have an Operating System (OS), also called *Primary Operating System*, that manages the available resources like memory and CPU time. This chapter will investigate the requirements for such an OS. An implementation is presented in the next chapter.

The usual feature set of an OS can be classified into process scheduling, memory management, interprocess communication and Input/Output. In addition to those features, the primary operating system has to provide a set of DECOS specific features. One of them is support for hard-real time and another one is *partitioning*.

Partitioning can be splitted into two dimensions, *spatial* and *temporal* partitioning. Spatial partitioning, as discussed in detail in Section 4.2, deals with the separation

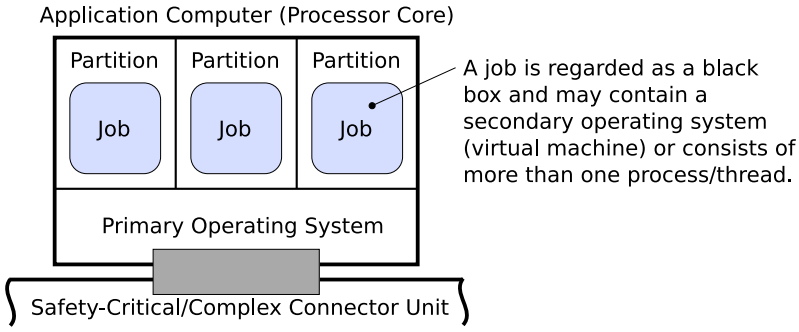


Figure 4.1.: Structure of Non Safety-Critical Application Computer Software [KOPS04]

	Time domain	Value domain
Temporal partitioning	The shared resource CPU is blocked by one partition → software in another partition misses its deadline.	A shared sensor is blocked by one partition → missed update for sensor value in another partition results in a wrong output.
Spatial partitioning	Data in memory is overwritten by one partition → software in another partition crashes due to invalid input	Data in memory is overwritten by one partition → software in another partition using this data calculates a wrong output.

Table 4.1.: Examples to show that insufficient temporal and spatial partitioning can result in failures in the time or value domain.

of resources. Software in one partition is not allowed to access private resources in another partition. Temporal partition (Section 4.3) ensures that the quality of service received from a shared resource, cannot be degraded by software in other partitions.

In the beginning of Chapter 2 it was stated, that the correctness of computational results in a real-time system depends on the correct logical result and the instant of time at which this result is produced. Table 4.1 shows, by listing examples, that insufficient temporal partitioning can lead to failures in the temporal *and* in the value domain. The same is true for spatial partitioning.

4.1. Operating System Overview

Before listing the features such a OS has to provide, some constraints have to be made. The first one is that a OS for a single processor system is assumed. If a DECOS node contains more than one application computer[1], more than one instance of the OS has to be running. Each instance takes over control over a distinct CPU, own memory and a dedicated I/O subsystem.

In general there are different needs for the OS, depending whether it is running in the safety-critical or in the non safety-critical subsystem. Software which runs in the safety-critical subsystem belongs to the highest criticality class (e.g., level A and B in [RTC92]) and has to be certified to this highest criticality levels. The OS running in the non safety-criticality subsystem does not have to be certified to such a high level and can provide an increased functionality and flexibility at the price of increased complexity [KOPS04]. The focus in this thesis will be on the safety-critical subsystem.

[1] In theory, a DECOS component has two application computers to separate the safety-critical from the non safety-critical jobs.

4.1.1. Features

Since the DECOS architecture has to support safety-critical jobs, it is already clear that an OS managing safety-critical jobs has to support hard real-time. The second big point is that this OS has to provide strict partitioning features to guarantee the fault-hypothesis for software faults.

Besides these two high level demands, the usual OS services which are commonly categorized into process management (scheduling and Inter-Process Communication (IPC)), memory management and I/O operations, have to be fulfilled as well.

Basic Features

Scheduling Section 2.1.1 already stated the advantages of static scheduling compared to dynamic online scheduling. Since the whole DECOS architecture is strictly focused on a-priori known, predictable events, a static scheduling approach perfectly fits to this primary OS.

Note that in DECOS it's possible to have several processes/threads inside one partition. In this case, the partitioning OS only schedules the partitions and a *partition operating system* has to take over the part of scheduling the processes (*multi-level scheduling*). This scheduling can be as simple as just executing several functions calls, implementing the job functionalities, in a row, up to a full featured virtual machine hosting the jobs.

Inter-Process Communication Usually, operating systems provide means of *inter-process-communication* like shared memories, message queues or pipes.

In the context of the partitioning OS this feature would map to inter-partition communication. This should never be necessary between two *usual* partitions since even if two jobs, belonging to the same DAS, are located on the same physical DECOS component they are only allowed/able to communicate via their DAS's virtual network. Using special means of inter-partition communication only available for jobs located at the same physical component would violate *location transparency* (like defined in [TvS02, p. 5]).

Inter-partition communication is needed as soon as high-level services are implemented inside partitions. For example, the prototype implementation which will be presented in Chapter 5 uses a dedicated partition to implement the SCU and its middleware services. Shared memory as a mean of inter-partition communication implements the *ports* used by the jobs to communicate with the SCU.

Memory Management deals with the allocation of memory for each job/partition. From the partitioning point of view, it to be ensured that no job is able to write

into memory belonging to another partition. Besides that, recall that all memory allocations have to be done in hard real-time.

Input/Output Interaction between jobs and the environment (I/O) is entirely done via *local interfaces* (compare with Figure 2.1, page 7). These local interfaces can be provided by the OS in form of an API to a device driver or by memory mapped I/O.

Intellectual Property Protection

It has to be possible to pre-compile software modules and use this object code for jobs. Especially in the automotive sector this feature is a necessity for software suppliers in order to maintain their competitive edge. Section 5.2.2 shows how object code is linked together with wrapper code to build a job.

On architectural level this requirement is supported by DECOS. A DAS developed by a vendor can be integrated into the system without knowing its internals.

4.1.2. Fault Hypothesis for Software Faults

The fault hypothesis for the DECOS architecture (Section 3.2.3) already declared a job as FCR for a *software fault*, commonly called *bug*. This assumptions only holds if the OS is considered free of bugs. Otherwise the FCR would be a whole subsystem (safety-critical or non safety-critical).

Before going into details the terminology should be clarified. The term *"bug" refers to an instance of errors, faults and failures that are inconsistent with a programmer's expectation of program behavior* [KM03]. In this thesis, with its separation between the terms error, fault and failure, a *bug* is considered as a software *fault*.

If one tries to map the fault-error-failure chain from 2.2.1 to software bugs, a *failure* occurs when program output is inconsistent with the programmer's expectations of program output. *Errors*, that cause this failures, are runtime states that are inconsistent with the programmer's expectations of runtime state. And finally, faults are caused by *programming faults*, *which are program fragments that are inconsistent with a programmer's understanding of the program fragment's semantics*[2].

Examples for such software faults could be:

- A programmer writes a loop to iterate through all elements of an array (her understanding of the program fragment). In fact the real semantic of the program fragment is to iterate through all elements of the array plus one more. This kind of bug is very frequent and is also called *off-by-one* error.

[2]Note, that in this paragraph the terms *software error* and *software fault* are swapped compared to [KM03] in order to comply with the terminology used by [ALR01] and in this thesis.

- A general problem on multitasking systems are resources that can be accessed by different programs. A programmer might think that it's perfectly okay to use a resource, while the real semantic of this action includes a change in the state of a different program.

Obviously the primary operating system cannot prevent software bugs inside single programs like in the first example but it has to ensure that the FCR for such a bug is the program/the partition that contains the job. A bug like described in the second example *can be prevented* by the OS if it ensures strict partitioning!

4.2. Spatial Partitioning

> Spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit) nor command the private devices or actuators of other partitions [Rus99].

This is a rather abstract definition and in this section it is mapped to the actual system platform by splitting it into three parts. First the software itself and private data in memory is considered. The next section deals with private data in transit which basically means communication. And finally private devices are studied.

4.2.1. Software and Private Data

Most common computer systems use the *von-Neumann architecture*. This computer architecture uses a common memory to store application (program) code and data. This main memory, usually called RAM, holds the application code and private data for the OS and all started processes[3]. An alternative approach is the *Harvard architecture* which introduces a clear separation between program and data memory.

No matter which kind of architecture is used, spatial partitioning must ensure that it's impossible for software running in one partition to write into memory belonging to another partition.

4.2.2. Communication Lines

A communication line can be considered as a private device belonging to the communication partners. Spatial partition must guarantee that only those communication partners are authorized to change data in transit.

As mentioned above, jobs communicate via virtual networks provided by the SCUs. Data inside the input or output buffers can be considered as *in transit* and its integrity

[3] This is the simple case and does not consider things like swapping.

has to be protected. Message buffers belonging to one partition shall only be accessed by this partition.

4.2.3. Private Devices or Actuators

It's important to distinguish between real *private* resources and shared resources that are only scheduled to a partition for a certain time. For the first kind, the requirements for spatial partitioning are easily defined: No other partition shall be able to access them at all. Resources that are shared among several partitions are discussed in the next section.

4.3. Temporal Partitioning

> Temporal partitioning must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition. This includes the performance of the resource concerned, as well as the rate, latency, jitter and duration of scheduled access to it [Rus99].

Before proceeding, those shared resources should be identified. From a physical point of view, the most obvious shared resource is the CPU. Another shared resource is the memory, seen as hardware component which is used by all partitions[4]. Those two definitely exists on every application computer[5], but depending on the implementation, there can be more (e.g. several devices sharing a PCI bus — the PCI bus is a shared resource). Note that there can also be *logical shared resources*. An example might be a limit on the total number of processes, set by the OS.

The definition of temporal partition stated above can be splitted into several parts. In this section, the scheduled access to shared resources is addressed first. Affecting the performance of a shared resource is discussed later on. Beforehand it can already be stated that problems with temporal partitioning are quite often more subtle that violations of spatial partitioning. Spatial partitioning is usually a yes/no decision whereas service received from shared resource might be affected "just a bit".

4.3.1. Scheduled Access

Temporal partitioning must ensure that no partition can affect the rate, latency, jitter or duration of scheduled access from another partition to a shared resource. An extreme example is a partition blocking a resource (like blocking the CPU by executing an endless loop) and making it completely unavailable for other partitions.

[4] This memory should not be mixed up with private memory already assigned to a partition.
[5] At least when following a common computer architecture like von-Neumann or Harvard.

Further examples for problems that can be classified as a violation of temporal partitioning in the area of scheduled access (to the CPU):

- Kill or crash software in another partition. This basically means that the OS will not schedule the CPU to this partition any more at all.
- Interrupt running software in one partition by an interrupt caused by a private hardware device of another partition.
- Software of one partition that is able to rise its own priority as seen from the scheduler, and therefore relatively downgrading the priority of all other partitions. This new priority configuration might lead to a different access schedule.

An example for the access time or the access jitter: Lets suppose that the sending buffer of a communication device is filled by data from a partition, waiting to be sent. Before this sending takes place, the device is scheduled to be accessed by another partition which immediately tries to send something. There are two possibilities now. Either the data already in the send buffer is discarded (which would indicate a spatial partitioning violation) or the access to the device is postponed until it's available again (temporal partitioning violation).

4.3.2. Resource Performance

Affecting the performance of a shared resource can go from disabling it to slow it down. Some straight forward examples would be:

- Disabling a device by software could be done via unloading a device driver.
- Scale down the CPU frequency.
- Slow down a device by falling back to some legacy mode. For example switching an USB device from the USB2 protocol back to USB1.1 or a wireless network card from IEEE802.11c to the slower IEEE802.11a operating mode.

Clearly those examples can only happen if the whole environment (Resources/Devices, device drivers, OS) supports such operations and the various partitions are authorized to trigger them.

A further example that is more subtle is the *memory performance* of a multilevel memory hierarchy. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The fastest memories are located closer to the CPU and are usually smaller and more expensive [PH98, chap. 7]. Let's suppose a two level memory hierarchy, consisting of a small and fast *cache* on the highest level and a bigger but slower main memory on the second level. The access time to data in the memory is different if it's already in the cache or if the access results in a *cache miss* and the data

has to be fetched from the slower main memory. A partition replacing all the data in the cache with its own, will decrease the memory performance for the next scheduled partition.

The Linux philosophy is "Laugh in the face of danger". Oops. Wrong One. "Do it yourself". Yes, that's it.

Linus Torvalds

Chapter 5.

Implementation of a Partitioning Operating System on Top of RTAI-LXRT Linux

So far, the theory behind the DECOS architecture and the need for a partitioning OS on DECOS nodes has been introduced. This chapter will go into implementation issues. The starting point is an already existing prototype implementation of a whole system using the DECOS architecture. Its hardware setup will briefly described in Section 5.1. More interesting, regarding this thesis, is the software setup of the DECOS nodes (Section 5.2). They are running the RTAI real-time Linux variant and use the LXRT extension for hard real-time in user space. Each of this user space tasks/jobs is regarded as a partition by the partitioning OS.

In the beginning of Section 5.3 chosen features of RTAI-LXRT are discussed in detail and a couple of hypotheses about how well the partitioning is *expected to work*, are brought up. This can be seen as *qualitative fault forecasting*. The validity of the hypotheses is shown by a *quantitative* method — running test cases.

The results gained in the first round of test are used in Chapter 6 to design an additional security layer resulting in a new job API to the primary OS. This new API is called POS (Primary Operating System) interface. The implementation is done in partition context but also in kernel-space by the RRI (Restricted RTAI Interface) kernel module.

5.1. Hardware Setup of the DECOS prototype

The existing DECOS prototype as in Figure 5.1, is a cluster with five integrated components like depicted in Figure 5.2.

One single board computer, equipped with a TTP communication controller, forms the Basic Connector Unit (BCU) and the Communication Controller (CC) for the underlying time-triggered core network. The safety-critical and the non safety-critical

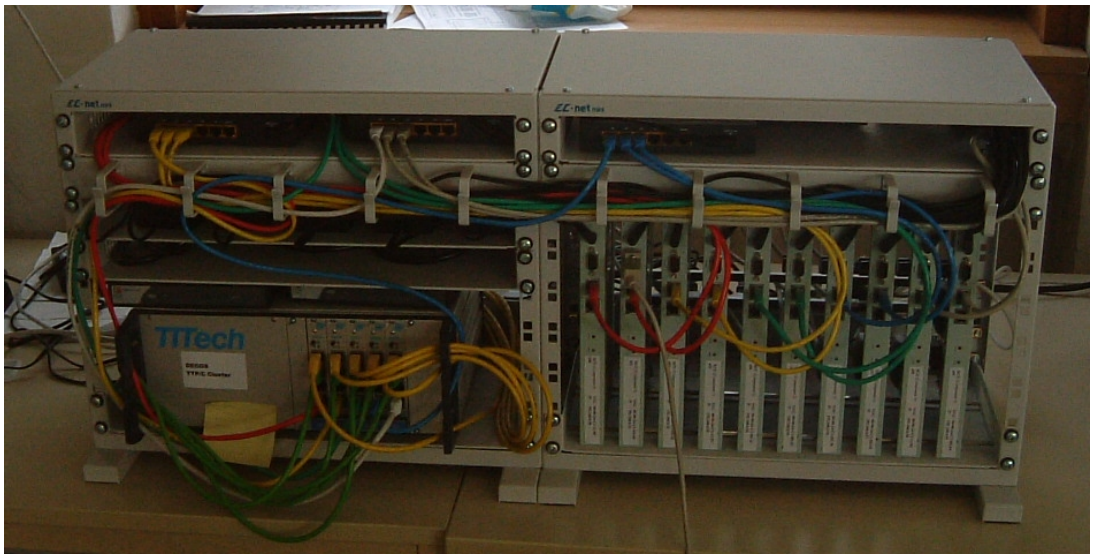


Figure 5.1.: The DECOS prototype cluster

subsystem are implemented by two additional single board computers. They implement the Safety-critical or Complex Connector Unit (SCU, XCU) and the application computers. As a matter of fact, each DECOS component is divided into three independent single board computer subsystems. These computers are interconnected via a 100 Mbps Ethernet network and use Time-Triggered Ethernet as communication protocol.

The primary operating system is running on the Secondary Connector Units, which use a Soekris Engineering net4521 [SOE] embedded system as hardware platform (see Figure 5.3). They are based on a 133 Mhz 486 class processor with up to 64 Mbyte of RAM. For communication purposes, two 10/100 Mbit Ethernet ports are built in. A MiniPCI and two PC-Card/Cardbus adapters are available for expansion. Programs and data can be stored on a CompactFlash card.

For testing purposes, the hardware setup was simplified, and only a single Soekris board connected to two other computers was used. Details on this can be found in Section 5.2.3.

5.2. Software Setup for the Secondary Connector Units

Since the SCU and the application computer are implemented on a single board computer the separation between those two parts of a DECOS component (compare Figure 3.5, page 24) is done in software. In fact, the high level services of the SCU are packed into a single partition, just like the jobs (Figure 5.4).

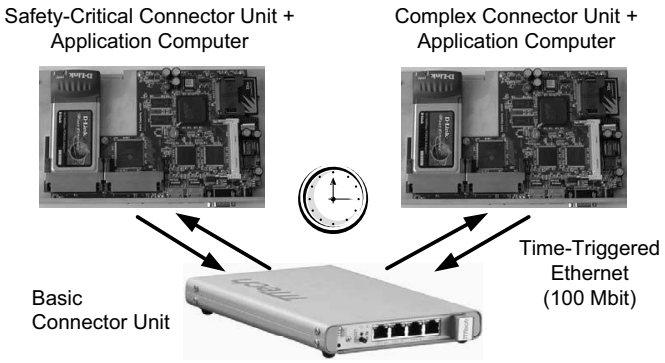


Figure 5.2.: A protoype DECOS component [HPOS05].

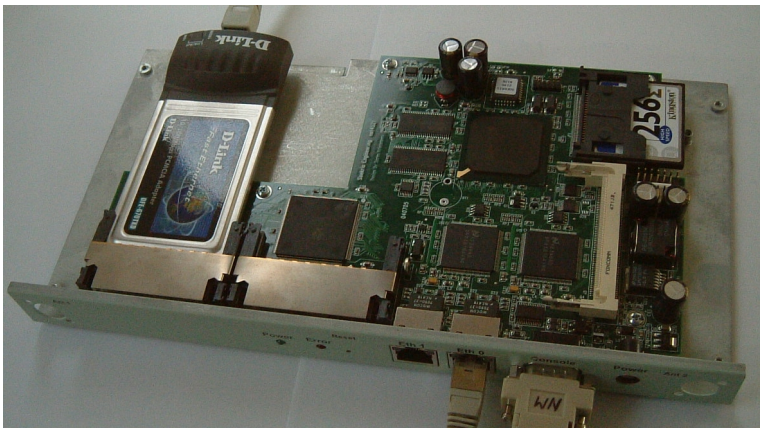


Figure 5.3.: A Soekris net4521 computer

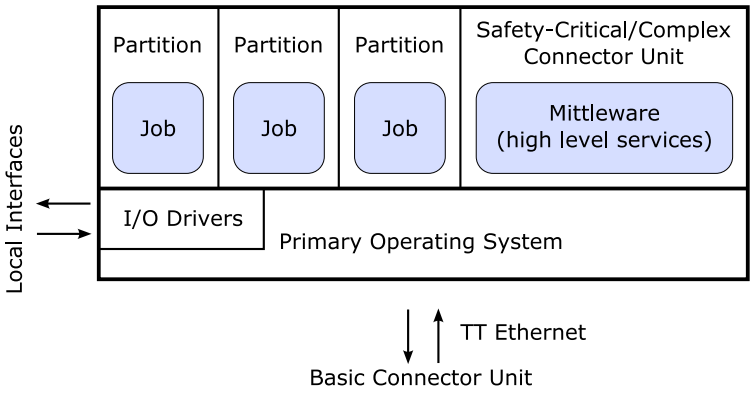


Figure 5.4.: Software running on the soekris application computer.

The primary OS is implemented on top of the real-time Linux RTAI (Real-Time Application Interface) v3.1 [MDP00]. The base Linux kernel version is 2.6.9. The execution environment formed by the primary OS and the different partitions, implemented as LXRT tasks, is described in Section 5.2.2.

5.2.1. RTAI-LXRT Linux

RTAI (Real-Time Application Interface) Linux[1] is an open source real-time Linux variant developed by the Dipartimento di Ingeneria Aerospazile Politecnico di Milano (DIAPM).

RTAI Linux used to include a patch to the Linux kernel to introduce a Real-Time Hardware Abstraction Layer (RTHAL). Due to patent problems, RTAI as of version 3.1, switched to the ADEOS (Adaptive Domain Environment for Operating Systems)[2] abstraction layer, which provides a similar functionality. From a functional point of view, one can think of ADEOS as a nanokernel which can be used to catch interrupts and distribute them to different hosted operating systems [Yag]. Like RTAI itself, the ADEOS layer is implemented as a Linux kernel module.

Figure 5.5 illustrates the different layers of the software architecture. RTAI runs on top of the ADEOS layer and uses its own real-time scheduler for its real-time tasks. The Linux kernel is executed as idle task with lowest possible priority.

LXRT Extension Module

RTAI real-time tasks are implemented inside Linux kernel modules. This makes them effectively part of the kernel and, as a result of this, they are executed in supervisor

[1] `https://www.rtai.org/`
[2] `http://home.gna.org/adeos/`

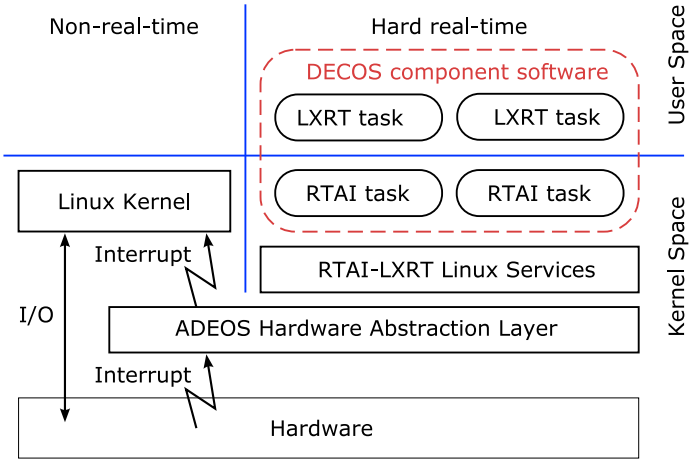


Figure 5.5.: Layers in the software architecture and separation into Non-real-time / Hard real-time and Kernel space / User space. DECOS component specific software is implemented as RTAI and LXRT tasks.

mode and not in user mode. Such software is able to access the whole memory without being restricted by any Memory Management Unit (MMU). From the viewpoint of (spatial) partitioning this is not acceptable. Fortunately, RTAI offers the LXRT (Linux Real-Time) extension that allows hard real-time tasks in user-space.

The LXRT module with its fully symmetrical API provides a safe and flexible tool to quickly implement hard real time programs in user space. Once the program is debugged, it can be easily migrated to the kernel for optimal performance if the application demands it [CM+00]. This quote introduces a very nice feature of LXRT, its symmetric API. From an implementation point of view, RTAI services can be used with the same function calls as in kernel space. The transition from user to kernel space and back is done by a RTAI real-time task agent (one for each LXRT task) that calls the native RTAI functions when required. This agent is created as soon as the function `rt_task_init()` is called and is completely transparent for the programmer of the LXRT task. More details about the features of RTAI-LXRT important for partitioning are discussed in Section 5.3.1.

Results on the performance overhead of such a user-space solution can be found in [BD00]. To summarize: The performance penalty is not very big (only a few microseconds more overhead) and becomes more and more negligible with fast CPUs.

5.2.2. Execution Environment

This section describes the execution environment including the primary OS and the different partitions as it's implemented on top of RTAI-LXRT Linux like described in

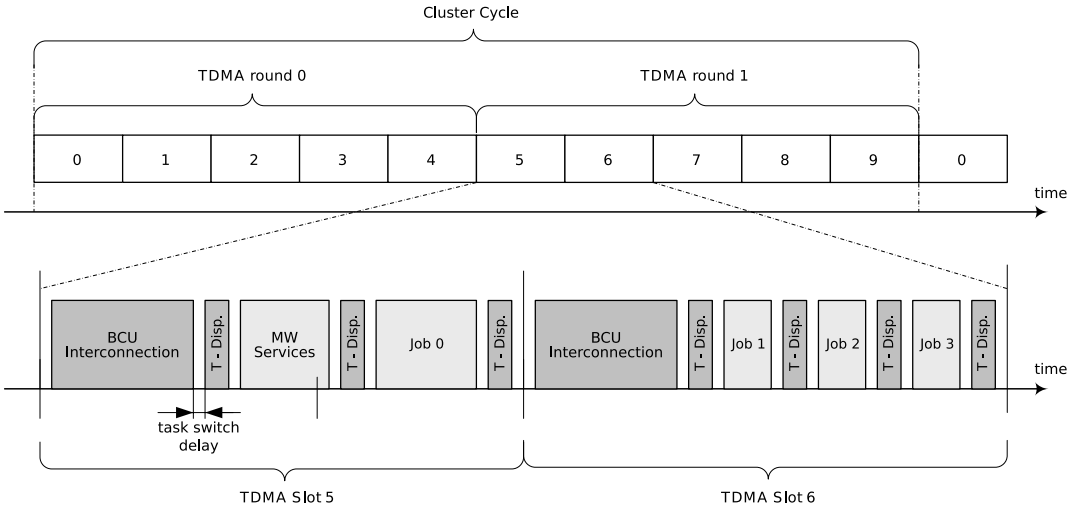


Figure 5.6.: Scheduling example for the SCU as part of the DECOS prototype cluster (from [HPOS05]).

[HPOS05].

A RTAI task acts as time triggered dispatcher/scheduler to enforce a static scheduled access to the CPU (depicted in Figure 5.6). Advantages of such a static scheduling have already been discussed in Section 2.1.1. *Simplicity*, *predictability* and the resulting possibility to look for a *feasible* schedule off-line in the system integration phase, are the reasons for choosing such a solution (see also [LKMZ00]).

The resulting scheduling is cyclic, with a period of *cluster cycle*. In the most general form, a cluster cycle consists of several TDMA *rounds* which can be sliced into several TDMA *slots* themselves. On the beginning of each TDMA slot, the primary OS uses a reserved time slot for communication with the BCU via TT-Ethernet. Right after that it starts to schedule the partitions. At the end of the TDMA slot, the scheduler stops and waits at a semaphore which is signaled by the arrival of a TT-Ethernet package[3]. This ensures, that the primary OS stays synchronized with the communication network. A pseudo-code implementation of the scheduler can be found in Figure 5.7.

Temporal partition is ensured by the fact, that the RTAI task running the scheduler is created with the highest priority.

Regarding the implementation of the partitions/jobs it's necessary to define the terminology first. In the prototype implementation each partition always hosts exactly one job. A job is implemented as a LXRT task and an each LXRT task is seen as a

[3]Per default, the package indicating the start of a new slot arrives every 2 ms.

scheduler

```
1  init_RTAI_task();
2  obtain_all_LXRT_tasks();
3
4  wait(semaphore);
5
6  while(1) {
7    BCU_interconnection();
8
9    while (!end_of_task_list) {
10     rt_task_resume(task[i]);
11     rt_sleep(task_exec_time[i]);
12     rt_task_suspend(task[i]);
13     i++;
14   }
15   wait(semaphore);
16 }
```

task_wrapper

```
1  init_LXRT_task();
2
3  init_point();
4
5  rt_make_hard_real_time();
6  rt_task_suspend();
7
8  while(1) {
9      entry_point();
10
11     rt_task_suspend();
12 }
```

Figure 5.7.: Pseudo code of the scheduler and the task wrapper.

Linux user-space process from the Linux kernel. Which of the terms — job, partition, LXRT task, Linux process — is used in the rest of this chapter depends on the current viewpoint but basically they are equivalent.

Each job is wrapped by a task wrapper. This task wrapper does the LXRT task initialization and calls specific job functions like shown in Figure 5.7. First the `init_point()` functions during the start-up and later on periodically the `entry_point()` function. Consequently a job, that is linked to the task wrapper object file has at least to provide those two functions as entry points.

IP protection is also supported by such a setup since the source code of the job implementation is not necessary to link it to the task wrapper. A compiled object file is enough.

5.2.3. Test Framework

In order to test the partitioning capabilities of the primary operating system, a test setup like pictured in Figure 5.8 is used. Instead of the whole cluster, the test hardware is just a single Secondary Connector Unit (SCU)[4] running the primary OS. This simplifies testing to a great extend and no generality is lost. For communication with the environment and supervising the execution, both, a serial connection to a workstation and a LAN connection to a server computer has been installed.

[4]Note that in this section the abbreviation SCU always means the general Secondary Connector Unit and not the specific Safety-Critical Connection Unit.

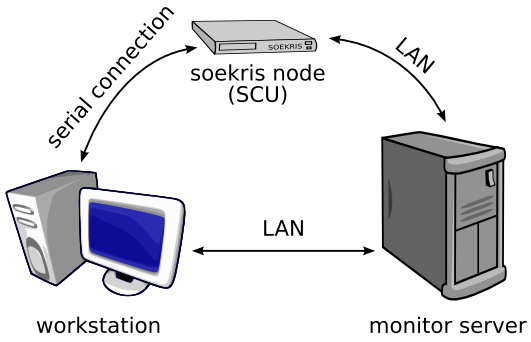


Figure 5.8.: Setup for testing the partitioning.

The execution environment was also adapted to reflect the changes in the hardware setup. Those adoptions are:

- The fact that there is no time-triggered Ethernet any more means that the semaphore that was triggered by the arrival of a new packet like described above, is not released via the network any more. Instead, a dedicated RTAI task (`ttsem`) periodically triggers the semaphore.

- Since there is no connection to a BCU, the time reserved for BCU interconnection makes no sense for this purpose any more. In the testing framework this timeslot is used to send timestamps — for the monitoring of job execution times and scheduling durations — to a monitoring server. Monitoring is discussed in detail in Section 5.2.4.

- Something that is not used in the test environment are the high level services (or middleware services). But since these services are implemented as an LXRT task, which runs in a partition just like all the other jobs, there is no loss of generality.

- To simplify the scheduling, the same scheduling will be used every TDMA slot (Figure 5.9). Each TDMA round contains only a single slot and each cluster cycle only a single TDMA round. This simplified setup makes multiple levels — cluster cycle → multiple slots → multiple different rounds — not necessary any more. The new terminology for the smallest period in the scheduling is just *round*.

  Furthermore the maximal number of different jobs in one round is limited to four.

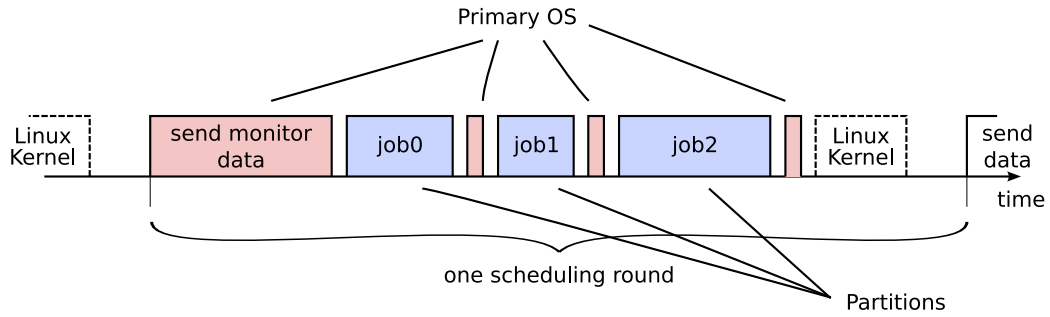


Figure 5.9.: Example scheduling on a SCU in the test environment.

The test framework is a loosely coupled system of several scripts and programs (Table 5.1). Each test case resides in its own directory (`test_name`) and is defined as a set of jobs that will be executed. Additionally it's possible to provide configuration files (C header files) for test-specific scheduling and startup scripts.

The open source scripting language Python[5] is used for the main script controlling the test framework. A simplified version of the `do_testing.py` script looks like this:

```
1 include config                          # read configuration file
2 test_list = get_list(config.tests) # create test objects
3
4 for test in test_list:
5     test.prepare()                  # prepare environment
6     reboot_s43()                    # reboot the SCU
7     test.start_mon()                # start monitord
8     sleep(config.sleep_time)        # wait for data
9     test.stop_mon()                 # stop monitord
```

A configuration file is used to define which tests should be carried out. In the following loop through all test, first the test specific prepare function is called. This function calls the `prepare.sh` shell script found in the test directory which copies jobs and header files into the build environment, overwriting the defaults if necessary. After a successful build, the executables, kernel modules and start-up scripts are copied into a directory that is shared via NFS (Network File-System).

This freshly prepared environment will be used by the SCU after the following reboot. The necessary files are copied over and the execution is started: (snippet from `start_rtnet.sh`)

```
1 # load kernel modules
2 insmod $DST_MODULES/log.ko
```

[5] `http://www.python.org`

`config.py`	Configuration file used by `do_testing.py` and `analyze.py`.
`do_testing.py`	Script to manage the execution of test cases.
`analyze.py`	Script to perform the statistical analysis of the gathered timestamps.
`prepare.sh`	Test specific shell script to setup the environment.
`start_rtnet.sh`	Test specific shell script executed on the SCU. Loads kernel modules (`log.ko`, `scheduler.ko` and `ttsem.ko`), starts the `jobs` and `testd`.
`monitord`	Daemon running on the monitor server. Receives monitor data via the LAN and stores the extracted timestamps in a database.
`log`	Kernel module to extend the Linux kernel on the SCU with a low overhead `udp_log()` function.
`testd`	Daemon running on the SCU. Reboots the computer as soon as an UDP packet arrives on port 12666.
`ttsem`	RTAI task running on the SCU. Simulates the time triggerd Ethernet and controls the lenght of the scheduler rounds by periodically signaling a semaphore (default: every 2 ms). (hard real-time)
`scheduler`	RTAI task that schedules the jobs on the SCU and sends monitoring data to the monitor server. (hard real-time)
`rri`	Kernel module used for restricting the API available to jobs. It will be introduced in Section 6.1.1 and is not used during the first round of tests.
`job0.c ...job`x`.c`	Test specific LXRT tasks. (hard real-time)
`task_wrapper.c`	Linked together with a job. Does the RTAI-LXRT initialization and periodically calls the entry point of the job.
`pos_interfacs.c`	Linked together with a job, just like the task wrapper. Implements the user-space part of the restricted JOB API and not used during the first round of tests.

Table 5.1.: The most important files and programs for the test framework

```
3  insmod $DST_MODULES/scheduler.ko
4
5  echo "starting testd ..."
6  $DST_APPS/testd &
7
8  echo "starting jobs ..."
9
10 sleep 1
11 $DST_APPS/job0 &
12 sleep 1
13 $DST_APPS/job1 &
14 sleep 1
15 $DST_APPS/job2 &
16 sleep 2
17
18 # load the ttsem kernel module
19 insmod $DST_MODULES/ttsem.ko
```

The test run finishes as soon as `testd` receives a notification via an UDP packet and reboots the computer.

5.2.4. Monitoring the Execution

As sketched in Figure 5.8, the SCU is connected to two computers. An Ethernet connection to a server and a serial connection (RS232) to a workstation. The Ethernet connection is used for managing the test cases like described in the previous section, but also for sending monitor data from the SCU to the monitor server.

The serial connection in combination with a terminal program was also used for monitoring. But in contrast to the Ethernet, the data collected this way was more of an *informal* nature for the development of the test framework (e.g. debug messages, warnings). The reasons for this are the much higher bandwidth of the LAN (100 Mbps vs. 19200 bps) and the convenient programming interface (socket interface)[6].

Collecting Data

As already stated, the serial connection was used for occasional warnings and debug messages from the SCU. The main source of information, necessary to reconstruct the execution behaviour of the different partitions, are timestamps sent via the LAN. Like

[6] A note on monitoring via the serial connection: This was done via the `printk()` for programs running in kernel space — the scheduler — or the `rt_printk` for the user space LXRT tasks. Both of this calls work in hard real-time on the cost of reliability. They just copy the messages into an output buffer which is printed to the attached terminal the next time the Linux kernel is scheduled. Since the bandwidth of the RS232 is limited this output buffer can fill up very quick, resulting in garbled output. As a matter of fact the data gained this way has to be taken with a grain of salt.

scheduler

```
1  /* resume job x */
2  rt_task_resume(task[x]);
3
4  jobx_0 = rt_get_cpu_time_ns();
5
6  rt_sleep(task_exec_time[x]);
7  /* job x is scheduled now */
8
9  /* wake up */
10 jobx_3 = rt_get_cpu_time_ns();
```

job x

```
1  jobx_1 = rt_get_cpu_time_ns();
2
3  /* execute workload */
4  entry_point();
5
6  jobx_2 = rt_get_cpu_time_ns();
7
8  /* give up CPU */
9  task_suspend();
```

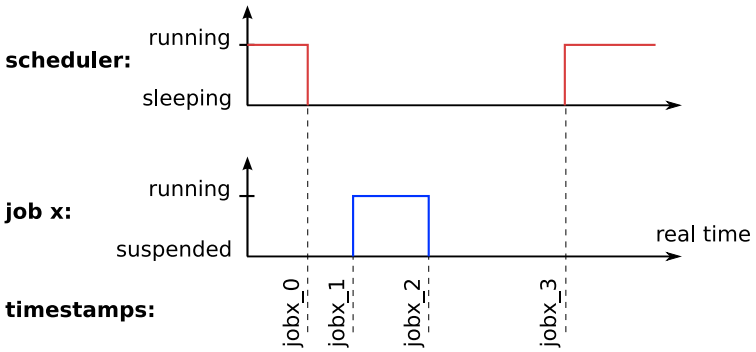


Figure 5.10.: Timestamps for one job

shown in Figure 5.10, a timestamp is taken by the scheduler right before it schedules the next job. As soon as this next job is executed it saves another timestamp prior to entering its `entry_point()`. The next timestamp is taken after the job returns from this function. As soon as the scheduler is running again it saves the last timestamp to complete the timing information about the execution of this particular job in this particular scheduling round. The two timestamp recorded by the job are stored in a dedicated shared memory available to the job and the scheduler.

The function use to get these timestamps is the RTAI function `rt_get_cpu_time_ns()` which returns the current CPU time in nanoseconds into a 64 bit variable. The overhead for this function call is approximately 5 μs.

Besides the four timestamps for each scheduled job, one additional timestamp is taken at the beginning of each scheduling round providing a reference time for this specific round.

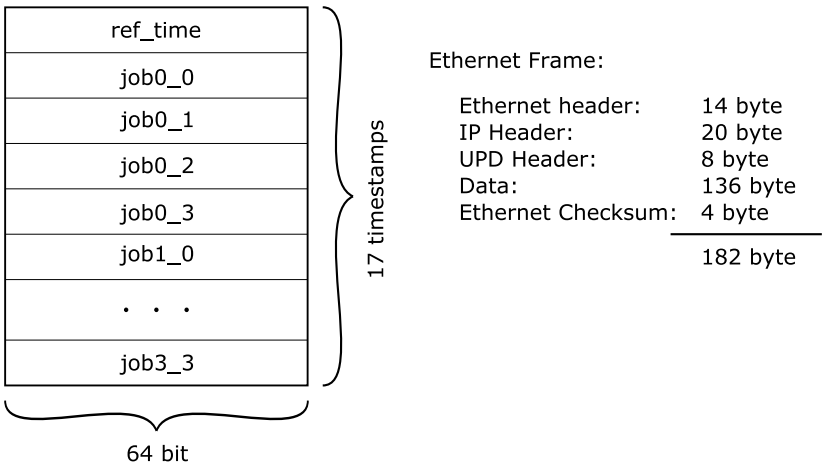


Figure 5.11.: Payload of one UDP monitoring packet.

Sending Data

The timestamps collected in one round are copied into a single UDP packet and sent at the beginning of the next round in the timeslot usually reserved for the BCU interconnection (see Figure 5.6). For the sake of simplicity, monitoring data is always sent for four jobs even if there are less than that (unused timestamps are set to 0). Together with the reference time, each round

$$(1 + 4 \cdot 4) \cdot 64 \text{ bit} \cdot \frac{1 \text{ byte}}{8 \text{ bit}} = 136 \text{ byte}$$

136 byte payload have to be sent. The default length for one round is 2 ms, which leads to a required net bandwidth of 544 kbps. This would be no problem for the underlying 100 Mbps Ethernet. Nevertheless, performance issues on the sending side showed up. The reason for this is the limited send timeslot. The first attempt, using the *kernel internal socket interface*, failed, since the required sending time was around 2 ms. A working solution[7] with a mean send overhead of 190 μs was implemented by a kernel module that extends the Linux kernel with a

```
1 void udp_log(uint8_t *buf, int len);
```

function. This function takes a buffer as argument, and creates a complete Ethernet frame with all the required headers and checksums and the content of the buffer as data payload (Figure 5.11). The target port as well as the MAC and IP address (of the

[7] Although this implementation worked most of the time with the PCMCIA cardbus network card, occasional DMA problems occurred, which caused lost packets. The switch to one of the built in network interfaces on the Soekris board solved this.

monitor server) is hard coded inside this kernel module. Afterwards this raw Ethernet frame is directly handed over to the Ethernet driver. Details about network drivers in the Linux kernel are described in chapter 14 of [RCKH05].

Storing Data

On the monitor server, a monitoring daemon (`monitord`) is running. It establishes a connection to the MySQL[8] database server, also installed on this machine, and creates a new table `test_name_raw`. This table has 17 columns of type `BIGINT` for storing the raw 64 bit timestamps and an additional column `count` for keeping track of the number of packets arrived.

After creating this table it starts listening on port 12340 for incoming monitor UDP packets. Each of these packets is stored into a row of the current table. The monitor daemon closes the database connection and quits, as soon as it receives a `SIGINT` signal (usually sent by the `do_testing` script).

5.2.5. Analyzing the Monitoring Data

Analyzing the gathered monitoring data fullfills two purposes: First, crashes are detected. A crashed job can be recognized because it is not able to update its timestamps any more. Second, by doing a statistical evaluations on different time durations, unusual patterns which indicate a violation of the temporal partitioning can be recognized.

The analysis program is written in Phyton and uses the Numeric[9] and Matplotlib[10] extension libraries. A simplified version of the main `analyze.py` script looks like this:

```
1 import config                              # read configuration file
2 test_list = testing.get_list(config.analyse)  # get test objects
3
4 for test in test_list:
5     if not test.data_available():          # check if data is available
6         print 'No data available for test ' + test.name
7         continue
8     print 'analysing data for test ' + test.name + '...'
9     test.analyse(verbose = True)           # analyse data
```

Like in the `do_testing` script, a config file is used to define which test cases will be analyzed. The analysis function itself performs the following steps:

[8] Open source database; `http://www.mysql.com`
[9] Scientifiy computing with Pyhton; `http://numeric.scipy.org/`
[10] A Python 2D plotting library; `http://matplotlib.sourceforge.net/`

(1) Check for **lost** or **duplicated packages**. This is done to detect major problems in the monitoring and was especially helpful in the earlier stages of development when there have been performance problems on the sending side.

(2) **Normalize timestamps**. So far, the timestamps are still some absolute timestamps in *ns*. A sample row from the table `test_name_raw` might look like this:

count	ref_time	job0_0	job0_1	job0_2	...
10679	26676693374	26677226403	26677275013	26677284232	...

By subtracting the reference time (timestamp at the start of a new scheduling round) from each timestamp and converting the resulting time differences into μs, the following representation of the same row can be found:

count	job0_0	job0_1	job0_2	job0_3	job1_0	job1_1	job1_2	...
10679	548	595	604	781	807	855	1073	...

Additionally, the first 500 rows are dropped to make sure that no initialization effects are presented in the data set any more. This new representation is stored inside a new table, named `test_name`.

(3) **Crash test**. Per definition, a job is marked as crashed if does not update its timestamps for more than 100 times in a row. This is a heuristic since it's not possible to distinguish a crashed from a very slow job just by looking at the timestamps. Actually, it is possible to distinguish a crashed job from one, that, for example, got stuck in an endless loop, by looking at the terminal output monitored via the serial connection to the SCU. A crashed job causes a LXRT warning and does not show up any more in `/proc/rtai/scheduler`.

In other words, a job must have finished the execution of its `entry_point()` function in no more than 100 of its timeslots[11] . Otherwise it is marked as crashed and excluded from the following statistical analysis.

(4) **Statistical analysis**. The general timing analysis calculates statistical data for the time duration the scheduler sleeps (`job`x`_3 - job`x`_0`) and for the time duration a job executes its workload (`job`x`_2 - job`x`_1`). It calculates the *minimum*, the *maximum*, the *mean*, the *standard deviation* and the *maximum deviation* of this time durations.

The procedure above is the standard procedure done for each test listed in the config file. Due to object oriented design it's very easy to create test specific classes, derived from the default `Test` class. This can be used to perform additional, specialized analysis for different tests.

A sample output of the default analysis looks like this:

[11] In the default scheduling setup with a round duration of 2 *ms*, this corresponds to 200 *ms*.

```
analysing data for test cpu_1...
Performing lost and duplicated package test...
     Overall: 0 out of 81937 missing (0.0000 %)
Performing data normalization...
     normalized data in table cpu_1
Performing crash test ...
     job 0 ok
     job 1 crash detected at 1001
     job 2 ok
Performing general timing analyse...
     cpu_1 scheduling:
           min      max     mean  std_dev  max_dev
     ------------------------------------------------
           225      248      238      1.4     13.3
           321      350      325      0.7     24.4
           322      362      331      1.6     30.8

     cpu_1 durations:
           min      max     mean  std_dev  max_dev
     ------------------------------------------------
             7       11        9      0.5      2.5
          job 1 crashed, no data available
             6       11        9      0.6      3.2
```

5.2.6. Additional Checks for Spatial Partitioning

The monitoring framework described in this chapter makes it possible to gather detailed information about the execution of the scheduled jobs and the scheduler itself. It's very well suited to recognize violations of temporal partitioning. To a certain degree it's also possible to use this framework to identify problems with the spatial partitioning. This is done in two test cases in Section 5.3.

This section introduces a solution for explicitly checking the validity of spatial partitioning. This is done by calculating checksums over different *read-only* portions of the private memory area of the jobs. A checksum that has changed indicates a violation of spatial partitioning.

A sample implementation of this idea has been done for this thesis in the task wrapper of the jobs and can be activated at compile time. Each time after the `entry_point()` function of a job returns, four different checksums are calculated and compared to the value of the previous round. The four read-only memory areas this checksums belong to, are:

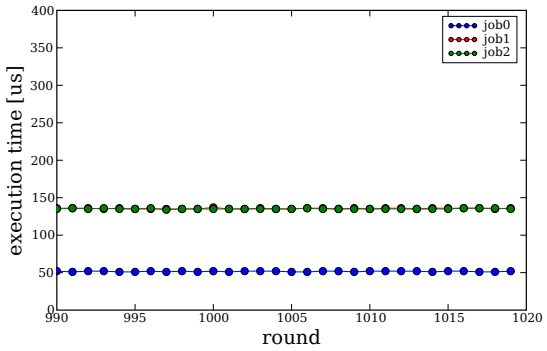


Figure 5.12.: Comparison of execution time with and without checksum calculation.

(1) The `main` function of the task wrapper.

(2) A 400 byte array on the stack.

(3) A 400 byte array in an allocated chunk of heap memory.

(4) A 400 byte array in a RTAI shared memory.

The algorithm used for calculating the checksum is a very simple one. It's just the lower 32 Bit of the sum of the corresponding memory area:

```
1  /* Calculates a simple 32 bit checksum for the memory area starting
2   * at `start` with a length of `length` bytes.
3   */
4  uint32_t calc_checksum (void *start, size_t length)
5  {
6      uint32_t *p = start;
7      uint32_t checksum = 0;
8      int i;
9      for (i = 0; (i * sizeof (uint32_t)) < length; i++) {
10         checksum += *p++;
11     }
12     return checksum;
13 }
```

Figure 5.12 illustrates the overhead that is caused by introducing these checksums. It shows the execution times of three jobs. All three of those jobs are execution the same workload, but one job — job0 — is running without the checksum calculation. Its execution time does not include the overhead for checksum calculation and comparison and is approximately 90 μs lower.

5.3. Testing the Partitioning

This section concentrates on testing the partitioning capabilities of the primary OS using the original DECOS prototype setup. The scheduled jobs are usual RTAI-LXRT tasks.

First of all, some basic RTAI-LXRT functionalities have to be explained in more detail in the following Section 5.3.1 to prepare the reader for the remaining sections. This includes the memory management of RTAI-LXRT and the RTAI naming service.

Since partitioning is always about shared resources, there is a brief tour through the hardware components of the used Soekris computer (Section 5.3.2 – 5.3.4). It features an analysis of the source code and the provided LXRT API to see how those hardware components can be used from within a LXRT job and where problems with partitioning might occur. Section 5.4 checks the hypotheses gained from this analysis by introducing multiple test cases.

5.3.1. Selected Features of RTAI-LXRT Linux

Memory Management

Since all jobs reside on the same main memory of the Soekris computer, memory is a critical point for the partitioning and will be handled here at first and in detail. Most of the memory issues are in the field of spatial partitioning but as will shown in Section 5.3.2 there are temporal concerns too.

The fact that RTAI/LXRT has to provide hard real-time service has impacts on the memory management in two ways:

(1) All physical address pages associated with the address space of a process have to stay in memory. If not, a page fault might occur when a certain address is accessed. Since it takes time to load that page into memory again, hard real-time is no longer given. Linux provides the `mlockall` [12] system call to ensure this. This system call locks all pages into the memory that are mapped into the process's address space at the time the call is done and at some time in the future.

Unfortunately this solves the real-time problem only for static memory. So it's perfectly suited to lock the program code itself into memory and to some extend also for the stack. RTAI offers the `rt_grow_and_lock_stack()` macro which expands the stack by an estimated[13] sufficient size and calls `mlockall` afterwards.

(2) For dynamically allocated heap memory, the problem is that allocating memory via the Linux kernel is not time bounded.

[12] see also: `man 2 mlockall`

[13] It's the job of the programmer to estimate a sufficient size.

Therefore RTAI offers dedicated functions[14] for hard real-time memory allocation that have basically one thing in common: They only work on an already allocated chunk of memory that has to be initialized as a *pseudo heap*.

Allocating and freeing memory on this heap is managed by a built-in RTAI dynamic memory allocation service[15]. This service is in pure software and offers great flexibility with respect to e.g. configurable page size but does not use the MMU for its internal memory protection.

No assumptions are made about the location of this underlying memory. It's even possible to initialize this real-time heap on top of a memory chunk on the stack. Hard real-time can only be guaranteed if this heap is initialized as non-expandable.

Besides the discussed points regarding hard real-time above, there is one more detail of RTAI-LXRT memory management that deserves a closer look. RTAI-LXRT Linux tries to offer a symmetrical API for both RTAI tasks running in kernel space and LXRT tasks running in user space. As a matter of fact there should be no differences to an application programmer if she is writing a RTAI kernel module or the same task as a LXRT process. Therefore seamless crossing of the border from kernel space to user space and vice versa is required. Regarding memory, RTAI-LXRT uses the `mmap(2)` system call to map chunks of kernel space memory into the user space (illustrated in Figure 5.13, detailed explanation in chapter 13 of [RCKH05]).

In fact, every heap or shared memory that is opened in a LXRT task is allocated in kernel space and mapped to the user space task. This mechanism ensures transparent sharing of memory among kernel-space RTAI tasks and user-space LXRT tasks. It is not only very convenient and effective but also quite save regarding spatial partitioning as will be demonstrated in the tests runs dealing with memory later on in Section 5.4.

Naming Service

RTAI offers a naming service which makes it possible to assign a name, consisting of 6 ASCII characters, to an arbitrary object reference. E.g. to a memory area, a semaphore or a task. Using the `nam2num` call, this name can be converted to a 32 bit[16] id that is used internally.

This naming service makes sharing resources among different RTAI-LXRT tasks possible. A tasks just needs to know the name of an object for accessing it. Although this is convenient for cooperating tasks, sharing of objects which should be private property of partitions violates spatial partitioning. Unfortunately most of the time

[14] `rt_halloc()`, `rt_malloc` and the lower level function that is called by the former two, `rtheap_alloc`
[15] source file: `rtai-core/malloc/malloc.c`
[16] Depending on the hardware architecture; but on the Soekris computer it's 32 bits.

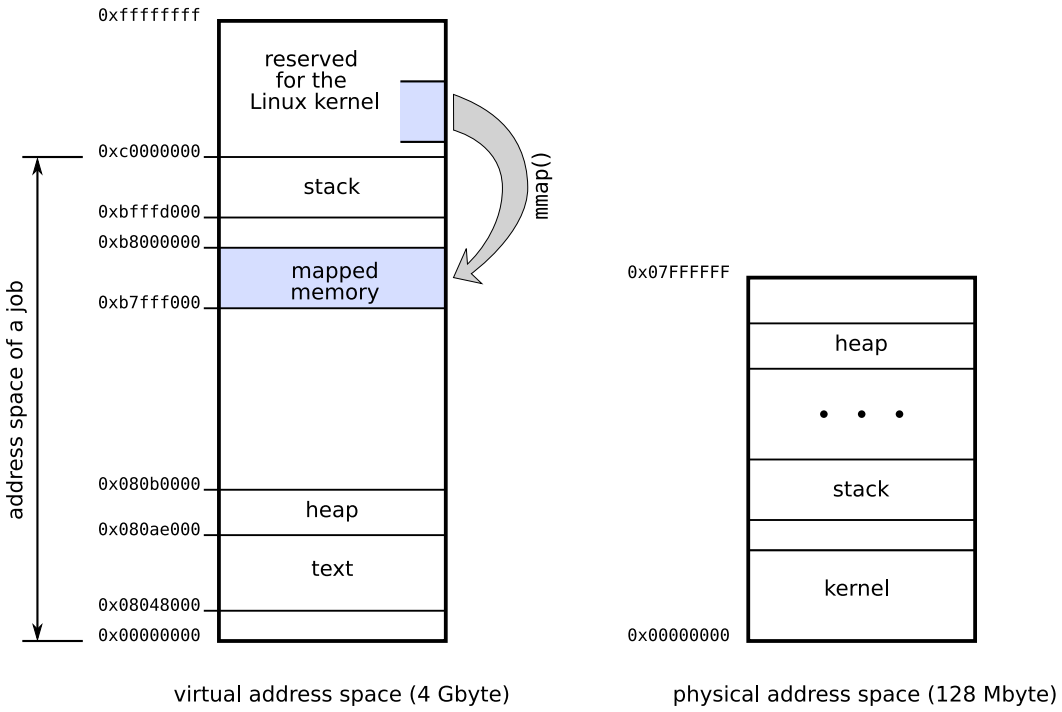


Figure 5.13.: Virtual and physical address space

it's not possible to bypass this naming service since such a name/id has to be supplied for each new shared memory, task, heap and so on.

If the name is kept secret one might look at it like a 32 bit key to protect the privacy of the object. But as soon as the name is known to a partition it can access the resource like it's real owner. Even if malicious intention is not within the fault hypothesis this is a risk since names like "SHM", "SEM" or "RTHEAP" are just to prominent and might lead to name clashes.

HYPOTHESIS 1 *The RTAI naming service has no support for permission checking at all. All registered objects can be accessed in all ways by all LXRT tasks that know the name.*

There are just a few points in the code where RTAI explicitly checks the id of an object before performing the requested operation. One example can be seen in the following snippet. It's executed during the closing of real-time heaps and prevents closing the global heap.

```
1 /* from file rtai-core/ipc/shm/shm.c */
2
3 if (!rt_drg_on_name_cnt(name) && name != GLOBAL_HEAP_ID) {
4     ...
5 }
```

5.3.2. Memory

Software and Private Data

As soon as an LXRT task is started, the compiled and executable program is loaded into the memory just like a usual Linux process. In the initialization stage, an additional kernel module, serving as agent and executing the native RTAI calls in its context, is loaded into the memory too. Those program parts, also called *text*, have to be protected from the access of other partitions.

Crucial for the spatial partitioning is that each user space process in Linux (and therefore each job) has its own virtual address space like illustrated in Figure 5.13. It shows the text region on the bottom, the memory reserved for the kernel on the top, and the regions for stack and heap in between.

Each memory address seen by a process is mapped into the real physical memory address by the operating system. A process ca not just access an arbitrary physical memory address. A special purpose hardware to support this mapping or virtual memory in general is called Memory Management Unit (MMU)[17].

[17]The inclusion of an on-chip MMU like Intel did it in the i386 architecture was a major step towards making todays modern multitasking operating systems possible [Mil90].

The Soekris computer, executing the primary OS, has a built-in MMU and supports these virtual addresses. The Linux kernel, controlling the MMU tables, takes care that no memory region of a partitions virtual memory is mapped to a physical memory address used by another partition.

Details about virtual memory, MMUs and the different hardware protection modes of the i386 architecture can be found in books about operating systems, like [Sta01].

HYPOTHESIS 2 *The text and the stack region as well as heap memory allocated via the standard Linux system calls are protected by the MMU from being overwritten by any other partition.*

Note that the real-time dynamic heap memory like implemented in RTAI is a bit special and will be handled in detail in Section 5.3.2.

Communication or Shared Memory

As introduced in Section 4.2.2 on page 29, spatial partitioning has also to ensure the integrity of data in transit. As far as partitions are concerned, this maps to their input and output ports to the SCU (Section 3.2.1). Since in the DECOS prototype cluster the functionality of the SCU is implemented in software by using a dedicated partition (see Figure 5.4), those communication ports have to be implemented via a sort of *inter-partition communication* or *inter-process communication* (IPC).

RTAI Linux offers multiple IPC mechanisms like shared memory, message queues, real-time FIFOs, mailboxes, mutexes, semaphores and conditional variables. All of these mechanisms can be used to transfer and share data between tasks and processes in both the real-time (RTAI and LXRT) and Linux user-space domain [MDP00].

Shared memory is the IPC mechanism that was chosen to be used in the DECOS prototype cluster. Its advantage is simplicity and flexibility. The usual difficulty when using shared memory, the explicit synchronization of accessing tasks, is done implicit by the time-triggered static scheduling. Following the general rule to keep the primary operating system as simple as possible, all other means of IPC should not be available and will not be discussed.

Opening a new real-time shared memory from a LXRT task is done via the `rt_shm_open()` call. This function takes an argument specifying the method for the memory allocation[18]. A reference to this new allocated memory is stored inside the RTAI naming service and finally the memory chunk is mapped to the virtual address space of the LXRT task.

HYPOTHESIS 3 *As long as the* name *of the shared memory is not known to a LXRT task it has no possibility to access the memory region.*

[18] See [RCKH05, chap. 8] for the difference of `vmalloc` and `kmalloc`.

Communication partners can join by using the same `rt_shm_open()` call. RTAI just increases a reference counter and performs the mapping to user-space. By using the `rt_shm_close()` function this reference counter is decreased and the memory is freed as soon as its reaches zero.

HYPOTHESIS 4 *If the* name *of a shared memory is known to a LXRT task it cannot only violate spatial partitioning by writing into the memory but also by closing/freeing the shared memory.*

Heap Memory

There are two basic methods for a LXRT task to acquire dynamic memory. If a job uses the Linux `malloc` system call to reserve heap memory it will be as safe, in regard to spatial partitioning, as the text section and the stack. The downside of this approach is that such a call cannot be done in hard real-time. RTAI offers a solution to this problem by introducing real-time heaps.

The most basic real-time heap provided by RTAI is the *global heap*. This heap memory region is allocated by default by RTAI and, after calling the `rt_global_heap_open()` function, can be used by every LXRT task. Before looking at spatial partitioning issues regarding this global heap it's possible to generalize a bit. The global heap is just a *group heap* with a special name — "RTGLBH". So problems found with the global heap will also apply to group heaps and vice versa. The snippet below shows that opening the global heap just calls the usual function to open a new heap with the global heap ID as first argument:

```
1 /* from rtai-core/include/rtai_shm.h */
2
3 #define GLOBAL_HEAP_ID  0x9ac6d9e5  // nam2num("RTGLBH");
4         ...
5 #define rt_global_heap_open()  rt_heap_open(GLOBAL_HEAP_ID, 0, 0)
```

The memory for the heap is allocated and initialized in kernel space and mapped to each LXRT task that opens it. Reserving and freeing memory is done by the memory allocation service without any additional involvement from the MMU. As a matter of fact, no segmentation fault is triggered if a partition writes into memory outside of its allocated area[19].

[19] Actually it's not very surprising that RTAI is vulnerable to such an attack (or programming mistake). Standard Linux also allows heap and stack overflows to a certain amount without complaining, which is the reason for a lot of security problems. In order to detect such memory handling bugs (especially the famous off-by-one errors) one can try to run the jobs in a hardened Linux environment that adds extra memory checks — like PaX (`http://pax.grsecurity.net`), a patch for the Linux kernel — or, use a special memory debugging tool like Valgrind [NF04] (`http://valgrind.org`) to detect such bugs during development.

HYPOTHESIS 5 *As soon as a partition opens a group heap, it's able to read and write on every address within this memory region, no matter if it's in use by another partition or not.*

The following snippet shows that in fact, a group heap is always placed on top of a shared memory, allocated with `rt_shm_alloc` (line 6) and initialized with `rt_set_heap` afterwards (line 8):

```
1  /* from rtai-code/ipc/shm/shm.c */
2
3  void *rt_heap_open(unsigned long name, int size, int suprt)
4  {
5          void *adr;
6          if ((adr = rt_shm_alloc(name, ((size - 1) & PAGE_MASK) +
7              PAGE_SIZE + sizeof(rtheap_t), suprt))) {
8                  rt_set_heap(name, adr);
9                  return adr;
10         }
11         return 0;
12 }
```

Memory Hierarchy

So far only spatial partition problem regarding the shared resource memory have been discussed. Section 4.3.2 on page 31 already introduced that a memory hierarchy with a main memory and a cache can lead to a violation of temporal partitioning.

The AMD processor used on the Soekris target has a built in 16-Kbyte write-back cache [AMD], so in theory it will be possible to observe this phenomena. In this thesis, this effect has not been considered in detail but it looks like the influences of other partitions on memory performance is negligible. A possible explanation for this is, that by the time a partition is scheduled again after a whole scheduling cycle, the cache does not contain much entries of this partition any more, no matter how the other partitions look like.

5.3.3. Private Devices or Actuators

Basically there are several possibilities for a jobs to use I/O. The first one is to use a standard Linux device driver by accessing the device entry in the `/dev` directory. As an alternative, the primary OS can offer an API to the I/O device and can use Linux kernel internal functions to access devices itself[20].

[20] This is the method used for sending the monitor packages (see Section 5.2.4).

In the first case, private devices can be implemented by using the standard Unix file permissions for the device entry in the `/dev` directory. This approach does not help in the prototype implementation since all jobs run as root and have access anyway.

If the primary OS offers the API to I/O devices, permission checking has to be implemented there. In fact, access to some of the general purpose I/O pins of the Soekris board is provided in the prototype. This was done for debugging purpose only and therefore no protection of any kind is implemented. Section 6 shows the concepts how this I/O pins can be assigned to separate partitions, simulating private I/O devices.

A completely different form of I/O access is memory mapped I/O instead of using a call interface. On a closer look, the introduced monitor framework does this. The task wrapper of the partitions writes the timestamps into a dedicated shared memory area shared with the primary OS. This timestamps are sent later on through the network device.

5.3.4. CPU

Influencing the scheduled access to the CPU is probably the first thing that comes in one's mind in the context of temporal partitioning. The most basic case is a job executing some kind of endless loop or just a function with a longer execution time than the remaining time in the partitions timeslot. In this case the task wrapper of the partition does not call the suspend call to give up the CPU[21].

HYPOTHESIS 6 *Since the scheduler task has an higher priority than all the partitions it does not matter if a partition tries to block the CPU. It will be preempted anyway and will not affect other partitions.*

This leads immediately to the next hypothesis:

HYPOTHESIS 7 *If a partition can set its own priority above the scheduler task of the primary OS it can block the CPU indefinitely.*

Other things that might force the CPU to execute other code that the partition job that should be executed at time according to the static scheduling might be *interrupts*. Those interrupts can be *hardware interrupts* or *software interrupts* also called *traps*.

HYPOTHESIS 8 *The ADEOS layer used by RTAI Linux catches all interrupts. The interrupt service routines (ISRs) will be executed in the idle time reserved for the Linux kernel and will not influence temporal partitioning.*

[21] See the pseudo code of the task wrapper in Figure 5.7.

Scheduling

job	min	max	mean	st.dev.	m.dev.
job0	324	378	330	4.1	47.1
job1	320	355	328	1.8	26.4
job2	320	349	324	1.0	24.8

Execution times

job	min	max	mean	st.dev.	m.dev.
job0	49	63	51	0.3	11.0
job1	49	78	51	0.5	26.7
job2	48	53	51	0.5	3.5

Job switching times

min	max	mean	st.dev.	m.dev.
20	36	27	0.7	8.9
21	42	27	0.7	14.3

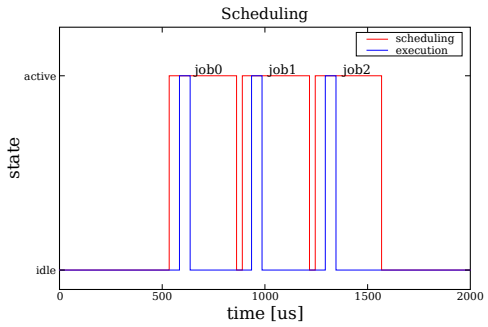


Table 5.2.: The *default* test featuring three jobs

Contrary to the other hypotheses, this one is not verified via test cases in Section 5.4. This is a fundamental functionality required for RTAI real-time Linux and is documented for example in [BD00].

5.4. Test Runs

This section lists the various test cases that have been run. If not stated otherwise a basic setup is used. It features a cycle period of 2 ms and three scheduled jobs, each of them with a timeslot of 300 μs.

Table 5.2 shows the scheduling as well as some statistical data for the default configuration. The depicted data was gained from an one hour test or approximately $500 \cdot 3600 = 1800000$ rounds. Data from the first 500 rounds has been removed.

The upper table deals with the length of the time interval dedicated for the various partitions. It's the length of the time interval starting when the scheduler releases the CPU for the next partition until it is resumed again. In other words, it's calculated as the difference between time stamp 3 and 0 as defined in Section 5.2.4. The second table analyzes the execution times of the partition jobs (time stamp 2 minus 1, see again Section 5.2.4). Finally, the third table shows some statistical data about the time it takes for the scheduler of the primary OS to switch to the next job.

All three jobs perform a simple arithmetic calculation. This default calculation will

be used for the first and the third job in all test cases if not stated otherwise.

For the following test cases, the second job is the "misbehaving" one. It will return immediately the first 999 times it is executed and starts with the real test, usually a attempt to break partitioning, in round 1000.

5.4.1. Virtual Memory

This set of test concentrates on the virtual memory address space that is available to the partitions. The goal is to show that it's only possible to access virtual memory assigned to the partition and not arbitrary areas.

Test 1: *mem_1*

Description This test features a job that tries to derefer a NULL pointer (address 0x00000000).

Results Causes a segmentation fault trap that is caught by the ADEOS layer. The job is terminated while all other jobs keep executing in their usual schedule. As shown in Table 5.3, the mean execution time for those remaining jobs does not differ at all from the default value. Furthermore, memory checksums don't indicate a violation of spatial partitioning.

Interesting is that the schedule changes because of new partition/job switching times. The graph on the right side clearly shows a significant change at around round 1000. Furthermore the table containing the switching times shows that the mean switching times drop from 27 μs to 22 respectively 16 μs. This has nothing to do with the reason for the crash, the invalid memory access, but with the implementation of the job switching.

Test 2: *mem_2*

Description A job that tries to access virtual memory that is mapped to the kernel memory (address 0xc00000000, see Figure 5.13).

Results Just like in the previous test *mem_1*, a segmentation fault is raised and the offending job is terminated. Regarding the temporal partitioning the same effects as in *mem_1* occur.

Test 3: *stack_1*

Description Test *mem_1* and *mem_2* already showed that arbitrary access to virtual memory addresses, not mapped to physical ones by the MMU, will lead to a segmentation fault trap and the termination of the job.

This test will try to access a different memory area in a slightly different, more indirect way. In its entry point function, the job1 overwrites the return address

Scheduling

job	min	max	mean	st.dev.	m.dev.
job0	321	341	333	1.0	12.2
job1	316	361	320	1.5	40.3
job2	321	356	325	1.3	30.8

Execution times

job	min	max	mean	st.dev.	m.dev.
job0	49	53	51	0.5	2.5
job1 crashed, no data available					
job2	49	53	51	0.4	2.8

Job switching times

min	max	mean	st.dev.	m.dev.
13	30	16	0.7	13.4
19	37	22	0.9	14.5

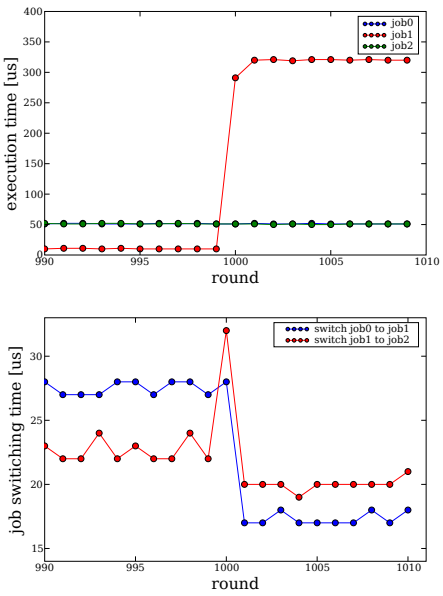


Table 5.3.: The *mem_1* test

located on the stack. This return address usually points back to the text segment of the task wrapper right after the jump into the entry point function. As soon as the job return from the function call, the partition continues its execution from this point. In other words: This test tries to access an invalid memory address, not by trying to read from or to write into, but to execute code located at this address.

In this test the return address is overwritten with with a different value, a common technique used in a lot of security exploits. Details about the layout of the stack in Linux can be found in [One96].

Results The job crashes with a segmentation fault after the frame pointer and the return address is overwritten. In other words: This method, just like in the previous tests, does not allow access to arbitrary memory areas either, although the access type is not read or write but execute. Since no memory is overwritten all checksums stay the same. The other jobs keep running, the influence on the scheduling is shown in Table 5.3.

Test 4: *stack_2*

Description This test is very similar to *stack_1* and is interesting because it shows the granularity of memory protection provided by the MMU. It also overwrites the stack, but instead of increasing the address to be overwritten is decreased

in each call. Like shown in Figure 5.13, the stack starts at address `0xbfffffff` and grows downwards.

The size of the reserved stack is 8000 bytes pre-reserved in the task_wrapper (already mentioned in Section 5.3.1) combined with the variables already on the stack at that time.

Results Since the MMU is only able to keep track of whole-numbered pages, the entire size is 3 pages, or 3 times 4096 bytes[22] . The job crashes with a segmentation fault as soon as it tries to write below the lower stack limit (at `0xbfffd000`) with the same effects on the scheduling as described in the previous tests.

Like in the previous test cases, the MMU successfully prevents the access to memory not mapped into the virtual address space of job1. Memory checksums calculated by the other jobs do not change.

5.4.2. Shared Memory and Real-Time Heap

This section deals with shared memory as preferred way of inter-process communication. Real-time heaps, as implemented in RTAI Linux, are based on top of a shared memory and are handled here too

Test 5: *shm_1*

Description This tests shows how easy it is for job1 to bypass spatial partitioning by writing into a shared memory belonging to job0. Obviously this is usually a feature and not a bug, but not in our special case of a partitioning OS. Like explained in Section 5.2.4, job0 shares a shared memory called "SHM0" with the primary OS for the exchange of the monitoring timestamps. A well behaving job1 uses only "SHM1" to present its timestamps to the OS but here it will also open "SHM0" and write zeros into the memory.

Results Since the content of the shared memories used for collecting the timestamps from one round is copied into the UDP monitoring package at the beginning of the next scheduling round, job1 is able to sabotage the data sent by job0. In this special setup job1 cannot disturb the monitoring data sent by job2 since it's scheduled later.

Table 5.4 presents raw timestamps (not normalized) received as monitor data for job0. It nicely shows how the timestamps job0_1 and job0_2, indicating the execution time of the job, are always 0 after round 1022. The sharp eyed reader will immediately notice two things: First, why does it take 20 rounds up to round 1020 until the timestamps are zero the first time? And second, why are there non zero values for round 1021?

[22]The default size for a memory page on the i386 architecture is 4096 bytes.

Execution times

job	min	max	mean	st.dev.	m.dev.
job0	0	53	0	1.8	52.9
job1	9	327	10	2.3	316.1
job2	48	53	51	0.4	3.1

Raw time stamps received for job0

count	job0_0	job0_1	job0_2	job0_3
1018	7382563402	7382609497	7382660621	7382885231
1019	7384594110	7384644396	7384695520	7384923482
1020	7386604703	0	0	7386938266
1021	7388583449	7388637087	7388689049	7388913659
1022	7390581471	0	0	7390911681
1023	7392564408	0	0	7392894618
1024	7394565782	0	0	7394895992

Table 5.4.: The *shm_1* test

The reason for the first point is that opening the shared memory requires multiple Linux syscalls — 3 to be exact — and therefore takes some time. Syscalls and their influence on other jobs will be investigated in detail in Section 5.4.4. The reason for the second phenomena is unknown. But it looks like some kind of race condition since it does not happen in each test run.

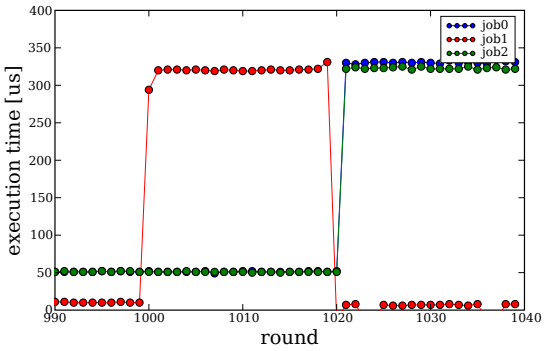
Note that the checksum calculation presented in Section 5.2.6 would detect this violation as well, albeit the checksum calculation doesn't work well on this shared memory containing the timestamps, since its content changes every round. The same test setup, using a different shared memory which is used read-only by job0, clearly results in a changed checksum indicating a violation of spatial partitioning.

Test 6: *shm_2*

Description Test *shm_1* already showed a nice way to bypass spatial partitioning of shared memory used as communication channel. This test uses an even more radical approach: job1 frees the shared memory SHM0, making it unavailable for job0.

Results

First of all, it's no problem to call the `rt_free_shm()` function for a shared memory even if it was never opened by the particularly job. RTAI only does reference counting and does not keep track who opens and frees a shared memory.



count	job0_0	job0_1	job0_2	job0_3	job1_0	job1_1	job1_2	job1_3	...
1019	556	605	656	885	905	905	1232	1232	...
1020	552	605	657	885	907	1915	1901	1263	...
1021	546	546	874	874	890	1924	1910	1213	...
1022	528	528	858	858	875	1915	1923	1198	...
1023	532	532	862	862	880	1921	1928	1200	...

Table 5.5.: The *shm_2* test.

Each time a shared memory is freed the reference counter is decreased by one and as a matter of fact there are no problems at all if SHM0 is freed only once since `SHM0` has been opened by job0 and the scheduler. Freeing `SHM0` twice results in a spectacular crash. Spectacular since the scheduler (running in kernel space) accesses invalid memory areas too as soon as it tries to read from `SHM0`. Table 5.5 shows a few lines of the table with the normalized timestamps which indicate that the behaviour of the execution environment becomes rather unpredictable. Job0 and job2 crash at round 1021. And starting from round 1020, the timestamps job1_1 and job1_2 are bigger than job1_3 although they are taken earlier!

The checksum calculation cannot detect this violation since the algorithm crashes itself when trying to access the freed memory.

Test 7: *heap_1*

Description Like introduced in Section 5.3.1, RTAI offers the feature of real-time heaps. They are implemented on top of a shared memory and therefore can be shared among different jobs too. It should already be clear from previous tests, that by knowing the RTAI name of the heap a job has full access to a heap. When using the provided interface to malloc and free memory chunks this is no problem since the memory manager takes care that no data is lost or overwritten.

In this test, job0 and job1 share a group heap with a size of 128 kbyte. Job0 allocates two times 4 bytes on this heap. The first time with `rt_named_halloc()`

which allows to assign a name to the newly allocated memory chunk, and the second time with the `rt_halloc()`. The first case is not very interesting, but the second because of this quote from the RTAI documentation: *Allocate a chunk of a group real time heap in kernel/user space. Since it is not named there is no chance to retrieve and share it elsewhere* [RTA].

Job0 initializes it's two memory areas with `0xaaaaaaaa` and `0xbbbbbbbb`. Additionally this job warns if those memory regions have been altered since the last call via stdout.

Job1 allocates a chunk of memory on this group heap too[23] and starts to write `0xcc` into the memory. Each round the address written to is increased by one byte.

Results

Job1 is able to overwrite the whole 128 kbytes of the heap and crashes after a few minutes (in round 140216) after finally hitting the limit of the reserved memory protected by the MMU.

If checksum calculation is done over the real-time heap, a warning is printed after each round to indicate an unexpected change of the checksum. A better visualization of how the heap is slowly overwritten byte pre byte provides the output of job0 during this test:

```
/ # Initial: value1 = 0xaaaaaaaa, value2 = 0xbbbbbbbb
Warning: value1 changed from 0xaaaaaaaa to 0xaaaaaacc
Warning: value1 changed from 0xaaaaaacc to 0xaaaacccc
Warning: value1 changed from 0xaaaacccc to 0xaacccccc
Warning: value1 changed from 0xaacccccc to 0xcccccccc
Warning: value2 changed from 0xbbbbbbbb to 0xbbbbbbcc
Warning: value2 changed from 0xbbbbbbcc to 0xbbbbcccc
Warning: value2 changed from 0xbbbbcccc to 0xbbcccccc
Warning: value2 changed from 0xbbcccccc to 0xcccccccc

/#
LXRT CHANGED MODE (TRAP, LxrtMode 0), PID = 409
LXRT releases PID 409 (ID: job1).
```

Test 8: *heap_2*

Description In this test job1 closes a real-time heap used by job0. In fact, closing the heap corresponds to freeing the underlying shared memory, as the following code snippet shows:

[23] One round before job0 to get a memory chunk with a lower address.

```
1 /* from rtai-core/includes/rtai_shm.c */
2
3 #define rt_heap_close(name, adr) rt_shm_free(name)
```

Results The results are mostly equivalent to those of test *shm_2*. Again job1 is able to close the heap even without ever opening it before. After the reference counter of the heap reaches zero, the underlying shared memory is freed and job0 crashes the next time it tries to access this memory.

5.4.3. The CPU as Shared Resource

In the following set of tests, job1 tries to influence the quality of service received from the CPU. Since there is no possibility to alter the clock frequency of the CPU the only possibility is to influence the scheduling durations of the CPU to jobs.

Test 9: *cpu_1*

Description In this very simple test case job1 return immediately the during the first 999 invocations and enters an endless loop in round 1000.

Results The results of the this test are shown in Table 5.6 and can be summarized as follows: Although job_1 utilizes the CPU as much as it can it's not able to influence either the execution time of the other jobs nor the partition switching times in a significant way.

Test 10: *cpu_2*

Description RTAI provides the `rt_change_prio()` function that allows changing scheduling priorities during runtime. In the default setup, the scheduler module is running with the highest priority, the Linux kernel, scheduled only during an idle time of 0.15 ms per round, at the lowest possible priority and the LXRT jobs at some priority between those two extremes.

In this test job1 tries to raise its own priority to the highest value. This is done in scheduling round 1000. In round 1005, job1 starts a calculation that takes significantly longer to complete than the length of its time slot.

Results Job1s request to be scheduled with the highest priority succeeds. And as shown in Figure 5.14, in round 1005 job1 is not interrupted any more and is able to block the CPU for much longer than the length of its timeslot[24]. A clear violation of temporal partitioning! The result is a scheduling round much longer than the usual 2 ms. Note that only the activation of the jobs scheduled after job1 is delayed. The execution times will not change because of that.

[24] It would also be possible for this job to block the CPU resource forever.

Scheduling

job	min	max	mean	st.dev.	m.dev.
job0	322	344	331	1.0	12.9
job1	316	347	320	0.7	26.7
job2	322	364	325	1.3	38.2

Execution times

job	min	max	mean	st.dev.	m.dev.
job0	49	53	51	0.5	2.5
job1 crashed, no data available					
job2	48	53	51	0.5	3.2

Job switching times

min	max	mean	std. dev.	max. dev.
21	53	28	0.7	24.2
19	27	22	0.5	4.5

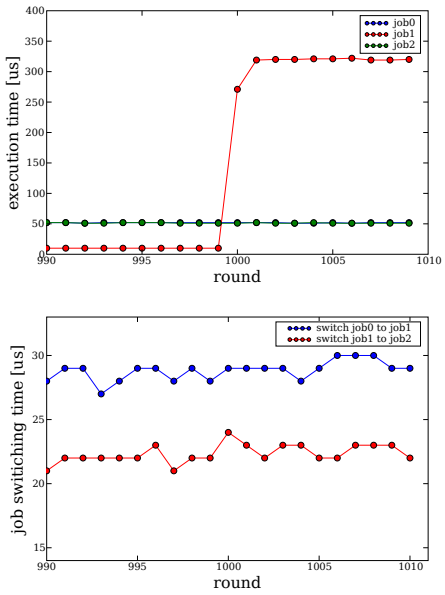


Table 5.6.: The *cpu_1* test

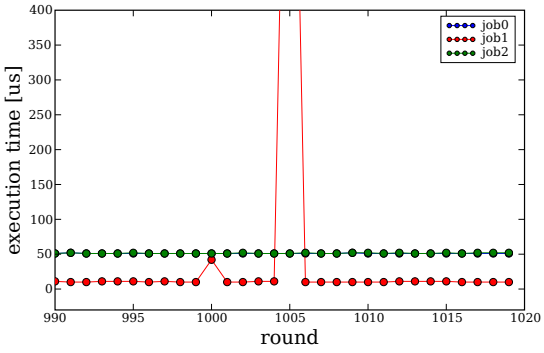


Figure 5.14.: The *cpu_2* test

```
/ # LXRT releases PID 410 (ID: busybox).

The system is going down NOW !!LXRT releases PID 412 (ID: job2).

Sending SIGTELXRT releases PID 408 (ID: job0). RM to all
processes. Sending The system is halted. Press Reset or turn off
power LXRT: REBOOT NOTIFIED -- KILLING TASKS Power down. Kernel
panic - not syncing: Attempted to kill init!
 <4>Adeos: Current domain=Linux on CPU #0 [stackbase=00000000]
    RTAI[cpuid=0]: priority=200, status=0x4, pending_hi=0x0
   Linux[cpuid=0]: priority=100, status=0x0, pending_hi=0x0
```

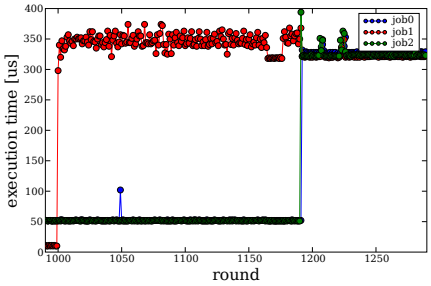


Figure 5.15.: The *syscall* test

5.4.4. Linux System Calls

To demonstrate the potential danger of system calls to partitioning just one, rather dramatic, test case is needed.

Test 11: *syscall*

Description In this test case job1 uses the `execl()`[25] to execute the *shutdown* command.

Results As expected the system is shuting down, stopping the primary OS and all partition jobs. The output of the serial connection and timestamps are illustrated in Figure 5.15.

The kernel panic at the end is not expected but does not really matter any more at this time. Note that for gaining this result, job1 has to run with root permissions.

It should be quite obvious to the reader that also spatial partitioning cannot be guaranteed if the jobs are allowed to use system calls.

The test case above shows a violation of temporal partitioning. Spatial partitioning can be violated by using the `/dev/kmem` device file to write into arbitrary memory regions. Depending on the target address, the checksum calculation would detect this attack.

5.4.5. Conclusions after the Tests

Table 5.7 summarizes the claimed hypotheses and the tests cases supporting them. Table 5.8 shows an overview about the different tests and their influence on partitioning.

[25] `man 3 execl` for further informations.

Hypothesis	Comments
1	The fact that the RTAI naming system does no permission checking can be shown by looking at the source code and is demonstrated by the tests *shm_1*, *shm_2* *heap_1* and *heap_2*.
2	The tests *mem_1*, *mem_2*, *stack_1* and *stack_2* show that the memory protection done by the MMU works like expected.
3	According to hypothesis 2, a LXRT task cannot access memory in the kernel space directly. For mapping it into its address space it has to know (or guess) the RTAI name of the memory region. In theory it could also open the `/dev/rtai_shm` device and guess the correct parameters for the `mmap` system call.
4	This hypothesis is supported by the tests *shm_2* and *heap_2*.
5	Test *heap_1* shows that there is no hardware protection between different memory areas allocated on the same real-time heap.
6	Test *cpu_1* directly shows that this hypothesis is valid.
7	Test *cpu_2* was written to support this hypothesis. The peak in the execution time for job1 in round 1005, illustrated in Figure 5.14, shows that a job can raise its own priority until it cannot be interrupted by the primary OS any more.
8	Not supported by a test case in this thesis but by existing work: [BD00].

Table 5.7.: Arguments for supporting the hypotheses claimed in between page 53 and 57.

	Spatial part. violation	Temporal part. violation
Test 1 — *mem_1*	NO	NO[26]
Test 2 — *mem_2*	NO	NO[26]
Test 3 — *stack_1*	NO	NO[26]
Test 4 — *stack_2*	NO	NO[26]
Test 5 — *shm_1*	YES	NO
Test 6 — *shm_2*	YES	NO
Test 7 — *heap_1*	YES	NO
Test 8 — *heap_2*	YES	NO
Test 9 — *cpu_1*	NO	NO
Test 10 — *cpu_2*	NO	YES
Test 11 — *syscall*	YES	YES

Table 5.8.: Overview about which test showed which kind of partitioning violation.

[26]This tests showed a negligible influence on temporal partitioning.

The first four tests, concentrating on spatial partitioning, show clearly the benefit of using LXRT user-space real-time tasks for the various partition jobs. The fact that only a virtual address room is provided to the different jobs leads to a clear separation of the available memory. On the temporal side those tests only caused peaks in the partition switching times (shown in Table 5.3 on page 60) which can be neglected.

These tests have also been run with the additional checksum calculation introduced in Section 5.2.6 to provide even stronger support for the claim of strictly separated virtual address rooms. No partition violations have been detected by this test runs.

The results of test 5 and 6 shows that using shared memory as a mean of IPC does not guarantee spatial partitioning because of the RTAI naming system. Besides that, it's also possible for a job to crash another one by freeing the shared memory it's using.

The following two tests shows that the results gained from the shared memory tests can be mapped to the real-time heap as well. Again, the RTAI naming system is the weak point.

A job trying to claim the CPU for its own has no chance to break temporal partitioning, as test *cpu_1* shows. Another test inside the CPU category does show a significant effect on temporal partitioning. In test *cpu_2* a job uses the `rt_change_prio()` LXRT call to raise its own priority to the highest possible value. Afterwards it cannot be interrupted by the scheduler any more and is able to block the CPU for a arbitrary long time

As a matter of fact, this test shows clearly that the huge set of RTAI function calls, available to LXRT jobs, raises potential problems for partitioning. Similar to this, test 11 shows that the even larger set of Linux system calls is not save either.

All in all the test runs show that just using a standard LXRT process for a job does not guarantee partitioning. The three main problems identified are:

(1) The RTAI naming system.

(2) Access to all LXRT calls.

(3) Access to Linux system calls.

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

Chapter 6.

Restricting the Job Environment

This chapter will tighten the environment for the jobs to handle the three main partitioning problems identified at the end of Section 5.4. Their API will be restricted to a small set of functions whose arguments are strictly checked.

6.1. Implementing a Restricted Job API

Figure 6.1 (a) illustrates the current situation where a job is implemented as a simple LXRT task. A job is part of its task wrapper which will periodically call the jobs entry point function. For an application programmer it's possible to include RTAI header files and header files from the standard C library (like `<stdio.h>`) without restrictions. As a matter of fact, a job is able to use RTAI calls[1] and Linux system calls which results in the partitioning problems demonstrated in the last section.

A practical solution to restrict the API from a jobs point of view is to use a header file defining the complete interface. A DECOS job is only allowed to use functions included in this interface definition and nothing else. Each call to one of this API functions has to go through a *security layer* that checks if a job is allowed to perform this function call. After this step the corresponding RTAI function is called. This might look like this:

```
1 int pos_function(uint32_t sign, unsigned long name, ...)
2 {
3     /* check if signature is valid and to which job it belongs */
4
5     /* check if job is owner of "name" */
6
7     /* call the corresponding RTAI function */
8 }
```

[1] The availability of RTAI calls depends on the included header files. But even if e.g. only `rtai_shm.h` is included some basic functions (like the naming service) is always available.

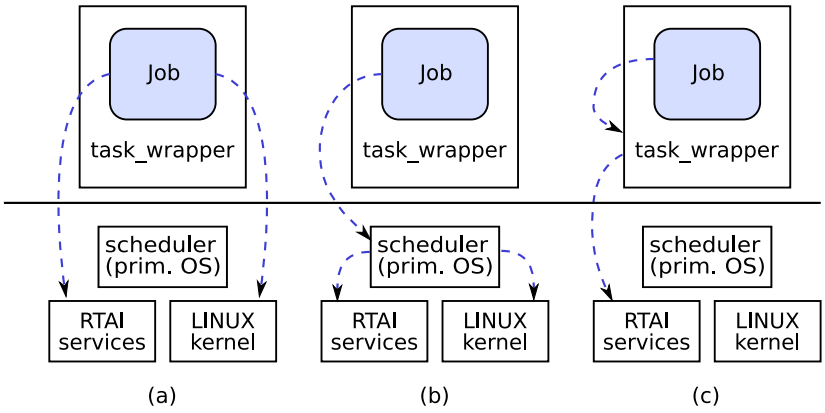


Figure 6.1.: Different possibilities to restrict the API for the jobs. (a) shows the current situation (b) proposes an implementation in kernel-space whereas (c) would be a user-space solution.

If the source code of a job is available, this can be easily checked. In the case that the job is only available as pre-compiled object file, checking is a bit more complicated but still possible. By using the standard Unix/Linux tools `nm(1)` or `objdump(1)` an object file can be checked for *undefined symbols*[2] . These undefined symbols are function calls to functions not available within the job. Per definition those external functions have to consist solely from the allowed interface functions.

For this *security layer* which is responsible for checking permissions, several implementations might be considered. Figure 6.1 (b) shows a possible solution in kernel space. In this case, the additional API layer is implemented as kernel module. The advantage of this approach is that there is one central place in the primary OS to handle incoming function calls for all partitions. The biggest problem with this solution is that it's usually not possible to directly call kernel space functions from user space.

A simpler solution is depicted in Figure 6.1 (c). The approach is to extend the already existing task wrapper to provide all necessary functions to the jobs. Again, the jobs are calling functions inside the task wrapper which checks the arguments and translates them to their corresponding RTAI functions. The drawback here is, that the code checking the function arguments is running in the same address space as the job that calls the function. Therefore the separation between job and security layer is not that strong. Due to this, the kernel space implementation was chosen to perform the parameter checks of the function calls. Nevertheless, a small part of the whole

[2] An other possibility would be to use the linker (`ld(1)`) to link against an object file, implementing the whole API. This fails as soon as there are undefined symbols.

additional software stack was implemented in user-space (see Figure 6.2).

6.1.1. The RRI Module

Following the conclusions from the last section, an additional security layer inside the kernel space was implemented. This kernel module will be called RRI (Restricted RTAI Interface) module from now on.

Additional to this kernel module some code in user-space is also used. This code (`pos_interface.c`) is linked to the jobs just like the task wrapper and provides the API to the RRI module (see also Figure 6.2.

LXRT — The RTAI Interface from User-Space

Section 5.2.1 already introduced the LXRT extension and its purpose: providing a hard real-time in user-space with an API to RTAI that is mostly the same as for a conventional RTAI kernel module. Each RTAI call is transfered into the kernel space and called again by a kernel module that was created during the initialization of the LXRT task.

The interesting part of this setup is the transfer of the functions calls from user-space to kernel-space, a thing which is usually done by syscalls. And in fact, LXRT calls work very similar. The arguments of a call are packed into a C struct and afterwards a *trap* (or a *software interrupt*) is signaled. This event interrupts the execution and is caught by the ADEOS layer which invokes the scheduling of the agent module which continuous with the function call. A more detailed description of this functionality can be found in the LXRT FAQ section of the RTAI documentation [RTA] and of course in the source code.

Restricting and Extending LXRT

The basic structure to keep track of available real-time functions in RTAI is the `rt_fun_entry` struct. All native LXRT functions are registered inside an array of such structs (`rt_fun_lxrt`).

```
1  /* from rtai-core/include/rtai_lxrt.h */
2
3  struct rt_fun_entry {
4      unsigned long long type;
5      void *fun;    /* function pointer */
6  };
7
8  /* from rtai-core/sched/rtai/sched_lxrt.c */
9
10 struct rt_fun_entry rt_fun_lxrt[MAX_LXRT_FUN]
```

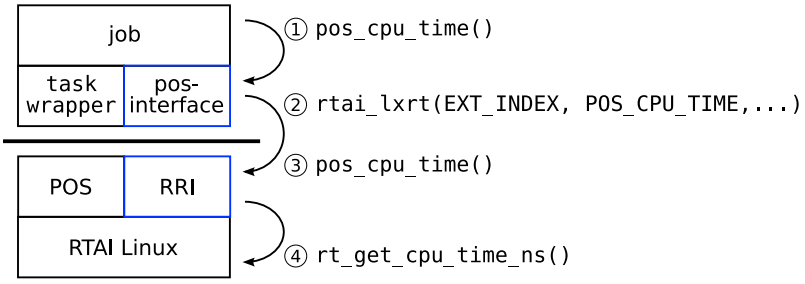


Figure 6.2.: Function calls resulting from the sample POS call `pos_cpu_time()`.

Each time a RTAI kernel module like, e.g. `rtai_shm` is loaded, it also adds its own provided functions to `rt_fun_lxrt`. Consequentially it also removes its entries again when it's unloaded. RTAI does not really provide an API function to remove entries from the `rt_fun_lxrt` array, but since the symbol is exported it can be rawly accessed from any code running in kernel space. Therefor, just looping through the whole array and overwriting all entries with an empty `rt_fun_entry` structure would removes all available native LXRT calls.

RTAI keeps track of external functions in an own array, `rt_fun_ext[]`. The first entry in this array points to the native LXRT functions. Since one goal of the RRI module is to prevent jobs from using those LXRT functions they have to be deactivated somehow. RTAI provides no convinient API for doing this, therefore the RRI module simply overwrites the LXRT functions array with an array of empty functions that. The primary OS can invoke this by calling the `disable_rtai_lxrt()` function exported by the RRI. A backup copy of the original functions is kept and can be put in place by calling `enable_rtai_lxrt()`. In theory this makes it possible to disable LXRT for single partitions and enabling it again for trusted partitions (like the middleware partition)[3].

Extending LXRT works by creating a new array of `rt_fun_entry` structs and register it in the `rt_fun_lxrt` array with a unique index.

```
1 static struct rt_fun_entry rt_pos_fun[] = {
2     [ POS_PRINT ]   = { NON_RT_SWITCHING, pos_print },
3     [ POS_CPU_TIME ]   = { NON_RT_SWITCHING, pos_cpu_time },
4     ...
5 };
6
7 set_rt_fun_ext_index(rt_pos_fun, EXT_INDEX);
```

[3]This has not been implemented but would be rather simple: Just by adding a flag to the static scheduling table to indicate if a partition is trusted or not.

pos_print()	Print function for debug purposes.
pos_cpu_time()	Get current CPU time in nanoseconds.
pos_heap_open()	Open a new group heap.
pos_malloc()	Allocate some heap memory in hard real-time.
pos_free()	Free some heap memory.
pos_heap_close	Close heap.
pos_shm_open()	Open a new shared memory.
pos_shm_close()	Close a shared memory.

Table 6.1.: POS API Overview

Figure 6.2 shows which functions are called if a job uses the new `pos_cpu_time()` call to get the current CPU time.

(1) The job calls the `pos_cpu_time()` function from the user-space implementation of the POS interface.

(2) Now the `rtai_lxrt()` function is used to generate a trap which will be catched by the ADEOS layer.

(3) After some internal magic, the agent module of the job calls `pos_cpu_time()` again, but this time the RRI implementation in kernel-space.

(4) The RRI module performs the parameter checking and finally calls the corresponding RTAI function `rt_get_cpu_time_ns()`.

6.1.2. Job API

Remark that in Section 4.1.1, a minimal feature set for the primary OS has been defined. Scheduling is done by the OS but the jobs are provided with a call to give up the CPU by suspending themselfes. IPC is handled solely over shared memories. The primary OS provides function calls to handle shared memory. A message interface on top of shared memory is provided by the task wrapper. For dynamic memory handling a real-time heap is provided. A summary of the job API can be found in Table 6.1 and a more detailed description in Appendix B. Note that most of this functions map directly to a specific RTAI function.

6.1.3. Checking Function Calls

So far this Section described how to restrict the jobs by providing them only a small subset of the RTAI functionality. But as shown in the test runs the main problem regarding partitioning is the RTAI naming service that does not do any permission checking at all.

The solve this problem, the RRI demands as first argument for each function call a signature like shown in the detailed API documentation. This signature is a 32 bit key, used to identify the calling job. In the implementation done for this thesis, a table with all job signatures is defined in a header file and included by the RRI module and the task-wrappers. This is convenient for developing but is not 100 percent secure since each job has the whole table in its own memory. It would be a better solution if the table of possible signatures is kept in memory only by the RRI. Since this is in kernel space it's not possible for the partitions to read this table. The partitions have to be informed about the value of their own signature via another way.

Besides the signature table, the RRI also holds a name table in memory[4]. In the prototype implementation this table holds up to 10 names per partition. If a name shows up in the row for a partition it is allowed to use the name.

Each time a partition uses one of the `pos_*` API calls, the first step inside the RRI is to check which job is the caller by looking at the signature. If an argument is an object name, the next step is a check if the identified job is allowed to use this name by looking in the name table. If it's not there, there is still the possibility that the call tries to register a new object; e.g. the `pos_shm_open` function. In this case it is checked if the new name is unique in the *whole* table for all partitions. The implication of this *"new object names during runtime have to be unique in the whole table"* rule is that all shared resources/names have to be defined statically during the system integration phase. This allows a detailed off-line analysis of possible partitioning concerns in advance.

An general example how a POS function is translated into the RTAI function is shown below:

```
1  int pos_function (uint32_t sign, unsigned long name)
2  {
3      int job_id = signature_to_id(sign);
4      if (job_id == -1)
5          return 0;      /* no job with this sign */
6      if (!name_belongs_to_id(name, job_id))
7          return 0;      /* job isn't the owner */
8
9      /* call the corresponding RTAI function */
10     return rt_function(name);
11 }
```

6.1.4. System Calls and Permissions

Up to now we restricted the API for jobs to a small set of functions (Section 6.1.2) which arguments are strictly checked (6.1.3). The only remaining possibility for jobs

[4]Note that this name table is hold only in kernel space memory and is not readable by the partitions.

to execute code outside their partition context is via system calls. This especially bad since since all jobs are running with `root` permissions.

On a first look it seems to be useful to use dedicated Linux user account for each partition. Without root permissions jobs are much more restricted and the Linux kernel handles the authorization. But this approach cannot solve all problems as can easily be seen when considering the following points:

- The granularity of permission checking by user permissions is not small enough. For example, each partition using some shared memory functionality has to have read and write permissions for the `/dev/rtai_shm` device file.

- The Linux 2.6.9 kernel, used for the soekris nodes, provides 285 different syscalls (listed in `include/asm-i386/unistd.h`). It would be a huge effort to look through all of this syscalls and check if they might cripple partitioning even if they are just called from a non root user process.

Just like LXRT calls, system calls are caught by the ADEOS layer which forwards them to the Linux kernel in the default setup. RTAI registers a handler function that performs this forwarding and an additional post-syscall handler function which is called after the syscall. It's very easy to overwrite this handlers with new ones and similar to the function that disables LXRT, the RRI module offers a `disable_rtai_syscalls()` function for this task. For reasons which are explained in the next paragraph it's also necessary to provide a complementary `enable_rtai_syscalls()` function which resets the configuration to the RTAI default again. The implementation of this feature requires a patch to the RTAI source code which is described in detail in Appendix C.

Simply deactivating system calls at some point in the start-up of the node would be the easiest solution but is not possible because of *implizit system calls* done by some LXRT functions. Although LXRT functions have been deactivated and replaced with the POS interface functions there is no way to do e.g. the `pos_shm_open` function without using system calls[5], most noteable the `mmap` system call.

In practice, there are four functions in the POS API that are using syscalls: `pos_heap_open()`, `pos_heap_close()`, `pos_shm_open()` and `pos_shm_close()`. Their implementation looks a bit different than for the other functions (as shown at the end of Section 6.1.3) and is done in user-space. Nevertheless, authentication and authorization is still in kernel space inside the RRI module. It provides two additional functions, `pos_shm_access()` and `pos_give_up_shm()` for this purpose. The first handles argument checking and enables system calls on success. The second one is called after all system calls have returned and disables further access to them again. The listing below shows the principle on the example of the `pos_heap_open()` call:

[5] At least not without a major rewrite of the RTAI shared memory system and the `mmap` functionality.

```
1  void *pos_heap_open(uint32_t sign, unsigned long name, int size)
2  {
3      struct { uint32_t sign;
4               unsigned long name;
5      } arg_shm_access = { sign, name };
6      struct { uint32_t sign; } arg_give_up_shm = { sign };
7      void *retval = NULL;
8
9      /* ask rri for permission, enables syscalls on success*/
10     if (rtai_lxrt(EXT_INDEX, sizeof(arg_shm_access),
11                   POS_SHM_ACCESS, &arg_shm_access).i[LOW] == 0) {
12         return NULL;
13     }
14
15     /* do the memory allocation and mapping in user-space,
16      * uses several syscalls (open, ioctl, mmap, close) */
17     retval = rt_heap_open(name, size, USE_VMALLOC);
18
19     /* release access again (rri module disables syscalls) */
20     rtai_lxrt(EXT_INDEX, sizeof(arg_give_up_shm),
21               POS_GIVE_UP_SHM, &arg_give_up_shm).v[LOW];
22
23     return retval;
24 }
```

6.1.5. Job Start-Up and Migration

This section describes the small changes in the scheduler and the task wrapper which lead to a new start-up behaviour. Furthermore it lists some points to consider when migrating applications that have been developed for the DECOS prototype cluster to the new restricted environment.

Start-Up

Disabling native LXRT functions and Linux system calls already during the initialization of the RRI kernel module would be possible but is not necessary. Instead the task wrapper has been changed. The start up of a job and its following periodic execution can be separated into three different sections.

The first section is the code inside the initialization phase of the task wrapper which execution stops at the newly inserted first `rt_task_suspend()` function call. This code is completely trusted and is therefore allowed to use the full set of LXRT functions and system calls.

scheduler

```
1  init_RTAI_task();
2  obtain_all_LXRT_tasks();
3
4  disable_rtai_lxrt();
5  disable_rtai_syscalls();
6  schedule_jobs_once();
7
8  wait(semaphore);
9
10 while(1) {
11   BCU_interconnection();
12
13   while (!end_of_task_list) {
14     rt_task_resume(task[i]);
15     rt_sleep(task_exec_time[i]);
16     rt_task_suspend(task[i]);
17     i++;
18   }
19   wait(semaphore);
20 }
```

task_wrapper

```
1  init_LXRT_task();
2
3  new_rt_task_suspend();
4
5  init_point();
6
7  rt_make_hard_real_time();
8  rt_task_suspend();
9
10 while(1) {
11     entry_point();
12
13     rt_task_suspend();
14 }
```

Figure 6.3.: Pseudo code of the scheduler and the task wrapper adapted to the RRI module. Differences to the pseudo code in Figure 5.7 have been highlighted.

After each of the partition jobs has stopped at this point, the scheduler disables LXRT functions and syscalls and schedules each job again exactly once. In this step the job init function is executed. Afterwards the job switches to hard real-time and suspends itself again just like in the original task wrapper. The control flow goes back to the scheduler once more which has now the chance to restrict the environment even further. A possible idea — which has not been implemented — would be to deactivate some POS API functions that shall be used only during the job init[6].

Migration Guide

Integrating the RRI module into an existing cluster should be rather easy since it's possible to do the migration in several steps. Just loading the RRI module is completely transparent to the existing jobs. After that it's possible to recompile some jobs and link them against the user-space part of the POS interface. As long as no kernel module calls the function to disable LXRT, jobs can use the POS and the LXRT interface in parallel.

As soon as a job uses only POS functions, the native LXRT functions can be disabled at least for this job. For this kind of mixed operation mode the schedule configuration table has to provide additional information whether a job still uses native LXRT functions or not. In this case the scheduler can enable/disable functionality depending on the next scheduled job. As soon as all jobs just use the POS interface a setup like described in the Section 6.1.5 above is possible.

6.2. Evaluation

The execution environment with the RRI modules mostly solves all three major problem areas that showed up in the test runs. The only downside is the slight performance penalty, caused by the additional authentication and parameter checking, of about 5 too 10 μs per POS function call in comparison with the native RTAI-LXRT functions.

Temporal Partitioning Table 5.8 shows that three tests violate the temporal partitioning in a significant way. Note that all of these tests achieve this by completely crashing the primary OS.

Test *shm_2* bypasses the RTAI naming system and tricks the primary OS into accessing a invalid memory address. By using a normal LXRT function, the job1 in test case *cpu_2* is able to stop the scheduling as well. And finally test case *syscall* shows that this is possible via a Linux system call as well.

Running this test with the RRI module prevents all of this problems.

[6]The four mentioned in Section 6.1.4 would be a good start.

Spatial Partitioning Spatial partitioning is violated by five test cases. Most of these problems are caused by the RTAI naming system that has no support for private objects at all. The RRI module demands a signature to prove ownership of an objects before accessing it and therefore solves this problem.

Jobs using Linux system calls are a threat to spatial partitioning as well. Catching them before they get executed and killing the offending job, like done by the RRI module, is an effective way to solve this.

6.2.1. Test cases revisited

The test cases introduced in Chapter 5 have been run again with the enabled RRI module, yielding the following results:

mem_1 A job derefering the NULL pointer crashes with the same effects as observed before (see also Figure 6.4). The RRI module does not influence the results.

mem_2 Like in the original test runs, the results here are the same as in *mem_1*. The RRI module does not influence the results.

stack_1 Overwriting the return address of the `entry_point` function and trying to execute code from an invalid virtual memory address still does not work. The RRI module does not influence the results.

stack_2 Similar to the other tests dealing with the virtual memory and the protection gained by it, the results are the same as before restricting the environment. The RRI module does not influence the results.

shm_1 The violation of spatial partitioning is successfully prevented since job1 cannot open the shared memory in the first place. The standard LXRT functions are not available and to access it via the POS interface functions, it would have been necessary to register the name of the shared memory in advance during system integration (see Section 6.1.3.

shm_2 Like explained above for the *shm_1* test, the RRI module prevents the violation of the spatial partitioning. LXRT functions are not available and the POS functions deny access to the shared memory since its name is not in job1's row of the name table. This effect is clearly shown in Figure 6.4. There is just a slight peak in the execution time of job1 at round 1000 when the RRI module checks the permission, but nothing else happens.

heap_1 The problems regarding spatial partitioning that showed up in this test case *cannot be prevented* by the RRI module directly. As soon as two jobs share a real-time heap, spatial partitioning cannot be guaranteed. Nevertheless, if the

jobs use different real-time heaps and the name table is set up correctly, the RRI module can ensure that they do not influence each other.

heap_2 Like in the first round of tests, the result here is equivalent to the test *shm_2*.

cpu_1 Just trying to block the CPU did not work with the original prototype setup and does not work with the new environment either. The RRI module does not influence the results.

cpu_2 Job1 is no longer able to raise its own priority, since calls to native LXRT functions are disabled and just return 0 without doing anything. Figure 6.4 illustrates how the execution of job1 is preempted at the end of its timeslot instead of being allow to exceed the allow time. The workload is distributed among the following five rounds.

syscall The RRI module prevents all system calls by partition jobs. The shutdown call, that is used as an example system call, is caught and job1, who initiated the call, is deleted. The resulting execution times are depicted in Figure 6.4. Note that there is a problem regarding temporal partitioning here: job0 and job2 are not called for two rounds!

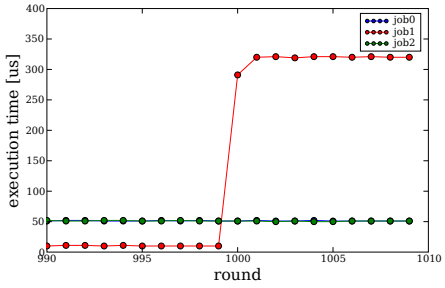
Nevertheless there are some remaining problems regarding the implementation which might be addressed in future work:

(1) The quality of the implementation can be improved. For example, there is a rather obvious race condition when a partition uses a POS function like `pos_shm_open()` that needs to enable system calls for a few rounds. During this time each other partition can execute an arbitrary syscall since they are enabled/disabled globally[7].

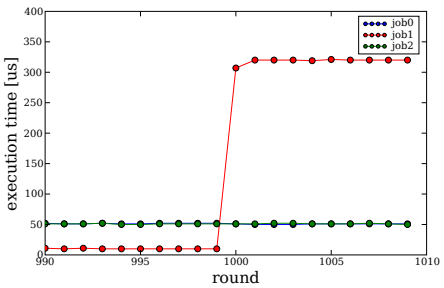
(2) The POS API itself can be improved to provide more features or a standardized interface like defined in the ARINC-653 specification [Air03].

(3) There is a temporal partitioning problem when shuting down a job that performs a system call. There has to be a better way than using `rt_task_delete()` to do this. See Appendix D for a discussion on the RTAI mailing list regarding this topic.

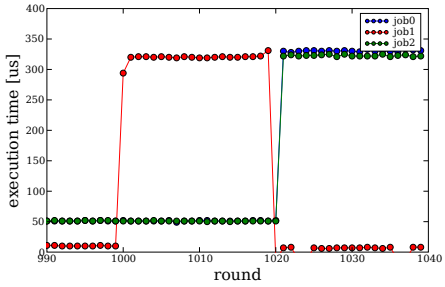
[7] Luckily, this effect does not appear in the init functions of the jobs, since they are executed sequentially. During normal operation, jobs should not use system calls anyway, if they are safety critical and hard-real time.



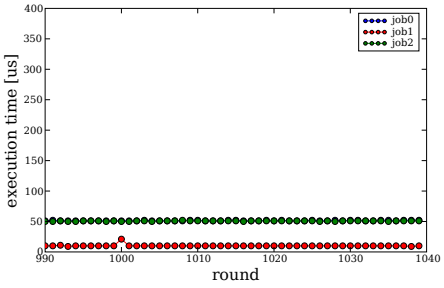
mem_1 default setup



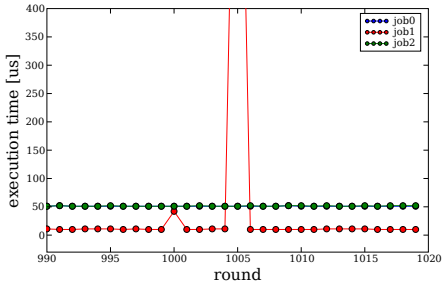
mem_1 with RRI module



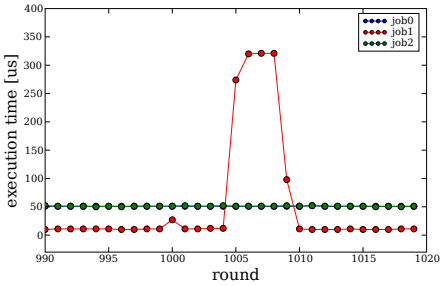
shm_2 default setup



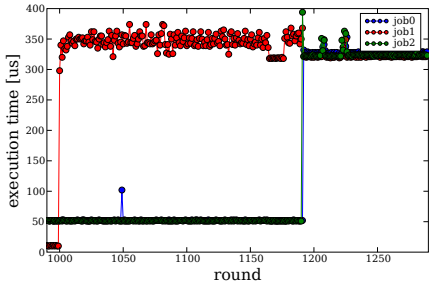
shm_2 with RRI module



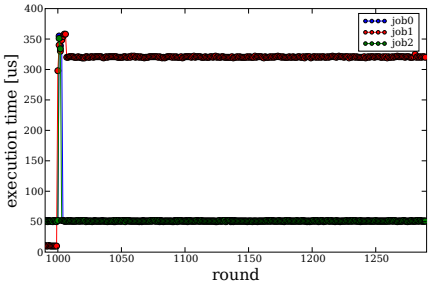
cpu_2 default setup



cpu_2 with RRI module



syscalls default setup



syscalls with RRI module

Figure 6.4.: Comparision of the execution times of various test cases with and without the RRI module.

God is a challenge because there is no proof of his existence and therefore the search must continue.

Donald Knuth

Chapter 7. Conclusion

With the advent of integrated system architectures, like the DECOS architecture, implementing a partitioning operating system on top of a real-time Linux kernel is especially useful for prototyping such architectures. The open-source nature of Linux and the huge number of accompanying projects offer a very flexible and convenient development environment.

This thesis showed a possible implementation of such a partitioning OS on top of RTAI-LXRT Linux. The main approach to achieve the goal of partitioning was to use the LXRT extension of RTAI Linux which allows real-time tasks in user-space. A conventional RTAI task, running in kernel-space, was used to schedule those user-space tasks according to a static schedule generated offline. Although this solution already provides a certain degree of partitioning, a closer investigation of the implementation uncovered some weak points. One of these points is, that RTAI Linux uses a naming system which allows real-time tasks to share objects (e.g., shared memories). This naming system provides no mean of authentication and prevents the allocation of truly private objects. Furthermore standard LXRT calls and Linux system calls, available to the jobs per default, can be used to bypass the partitioning. The validity of this theoretic investigation was shown by implementing and running a couple of test cases.

At first, the results from these test cases showed that by using separated user-space processes for the partitions, spatial partitioning for the private memory of these processes holds. No way was found for a partition to access any memory outside of its own virtual address room. This is however not true any more for the shared memory used for inter-partition communication. Per definition this memory has to be reachable for several partitions. Access to shared memory is done via the RTAI naming service. The non existence of any authentication steps, before being able to access the shared memory in an arbitrary way, was the first significant problem found in the (spatial) partitioning. More test cases showed that the unrestricted availability of Linux system calls and LXRT calls to the jobs inside the partition is a problem as well. Both of these interfaces can be used to violate partitioning.

As a solution to the identified problems this thesis introduces an additional kernel

module, the so called RRI (Restricted RTAI Interface) module. This RRI module provides functionality to completely disable access to the native LXRT interface for the jobs running inside the partitions. They are restricted to a completely new interface that is basically a wrapper for a few selected LXRT functions. The main difference is, that, contrary to the native functions, this API implements a basic form of authentication. A valid signature is necessary to access private or shared memory. Furthermore, the RRI module is also able to control the access to Linux system calls. In order to do this, the function registered by RTAI as pre-system call hook in the ADEOS hardware abstraction layer is overwritten with a function defined inside the RRI module which does not propagate the system call and shuts down the job that invoked the system call in the first place. In order to implement this functionality, a patch against the RTAI sources was necessary.

The resulting, restricted environment yields significantly better results in the test cases concerning partitioning. Nevertheless further work on the implementation can be done in several directions. A new job API, providing more features and following some kind of standard, like the ARINC-653 specification, would be a prominent example. Further on, a better integration of the checksum calculation into the monitoring framework is needed. The overhead can be reduced to an acceptable degree by executing only a small part of the overall workload per round and the results should be sent over the network as well. Perhaps the most important point to consider is the temporal partitioning problem that still exists when using Linux system call inside partitions.

Appendix A.

Acronyms

AADL ..... Architecture Analysis and Design Language
ADEOS ..... Adaptive Domain Environment for Operating Systems
API ..... Application Programming Interface
ARINC ..... Airlines Electronic Engineering Committee
ASCII ..... American Standard Code for Information Interchange
BCU ..... Basic Connector Unit
CAN ..... Control Area Network
CC ..... Communication Controller
CNI ..... Communication Network Interface
CPU ..... Central Processing Unit
DAS ..... Distributed Application Subsystem
DECOS ..... Dependable Embedded Components and Systems
DMA ..... Direct Memory Access
DoS ..... Denial of Service
DSoS ..... Distributed System of Systems
ECU ..... Electronic Control Unit
EDF ..... Earliest Deadline First
FCR ..... Fault Containment Region
GPS ..... Global Positioning System
I/O ..... Input/Output
IMA ..... Integrated Modular Avionics
IP ..... Internet Protocol
IPC ..... Inter-process communication
LIF ..... Linking Interface
LIN ..... Local Interconnect Network
LXRT ..... Linux Real-Time
MAC ..... Media Access Control
MEDL ..... Message Descriptor List
MMU ..... Memory Management Unit

MTBF ..... Mean Time between Failures
MTTF ..... Mean Time to Failure
MTTR ..... Mean Time to Repair
NFS ..... Network File System
OO ..... Object Oriented
OS ..... Operating System
PCI ..... Peripheral Component Interconnect
PCMCIA .... Personal Computer Memory Card International Association
POS ..... Primary Operating System
RAM ..... Random Access Memory
RRI ..... Restricted RTAI Interface
RTAI ..... Real-Time Application Interface
RTHAL ..... Real-Time Hardware Abstraction Layer
RTOS ..... Real-Time Operating System
SCU ..... Safety-Critical Connector Unit
SCU ..... Secondary Connector Unit
SOC ..... System-On-a-Chip
SQL ..... Simple Query Language
TDMA ..... Time Division Multiple Access
TTA ..... Time-Triggered Architecture
TTE ..... Time-Triggered Ethernet
UDP ..... User Datagram Protocol
UML ..... Universal Modeling Language
WCET ..... Worst Case Execution Time
XCU ..... Complex Connector Unit

Appendix B.

Primary Operating System — Job Interface

This chapter describes the API that application programmers can use for their jobs. Most of these functions are just mapped to a corresponding RTAI function. As a matter of fact it might be useful to look at the RTAI API documentation [RTA] for more detailed information.

The main difference to the RTAI functions are the additional *id_key* parameters which are used by the RRI module for permission checking. Finally, for some calls, the provided API is even simpler that the RTAI counterpart since some optional arguments have been removed, like the memory allocation method for the heap and shared memory open functions.

```
int pos_print(const char *text)
```

A function that can be used to print some debug info on stdout. Maps to the `rt_printk` RTAI function.

Parameters

text: The text to be printed.

Returns

The number of printed characters, 0 on failure

```
RTIME pos_cpu_time(void)
```

Get the current CPU time in *ns*. The absolute value isn't very useful but the difference between two timestamps might be. Maps to the `rt_get_cpu_time_ns` RTAI function.

Parameters

none

Returns
The current CPU time in *ns* inside a 64 bit variable.

```
void *pos_heap_open(unsigned long name, int size)
```

Opens a new private heap for the job. Maps to the `rt_heap_open` RTAI function using the `USE_VMALLOC` method to allocate the memory. Note that this function can't be done in hard real-time and should therefore only be used during the initialization stage.

Parameters
name: The ID of the new heap. *size*: Size (in bytes) of the new heap.

Returns
A valid address on success or `NULL` in case of an error.

```
void *pos_malloc(int size)
```

Allocate some real-time heap opened previously with the `pos_heap_open` call. Maps to the `rt_halloc` RTAI function.

Parameters
size: Size of the memory chunk in bytes.

Returns
A pointer to the newly allocated memory or `NULL` in case of an error.

```
void pos_free(void *adr)
```

Frees some previously allocated heap memory. Maps to the `rt_free` function call.

Parameters
adr: A pointer to the memory chunk to be freed.

Returns
—

```
int pos_heap_close(unsigned long name)
```

Closes the real-time heap. Maps to the `rt_heap_close` RTAI function.

Parameters
name: The ID of the heap that should be closed.

Returns

The size of the succesfully freed heap, 0 on failure.

```
void *pos_shm_open (unsigned long name, int size)
```

Allocate a chunk of memory to be shared with the primary OS or some other partitions. Maps to the `rt_shm_open` RTAI function using the `USE_VMALLOC` method to allocate the memory. Note that this function can't be done in hard real-time and should therefore only be used during the initialization stage.

Parameters

name: The ID of the new shared memory. *size*: Size (in bytes) of the new heap.

Returns

A valid address on success or `NULL` in case of an error.

```
int pos_shm_close(unsigned long name)
```

Closes a shared memory. Maps to the `rt_shm_free` RTAI function.

Analogously to what done by all the named allocation functions the freeing calls have just the effect of decrementing a usage count, unmapping any user space shared memory being freed, till the last is done, as that is the one the really frees any allocated memory. [RTA]

Parameters

name: The ID of the shared memory to be freed.

Returns

The size of the successfully freed memory, 0 on failure.

Appendix C.

Patch against the RTAI sources

This chapter describes the patch against the RTAI source code (against 3.1r2 but easily adaptable for newer versions) which was necessary for the implementation of the public `enable_rtai_syscalls()` function (see Section 6.1.4).

The patch is needed since the two handler functions that are responsible for syscalls in RTAI Linux per default (`lxrt_intercept_syscall()` and `lxrt_intercept_syscall_epilogue()`) are defined static and can't be accessed by the RRI module. To solve this issue this patch removes the `static` keywords, exports the symbols to the kernel symbol table and adds function prototypes in the `rtai_lxrt.h` header file.

```
1  --- rtai-core/sched/rtai/sched_lxrt.c.orig
2  +++ rtai-core/sched/rtai/sched_lxrt.c
3  @@ -2096,7 +2096,7 @@
4           adeos_propagate_event(evinfo);
5   }
6
7  -static void lxrt_intercept_syscall(adevinfo_t *evinfo)
8  +void lxrt_intercept_syscall(adevinfo_t *evinfo)
9   {
10          adeos_declare_cpuid;
11
12 @@ -2120,7 +2120,7 @@
13          }
14  }
15
16 -static void lxrt_intercept_syscall_epilogue(adevinfo_t *evinfo)
17 +void lxrt_intercept_syscall_epilogue(adevinfo_t *evinfo)
18  {
19          RT_TASK *task;
20          if (current->this_rt_task[0] &&
21          (task = (RT_TASK *)current->this_rt_task[0])->is_hard > 1) {
22 @@ -2601,5 +2601,7 @@
```

```
 EXPORT_SYMBOL(get_min_tasks_cpuid);
 EXPORT_SYMBOL(rt_schedule_soft);
 EXPORT_SYMBOL(rt_do_force_soft);
+EXPORT_SYMBOL(lxrt_intercept_syscall);
+EXPORT_SYMBOL(lxrt_intercept_syscall_epilogue);

 #endif /* CONFIG_KBUILD */
```

```
--- rtai-core/include/rtai_lxrt.h.orig
+++ rtai-core/include/rtai_lxrt.h
@@ -483,6 +483,10 @@

  int set_rt_fun_entries(struct rt_native_fun_entry *entry);

+void lxrt_intercept_syscall(adevinfo_t *evinfo);
+void lxrt_intercept_syscall_epilogue(adevinfo_t *evinfo);
+
 #ifdef __cplusplus
  extern "C" {
 #endif /* __cplusplus */
```

Appendix D.

Discussion on the RTAI Mailing List

The following mail conversation took place on the RTAI mailing list on the 20th of September 2006 and discusses ways to improved the implemented method of disabling Linux system calls for LXRT tasks.

```
From: "Bernhard Leiner" <mailinglists.bleiner@gmail.com>
To: rtai@rtai.org
Subject: disable system calls for LXRT Tasks [RTAI v3.1r2]

Hello,

In my project I'm more or less trying to implement a sandbox for
untrusted code (which has to run in hard real-time).

Before asking my actual questions, lets describe my setup a bit
further:

The "untrusted" code is an object file with unknown content. The only
thing I know is the interface to it which consists of an init()
function and a work() function.  I created a task-wrapper that
contains a main function, does all the necessary LXRT initialization
and finally calls init() once and work() periodically. This task
wrapper is linked together with the unknown object code. So far this
setup works pretty well.

Now I'm trying to restrict the possibilities of the unknown code. One
of the things I would like to do, is to make it impossible for the
resulting LXRT task to use Linux system calls in its work() function.
To do this, I created a RTAI kernel module, synchronized to the LXRT
task with semaphores, which should perform the task of disabling
system calls.
```

Since there is no "official" way to do this I'm currently messing around in the interface to the ADEOS layer. More precisely I'm trying to register new handler functions for ADEOS_SYSCALL_PROLOGUE and ADEOS_SYSCALL_EPILOGUE.
The best solution I came up with is the following handler function for the prologue:

```
static void new_intercept_syscall(adevinfo_t *evinfo)
{
    RT_TASK *task;

    adeos_declare_cpuid;
    /* propagate event if not from RTAI domain */
    if (evinfo->domid != RTAI_DOMAIN_ID) {
        adeos_propagate_event(evinfo);
        return;
    }
    /* The original plan was NOT to propagate the event in this
     * case but that didn't work (system freeze). Now the syscall
     * is just propagated but the task is immediately deleted
     * afterwards. It looks like this works somehow since the syscall
     * isn't executed. Perhaps deleting a task also deletes all
     * pending events...
     */
    adeos_propagate_event(evinfo);
    adeos_load_cpuid();
    task = rt_smp_current[cpuid];
    rt_task_delete(task);
}
```

Since I didn't suceed in ignoring syscalls I just delete the tasks that misbehaves. This is more or less ok too...
Can somebody with knowledge about RTAI internals please tell me if
    1) my comment in the code is correct?
    2) there is a much better way to achive this?

Thanks a lot for reading this far :)

kind regards,
    Bernhard Leiner

And here the answer from Paolo Mantegazza, the lead developer of RTAI Linux:

```
From: Paolo Mantegazza <mantegazza@aero.polimi.it>
To: Bernhard Leiner <mailinglists.bleiner@gmail.com>
CC:  rtai@rtai.org
Subject: Re: disable system calls for LXRT Tasks [RTAI v3.1r2]

I'd do not do that at the ADEOS level if I was using the RTAI
scheduler already. Syscall are intercepted already in it and
forbidding syscall is just a matter of checking the process that is
making it and do what appropriate.

If I had to do it I would add and export a pointer to a function
initialised at NULL. The RTAI call prologue should check it and
proceed as usual if it is not set. The you can set it to a function
that checks the caller and ask RTAI to return doing nothing. Another
way is to redirect all Linux system calls to a single dispatcher of
yours, just to change a table with some 200 names, then to do the same
stuff in Linux directly.

Paolo.
```

Bibliography

[Air03] Airlines Electronic Engineering Committee, Aeronautical Radio Inc., Annapolis, MD. *ARINC Specification 653-1*, October 2003.

[ALR01] A. Avizienis, J.C. Laprie, and B. Randell. Fundamental concepts of dependability. Technical Report Research Report 01-145, LAAS-CNRS, Toulouse, France, April 2001.

[AMD] AMD. *Elan SC520 Microcontroller*.

[AUT05] AUTOSAR; `http://www.autosar.org`. *AUTOSAR - Main Requirements, Version 2.0*, December 2005.

[BD00] E. Bianchi and L. Dozio. Some experiences in fast hard realtime control in user space with RTAI-LXRT. In *Realtime Linux Workshop Orlando*, 2000.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[BF04] R. Black and M. Fletcher. Next generation space avionics: a highly reliable layered system implementation. In *The 23rd Digital Avionics Systems Conference (DASC)*, volume 2, pages 13.E.4 – 131–15, October 2004.

[Bir00] Tim Bird. Comparing two approaches to real-time linux. *Linux Devices, http://linuxdevices.com*, December 2000.

[Bos91] Robert Bosch GmbH., Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.

[Bra97] S. Bradner. RFC 2119; key words for use in RFCs to indicate requirement levels. http://www.ietf.org/rfc/rfc2119.txt, March 1997.

[But06] Giorgio Buttazzo. Research trends in real-time computing for embedded systems. *SIGBED Rev.*, 3(3):1–10, 2006.

[CM+00] Pierre Cloutier, Paolo Mantegazza, et al. Dipam-rtai position paper. In *Real Time Operating Systems Workshop (RTSS)*, November 2000.

[DW05] Sven-Thorsten Dietrich and Daniel Walker. The evolution of real-time linux. In *Seventh Real-Time Linux Workshop, Lille, France*, November 2005.

[HHL03] S. Hairong, J. J. Han, and H. Levendel. Availability requirement for a fault-management server in high-availability communication systems. In *IEEE Transactions on Reliability*, volume 52, pages 238–244, June 2003.

[HPOS05] B. Huber, P. Peti, R. Obermaisser, and C. El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the decos architecture. In *Third International Workshop on Intelligent Solutions in Embedded Systems*, pages 3–16, May 2005.

[IIK+06] Hiroaki Inoue, Akihisa Ikeno, Masaki Kondo, Junji Sakai, and Masato Edahiro. Virtus: a new processor virtualization architecture for security-oriented next-generation mobile terminals. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 484–489, New York, NY, USA, 2006. ACM Press.

[KB03] H. Kopetz and G. Bauer. The time-triggered architecture. In *Proceedings of the IEEE*, volume 91, pages 112–126, January 2003.

[KM03] A. J. Ko and B. A. Myers. Development and evaluation of a model of programming errors. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 7 – 14, October 2003.

[Kop97] Hermann Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Systems*. Kluwer Academic Publishers, fourth edition, 1997.

[Kop03] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *The Sixth International Symposium on Autonomous Decentralized Systems, ISADS 2003.*, pages 139–146, April 2003.

[KOPS04] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri. From a federated to an integrated architecture for dependable real-time embedded systems. Technical report 22, Vienna University of Technology, Real-Time Systems Group, 2004.

[KS03] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *In Proceedings of Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.

[LD05] `http://linuxdevices.com` Linux Devices. Embedded linux market snapshot, 2005.

[LH02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 35(1):88–93, January 2002.

[LIN03] LIN Consortium. *LIN Specification, Version 2.0*, September 2003.

[LKMZ00] Y. H. Lee, D. Kim, M.Younis, and J. Zhou. Scheduling tool and algorithm for integrated modular avionics systems. In *Proceedings of the 19th Digital Avionics Systems Conferences (DASC)*, volume 1, pages 1C2/1 – 1C2/8, October 2000.

[McK05] Paul E. McKenney. Attempted summary of "rt patch acceptance" thread. Linux Kernel Mailing List, June 2005.

[MDP00] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, 2000(72es):10, 2000.

[Mil90] Milan Milenkovic. Microprocessor memory management units. *Micro, IEEE*, 10(2):70–85, April 1990.

[MN99] M. Mock and E. Nett. Real-time communication in autonomous robot systems. In *Proceedings of The Fourth International Symposium on Autonomous Decentralized Systems, 1999. Integration of Heterogeneous Systems.*, pages 34–41, March 1999.

[Mor91] M. J. Morgan. Integrated modular avionics for next generation commercial airplanes. *Aerospace and Electronic Systems Magazine, IEEE*, 6:9 – 12, August 1991.

[MZ94] N. Malcolm and W. Zhao. The timed-token protocol for real-time communications. *IEEE Computer*, 27(1):35–40, January 1994.

[NF04] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, January 2004.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack (`http://www.phrack.org`)*, 7, November 1996.

[Ope04] Open group; `http://www.opengroup.org`. *POSIX IEEE Std. 1003.1*, 2004.

[OPK05] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *10th IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 241–253, February 2005.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organization & Design — The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Fransisco, California, USA, second edition, 1998.

[PO06] P. Peti and R. Obermaisser. A diagnostic framework for integrated time-triggered architectures. In *9th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 539–549, April 2006.

[POT+05] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 2–13, May 2005.

[RCKH05] Alessandro Rubini, Jonathan Corbet, and Greg Kroah-Hartmann. *Linux Device Drivers*. O'Reilly, third edition, February 2005.

[RS94] K. Ramamritham and J.A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of the IEEE*, volume 82, pages 55–67, January 1994.

[RTA] RTAI Linux `www.rtai.org`. *RTAI-LXRT API Documentation*.

[RTC92] Radio Technical Commission for Aeronautics, Inc. (RTCA), Washingtonm, DC. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[Rus99] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms and assurance. NASA contractor report CR-1999-209347, NASA Langley Research Center, October 1999.

[S+98] John A. Stankovic et al. *Deadline Scheduling for Real-Time Systems, EDF and Realated Algorithms*. Kluwer Academic Publishers, 1998.

[Sim96] H.A. Simon. *The Sciences of the Artificial*. MIT Press, third edition, October 1996.

[SOE] Dual PC-Card/Cardbus 133 Mhz 486 based mainboard for wireless applications. `http://www.soekris.com/net4521.htm`.

[Sta88] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):93–108, October 1988.

[Sta01] William Stallings. *Operating Systems*. Prentice Hall, Upper Saddle River, New Jersey 07458, USA, fourth edition, 2001.

[TTT02] TTTech Computertechnik AG., Schoenbrunner Strasse 7, A-1040 Vienna, Austria. *Time-Triggered Protocol TTP/C – High Level Specification Document*, 2002.

[TvS02] A. S. Tannenbaum and M. van Steen. *Distributed Systems — Principles and Desing Paradigms*. Prentice Hall, Upper Saddle River, New Jersey 07458, USA, 2002.

[vD06] Leendert van Doorn. Hardware virtualization trends. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 45–45, New York, NY, USA, 2006. ACM Press.

[Yag] Karim Yaghmour. *Adaptive Domain Environment for Operating Systems*. The Adeos Project `www.adeos.org`.

[YPW97] Li Yanbing, M. Potkonjak, and W. Wolf. Real-time operating systems for embedded computing. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 388–392, October 1997.