Asterix: A prototype of a small-sized real-time kernel

Andreas Engberg, 760224
Anders Pettersson, 620427
e-mail : {aeg98001,apn98009}@student.mdh.se

Mälardalen Real-Time Research Centre, MRTC
Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN

Abstract

Embedded systems are a fast growing and exciting market. A real-time operating systems is often a necessity in such systems. Most of todays real-time operating systems, both educational and industrial, supports state-of-practice methods for scheduling, analyzing and developing. Many off-the-shelf real-time operating systems lack desirable state-of-the-art features and tools for analyzes and development. The Asterix Framework project aims to fill that gap. This document presents an implementation of a real-time kernel that support state-of-the-art methods. The kernel supports algorithms such as Fixed Priority Scheduling, Immediate Inheritance Protocol, monitoring of both applications and the kernel, interprocess communication by Wait- and Lock-Free channels. The collaboration with Obelix Configuration Tool lead to that memory allocation, initializing of memory areas, etc. can be done off-line and the implementation can be kept simple and hence decrease the size of the kernel. The source code is released as Open Source to make it available for researchers, students and developers in the industry to test new ideas.

Available from Mälardalen Real-Time Research Center, Department of Computer Engineering, Mälardalen University, P.O. Box 883, SE-721 23 Västerås, SWEDEN

Contents

1 Introduction **4**
- 1.1 Background . . . 4
- 1.2 Requirements of the Asterix real-time kernel . . . 5
- 1.3 Summary . . . 5
- 1.4 Document Outline . . . 6

2 Name definition in the Asterix Framework **7**
- 2.1 General definitions . . . 7
- 2.2 Definitions specific to the Asterix real-time kernel . . . 7

3 Basics **9**
- 3.1 A general RTOS . . . 9
- 3.2 Operating systems . . . 9
  - 3.2.1 Mutual exclusion . . . 9
  - 3.2.2 Synchronization . . . 10
  - 3.2.3 Interprocess Communication . . . 10
- 3.3 Asterix RTOS . . . 11

4 The Asterix Kernel **12**
- 4.1 System timer . . . 12
- 4.2 Task-model . . . 12
- 4.3 Design . . . 13
  - 4.3.1 Tasks . . . 13
  - 4.3.2 Task Control Block . . . 14
  - 4.3.3 Task Control Block List . . . 14
  - 4.3.4 Tasklist . . . 15
  - 4.3.5 Readyqueue . . . 15
  - 4.3.6 Error Handlers . . . 15
  - 4.3.7 Execution-time measurement mode . . . 16
  - 4.3.8 Jitter . . . 16
  - 4.3.9 Kernel and User Mode . . . 16
- 4.4 System-calls . . . 16
  - 4.4.1 t_return . . . 16
  - 4.4.2 yield . . . 16
  - 4.4.3 self . . . 17

5 Operating system **18**
- 5.1 Semaphores . . . 18
- 5.2 Signals . . . 18
- 5.3 Wait- and Lock-free Communication . . . 19
- 5.4 Interrupts . . . 20

6 Hardware **21**
- 6.1 Description . . . 21
- 6.2 Problems . . . 21
  - 6.2.1 Replacing existing programs . . . 21
  - 6.2.2 Debugging . . . 21
  - 6.2.3 Architecture . . . 21

7 Analysis **23**
7.1 Kernel . . . 23
7.1.1 Jitter and Execution-time . . . 23
7.1.2 Memory . . . 23
7.2 Operating system . . . 24
7.2.1 Semaphore . . . 24
7.2.2 Signals . . . 25
7.2.3 Wait-free communication . . . 25

8 Future work **26**
8.1 Improvements . . . 26
8.1.1 Interrupts . . . 26
8.1.2 Wait- and Lock-free channels . . . 26
8.1.3 Signal extension . . . 26
8.1.4 Execution-time measuring . . . 26
8.2 Next generation . . . 26
8.2.1 Minimize memory . . . 26
8.2.2 Monitoring . . . 27
8.2.3 Soft tasks . . . 27
8.2.4 Execution-time jitter on tasks . . . 27

9 Conclusions **28**

Chapter 1

Introduction

In this chapter we first give a brief introduction to what real-time systems are. We then give an introduction and background to a our Master thesis and continues with a brief description of some of the algorithms used.

1.1 Background

Embedded systems are a fast growing and exciting market. An example is a computer system that controls the speed of an electrical motor. A system like this is often called a *Real-Time System* (RTS). In [11] a RTS is defined by:

> *A **real-time system** is a system that reacts on external events and perform actions within a fixed time. Correctness is not only depending on correct result but also on the point in time when the result was delivered.*
>
> *A **hard real-time system** is a system where the cost of not fulfilling the functional and temporal requirements is very high.*
>
> *A **soft real-time system** tolerates errors with respect to the functional and temporal requirements. This means that constraints may be broken (typically with an upper bound defined over a time interval), and that a service may be accomplished a bit late occasionally (again within an upper bound).*

In order to handle such systems an operating system that supports mentioned definitions is needed, in other words a *Real-Time Operating System* (RTOS).

When selecting a software solution, for a new embedded system, developers face a number of technology choices:

- *Microprocessor* must be chosen on basis of cost, performance, power and application requirements.
- *Real-Time Operating System*, commercial or in-house development to fulfill the requirements of the embedded system.
- *Software Development Tools* which may be bundled with the commercial RTOS or selected for a particular microprocessor.

There exists a large number of commercial RTOS on the market. Most of them are based on old assumptions of RTS and do not supply the necessary infrastructure needed to apply modern/contemporary scheduling and analysis theory. This causes problems when new ideas need to be tested and evaluated, since the real-time aspect differs between state-of-the-practice and state-of-the-art. There already exists state-of-the-art RTOS, for example MiThos [9], the Spring kernel [15] and Emeralds [19], all with different advantages and drawbacks. These systems are often not general solutions but aimed at something special, for example special hardware or scheduling theory. Most of the state-of-the-practice (commercial, off-the-shelf) RTOS are in some way configurable in order to fit in a general embedded system solution. The features in these RTOS vary a lot, and also the way they are distributed to the purchaser. Some of them are delivered as modules and others as open source code that can be modified and compiled to a specific system. If the developer needs to modify the RTOS, the source-code is necessary, which may be a problem if the source-code is not open. Different kinds of synchronization, communication and collaboration between tasks are supported and so on. There exists more or less practical development environments that could be used to facilitate the configuration of a specific embedded system [16] [18]. In fact many of these commercial RTOS (if not all) lack desirable state-of-the-art features of a RTOS. To many questions answers are hard to find, for example:

- Is the system predictable?
- Does the system support debugging of any kind?
- Which scheduling principles are supported?
- How does the error handler work, if there is one?
- Is priority inversion prevented?
- Is the system multitasking, is it preemptive?
- Which development tools exists?

This gave birth to the idea of a new RTOS at the Department of Computer Engineering at Mälardalens University. The name of the project is the Asterix Framework. The idea is to develop a new analyzable

distributed RTOS, a communication system, a powerful development environment and analysis tools. In other words a complete set of development tools to configure and analyze a real-time system. The Asterix Framework have the following features:

- A task-model which supports state-of-the-art scheduling theory.
- Support for debugging and monitoring.
- Wait- and Lock-free interprocess communication.
- Compiling kernel, in other words only the parts of the system that are utilized are included.

The task-model includes both preemptive scheduling of statically generated schedules [1] and *Fixed Priority Scheduling* (FPS) [4]. It also supports both strictly periodic and event triggered tasks. The tasks terminate each time they finish execution.

A multi-tasking RTOS, like the Asterix Framework, enables the user to divide an application into separate, individual programs called *tasks*. A task is the basic unit of execution in any application that runs under Asterix. Each task can be started, suspended, resumed and terminated separately. Asterix handles two kinds of tasks, hard and soft. A hard task has hard time-constraints which must be kept or the whole system is taken into a failure mode. The hard tasks in the system are included in the system analysis, which makes sure that all time-constraints are possible to fulfill. A soft task do not have hard time-constraints and a missed deadline, for example, will not endanger the system. In other words, soft tasks will be assigned execution resources when there is time left from the hard task execution.

In order to interest embedded real-time system developers, the Asterix Framework is portable and the source-code open. The architecture of the Asterix Framework is illustrated in figure 1.1.

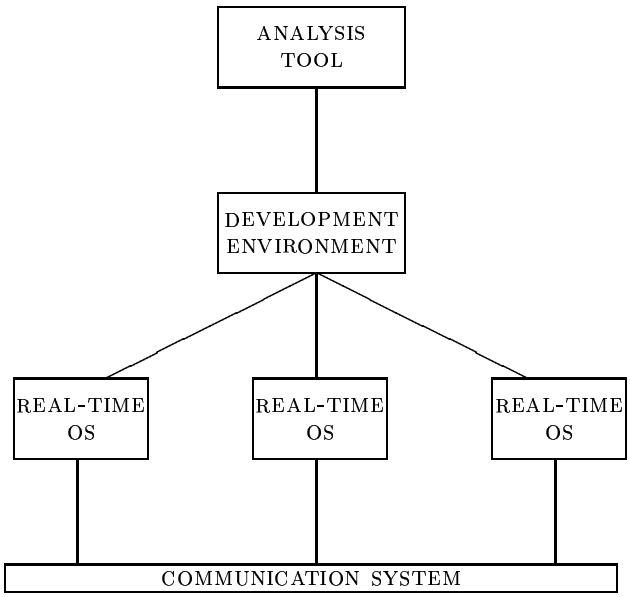

Figure 1.1: The Asterix architecture

1.2 Requirements of the Asterix real-time kernel

In the Asterix Framework all proposals have minimum requirements, below is a numeration of some of the criteria for the Asterix real-time kernel.

- The task model must support Static Scheduling, Fixed Priority Scheduling with or without offsets.
- A task must consist of at least the attributes periodicity, offset, priority and deadline.
- Support for future implementation of monitoring.
- The kernel must support wait- and lock-free communication between tasks, and synchronization mechanisms such as signals and semaphores.
- The kernel overhead must be predictable and computable.
- All jitter that may be caused by the kernel must be minimized.
- The system must be compilable, portable and scalable.

Other features that the kernel supports, but which were not required in the thesis work: Support for soft tasks, mail-box communication between tasks, error and/or exception handling, execution-time measurement mechanisms for tasks and critical sections, monitor task-switches, timestamps and minimize a task's execution time jitter.

1.3 Summary

In the Asterix Framework project two proposals were available, Asterix the real-time kernel and Obelix the configuration tool. This document describes the work of designing, analyzing and implementing the real-time kernel. The work was done during a 20 weeks period, at the Real-Time System Design Laboratory at Mälardalens University. The real-time operating systems that exists today often lacks support of development tools and state-of-the-art methods. If such real-time operating system are to be found, often the source code are restricted and not availably. A solution[1] with Open Source code is presented. The algorithms in the kernel are as simple as possible but still practically useful. By keeping the implementation simple leads to that the kernel is very restrictive in using resources. The Obelix Configuration Tool is

[1]Based on the ideas in *Monitoring, Testing and Debugging of Real-Time Systems*, doctoral thesis by Henrik Thane

responsible for allocation of resources. Memory allocation routines, initializing routines, etc. can thereby be omitted in the kernel and memory optimization can be done off-line. The task model supported by Asterix includes Fixed Priority Scheduling with or without offsets. This makes the kernel flexible as many schemes based on other scheduling algorithms can be converted to a Fixed Priority scheme, for example static scheduling. It also includes synchronization of tasks such as signals and semaphores. Communication between task are performed by Wait- and Lock-Free channels.

1.4 Document Outline

The outline of this document is such, that in chapter 2 the terminology used in this document are defined. First terms that can be applied to the Asterix Framework are defined, and then terms specific to the Asterix real-time kernel. Chapter 3 describes the basics of a Real-Time Operating System and sets the aim of Asterix the real-time kernel. How the kernel is designed and implemented is presented in chapter 4. Methods that can be used to decrease the usage of RAM is also described. In chapter 5 the interface to the kernel are explained, it is also described how to use interface and the design and implementation. The first version is implemented on the Lego Mindstorm RCX. How the hardware architecture looks like is briefly explained in chapter 6. The RCX is shipped with a ROM which advantages and drawbacks encountered are also briefly described. The analyzes and the outcome of the implementation are presented in chapter 7. Some ideas of future work are given in chapter 8. Finally, summary and conclusion of the work are discussed in chapter 9. This document can be read by our supervisor, students in a computer science program, those who are interested in embedded system and in particular small sized real-time kernels. It can also be read by those who continues to develop the Asterix Framework. Preferably the reader of this document have basic knowledge of real-time systems. This document is best read with the Asterix Manual and Obelix Development Environment and manuals related to Obelix.

Chapter 2

Name definition in the Asterix Framework

In this chapter we give some definitions, first in general and then specific for this document.

2.1 General definitions

Asterix Framework

A distributed real-time system, including analysis tools, development environment, real-time kernel and communication system.

Asterix

Asterix is a small-sized real-time kernel that supports state-of-the-art methods.

Obelix

Obelix is an easy-to-use Development Environment for the Asterix Framework.

Miraculix

Miraculix is the analysis tool for Asterix and Drakar.

Drakar

Drakar is the communication system, connecting a distributed Asterix system.

System / Asterix System

Asterix system is both the real-time executive and the user defined application, compiled together. An Asterix system is downloaded to a target system.

Mode

An Asterix system (see above definition) can be run in two different modes, test-mode and normal-mode.

Application

An application is a program written by developer(s). The application may consist of arbitrary number of tasks and resources. An application can be programmed to have different application modes (schedules).

Task

A task is the basic unit of execution in any application that runs under Asterix. The Asterix Framework supports two classes of tasks, hard and soft.

Wait- and lock-free communication

Wait- and lock-free communication is a form of state based interprocess communication. Writers and readers communicate over a channel that is made up by a number of buffers. Wait-free communication guarantees instant access of the channel and a task cannot be blocked.

Target system

The target system is the platform/hardware on which Asterix system is executing. In the first version the target system is a Lego Mindstorm RCX with a Hitachi H8 processor. Asterix is portable so different target systems can be used.

Configuration file

The configuration file is the input file to the Obelix Configuration Tool. This file describes the user application on a higher level than ordinary source code. It is written in ASCII format, therefore it is human-readable. The configuration file can be produced by an application programmer or an application design tool.

2.2 Definitions specific to the Asterix real-time kernel

Task

In the kernel four types of tasks are supported, soft periodic, hard periodic, soft aperiodic and hard aperiodic. A periodic task will execute once within a predefined time-interval with a start time that could be the start of the time-interval or an offset in relation to the time-interval. It also has a deadline in relation to the start of the time-interval. An aperiodic task do not have any periodicity, instead it will execute when a specific event has occurred. The hard tasks are guaranteed from an off-line scheduler or response

time analysis to meet their timing constraints. Soft tasks may or may not meet their timing constraints.

Task Control Block

Information about a task is gathered in a structure called *Task Control Block* (TCB). This structure consists of several fields that hold information about the task it belongs to. When a task-switch occurs, the TCB is used to store and restore information. To represent all the tasks in the system, each task's TCB is stored in a list called the Task Control Block list.

Kernel Overhead

Kernel overhead is the amount of CPU-time and and the amount of memory that the kernel uses. CPU-time and memory are used in system-calls, interrupts, or when the kernel is invoked.

Jitter

By jitter, the difference in execution time between calls to the same function is meant. There exists several other types of jitter, but these are not considered in this work. Such jitter could be the difference in release times of a task, the difference of a task's execution time, difference in clock-ticks arrival into the system.

Predictability

Predictability is the behavior of the system. Such behavior can be timing and memory usage. The knowledge of the behavior is essential when off-line scheduling is used.

Size-of function, σ

The sizes of data-structures will be altered depending on the hardware. Therefore, a size-of function σ is needed to specify the size of a particular data-type or structure on a specific hardware-platform. This function is used when calculating the memory overhead.

Chapter 3

Basics

In this chapter a general RTOS is described, followed by an explanation and problems with different methods implemented in the operating system, such as:

- *Mutual exclusion*
- *Synchronization*
- *Interprocess Communication*

Finally, a walk-through of how the Asterix kernel mix state-of-the-art techniques with traditional functionality is given.

3.1 A general RTOS

In a multi-tasking RTOS, such as Asterix, tasks can preempt each other and create an illusion of several tasks 'executing' in parallel. This means that the kernel must know whether a task is able to execute or not. By using a 'state' flag for each task, the kernel can, in an easy way, determine which tasks that are valid for execution.

Another issue is that some tasks cannot be interrupted by other tasks and therefore some sort of hierarchy is needed. A priority for each task is introduced and thereby let task execute in order of importance. This will prevent lower prioritized tasks to preempt tasks with a higher priority. The kernel checks, at certain time-intervals, which task that currently has the highest priority and is ready to execute. This task preempts the current executing task and is given the opportunity to run until it terminates or gets preempted by another task with higher priority.

3.2 Operating systems

The actual task-switching is done by the kernel, but an extension of the kernel is required to protect shared resources or let tasks communicate or synchronize. This functionality is put in the operating system and in this section, general approaches to these techniques are explained.

3.2.1 Mutual exclusion

Mutual exclusion is a technique to ensure that only one task at a time can access a shared resource. One way to implement mutual exclusion is to use *semaphores*. The major problem with semaphores occurs if two or more tasks wants to access a resource. These situations can cause *Priority Inversion* (PI), deadlock or starvation in the system. To illustrate the problem, a scenario of n tasks is assumed. Two tasks, task A with the lowest priority and task Z with highest priority, share a semaphore S. If task A takes the semaphore S before Z, Z will be delayed until A releases S, as illustrated in figure 3.1. The problem gets worse if all tasks with higher priority than A, preempts A and due to that A has the lowest priority, task Z will be delayed even more, or worse, never be able to execute again. A solution to this is to let the task that owns the semaphore inherit the priority of the highest prioritized task that have access to that semaphore, as shown in figure 3.1.

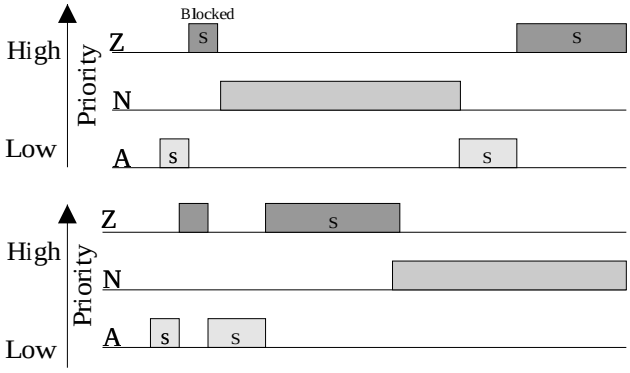

Figure 3.1: Priority inversion and how it is solved by using Immediate Inheritance Protocol.

An example of a classical priority inversion problem from real life can be read about in Risks Digests [6]. It is a story about the software in the Pathfinder spacecraft, landed on the Martian surface July 4th 1997.

Deadlock and starvation can easily be described as a four-way junction for cars. All cars enter the junction at the same time. In the deadlock case, see figure 3.2, all cars ends up in the middle of the junction but none will be able to move any further. Starvation occurs if all cars went back (put in reverse) and waited for n seconds and then enter the junction again (still at the same time), see figure 3.2. The cars are avoiding the deadlock situation but will not be able to pass the junction.

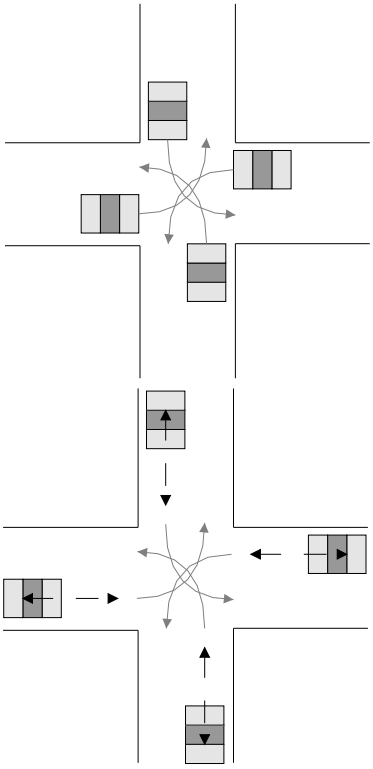

Figure 3.2: Deadlock represented as a four-way junction of cars(above).In the starvation case, deadlock is avoided but since all cars are still entering the junction at the same time, no one will pass.

There exists many ways to design semaphores. The easiest is to just put a restriction in the way of using the semaphore, but this is not very suitable for a RTOS since problems may still occur and it is up to the designer of the task-set to make sure that no mutual exclusion cause problems. Some protocols, like the classical Priority Inheritance Protocol [14], varies a lot in complexity which make them difficult to implement and to use in a RTOS regarding timing aspect. Better method are thus needed.

3.2.2 Synchronization

Tasks needs to be synchronized with each other. For instance if a task A and Task B is waiting for a *Analog-to-Digital Converter* (ADC) to produce a value. It takes 10 - 20 milliseconds for the ADC to produce a value. If no synchronization is used, as shown in figure 3.3, task A and task B can read the ADC at any specific time, even when the conversion is not complete.

A better approach is to let the other tasks know when the ADC has a valid value. Task C is controlling the ADC and informs task A and task B that a new value has been produced. The most common solution to this problem is to use signals, see figure 3.4. By letting task A and B wait for signal s, the ADC can convert a new value with any disturbance form A or B. When this is done, task C can raise s and thereby allow task A and task B to read a newly produced and

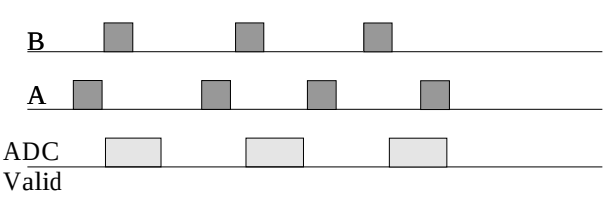

Figure 3.3: No synchronization allows task to read illegal values from the ADC.

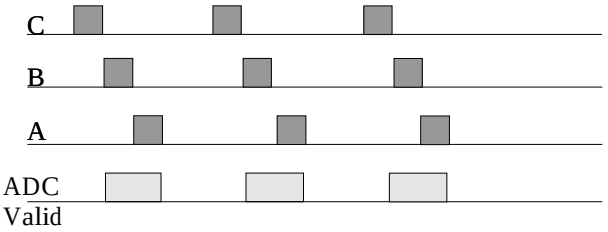

Figure 3.4: Synchronization with signals. Task C is informing the other tasks when a valid value can be read.

correct value.

Another way to solve this is by using a semaphore that protects the value, see figure 3.5. The only time a value can be read by task A and task B, is when they have taken a semaphore. The ADC only produces a value when task C owns the semaphore and after the conversion, task C releases the semaphore and tries to take it again when the ADC wants to produce a new value. Hopefully, task A and task B will take the semaphore within that time or else task C will try to access the semaphore again.

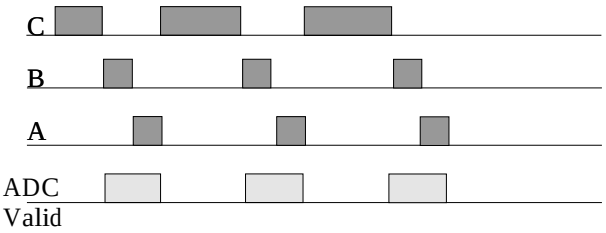

Figure 3.5: Synchronization with a semaphore. Task C owns the semaphore until a valid value is produced by the ADC.

The semaphore approach is much more difficult to use for this purpose, not only because it is more difficult to coordinate the tasks in such a way that no problem with mutual exclusion occurs, section 3.2.1, but also that the tasks A and B cannot read the ADC value independent of each other.

3.2.3 Interprocess Communication

In a RTOS, tasks frequently communicate with other tasks, either via shared resources, mailboxes or some other communication primitives. Which primitive to choose is dependent on four major issues:

Availablilty Data is valid as soon as it has been produced.

Independency No task should be waiting on another to access a valid data.

Static The data can be read anytime during the execution of task and still give the same result.

Atomicity A task that reads a value but get preempted, must read the original value, even if a newer one exists.

In the case of the shared resource, the availability issue is easily achieved but the independency, static and atomicity issues are almost impossible since a task that is accessing the data will not allow anyone else to access the resource, see section 3.2.1. Mailboxes represents a traditional way to implement IPC, but this technique has its disadvantages as well. First of all, if task A want to send message M to task B, but task B's mailbox is full, the message will be undeliverable. Even if task B's mailbox was not full, task B must still process all messages in the mailbox until it can read M. This can be solved by using priorities on the messages [8] but the solutions is rather complex.

3.3 Asterix RTOS

The Asterix Framework is to be used as a platform for elaborations in computer engineering courses and as a test platform for researchers. The framework must be portable to different types of CPU's and it must be easy to use, so the user of the framework can focus on his own work and not to learn a complex system.

To design a kernel that is easy to use and easy to implement the idea were to support algorithms that were as simple as possible but still were usable. This leads to several advantages: The first prototype can be designed and implemented in a short time, complexity of the analysis can be kept simple, calculation of various overhead can be done by simple formulas.

By choosing *Fixed Priority Scheduling* (FPS) as scheduling algorithm, both FPS and Static Schedules can be used since a static scheme can be expressed as a FPS scheme when offsets are used. This is briefly explained in section 4.2.

In Asterix, an algorithm called *Immediate Inheritance Protocol* (IIP) [4] is used in order to avoid problems with mutual exclusion. IIP is basically a simplification of Priority Ceiling Protocol [14]. A description of the protocol can be found in section 5.1.

To maintain the demands of predictability the focus is on minimization of jitter and kernel overhead. Minimal jitter can be achieved by careful programming. But to minimize kernel overhead it is not sufficient to program in a 'smart' way. The idea to minimize the amount of CPU-time and memory used by the kernel is achieved by a compiling kernel. This means that all tasks, semaphores, etc. are compiled together with the kernel and therefore creates a system that is large enough for the specific target system. Another benefit is that all structures used within the kernel can be initialized off-line, which minimize the cost when starting the system. One major disadvantage with this approach is that the designer of an application must initiate all data-structures before compiling. This can be a problem if no tools are available. In the Asterix Framework, a tool called the Obelix Configuration Tool [3] can be applied to automatically generate and initiate these data-structures and thereby give the designer the possibility to describe the application on a higher level of abstraction.

Chapter 4

The Asterix Kernel

In this chapter, an explanation of the timer in the kernel is given, followed by a description of the task model supported by the Asterix kernel. The design and implementation of the kernel is presented last in this chapter.

4.1 System timer

The heart of Asterix is the system timer. The timer is set up to periodically activate the kernel. When the kernel is activated by the timer, it checks if the current task is the one with the highest priority, if not the kernel switches tasks. These time intervals are called *system ticks*. All time references that tasks make are given in the number of system ticks. The resolution of the timer can be altered by the designer of the system. If the resolution is high, yielding a short time between the timer-ticks, a greater part of the CPU-time will be required by the kernel. On the other hand, if the resolution is to low, the useful usage of the CPU can be low if tasks will finish their execution within a tick, see figure 4.1.

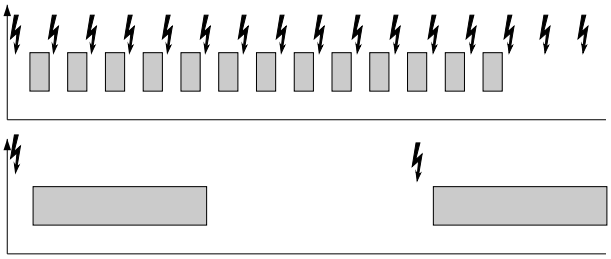

Figure 4.1: Two different system timers with high resolution (above) and low resolution (below).

4.2 Task-model

A task in Asterix can either run periodically or be triggered by an event. An task that is event triggered is called *aperiodic task*, and a periodically task is called *periodic task*. Periodic tasks can have a displacement, *offset*, set in relation to the period start. Even if tasks can start in different ways, they will still have a latest time, *deadline*, when they must terminate their execution. When a task has completed its execution, it will terminate and wait until a new period or a new event.

Priority is an unique value that represent the significance of the task. The priority determines in which order the tasks are dispatched. The kernel will always execute the task that holds the highest priority and is ready to execute.

A task is defined as: $task < P, T, O, DL >$
where P is the priority of the task, T is the period, O is the offset and DL is the deadline. If the task is aperiodic, it will not have a period or offset.

The tasks model in Asterix supports *Fixed Priority Scheduling* (FPS) with or without release time offsets and *Static Scheduling*. FPS is an algorithm that creates a set of tasks, where each task has a statically assigned priority. The priority is determined by off-line analysis. There is no schedule created, the scheduling is performed during run-time by a scheduler that let the task with the highest priority execute.

Task	period	Deadline	Priority
A	20	20	Highest
B	30	30	High
C	40	40	Normal
D	50	50	Low

Table 4.1: A set of tasks, where priorities are assigned off-line. Feasibility according to Fixed Priority Scheduling algorithm. An online scheduler let the task with the highest priority execute, if it is available for execution

Static Scheduling is an algorithm that creates a schedule with a given set of task by calculating the *least common multiple* (LCM) of the tasks periodicity. This is done pre-runtime and the schedule created is a table with the tasks period set to LCM and the release time of each task is determined by the offset. If a task appears more than once in the LCM period, each instance is treated as an unique task. The kernel dispatches the tasks from the created table. As stated earlier, the Asterix task model supports both FPS and Static Scheduling. This can be done by adding offsets to FPS. Static Scheduled tables can then be expressed as a FPS scheme by setting offset to each instance of a task equal to the task's release time. The priorities corresponds to each task's release time. All tasks in the set get a period set to the LCM of the tasks' original period.

Task	Release Time	period
A	0	LCM
B	10	LCM
A'	20	LCM
C	30	LCM
A''	40	LCM
D	50	LCM

Table 4.2: A set of task Static scheduling

The task-model consists of four states which task can occur in. Valid states and transition between the states are shown in figure 4.2. In table 4.3, an overview of the transitions in the task-model is given, with a brief description of what causes the transitions.

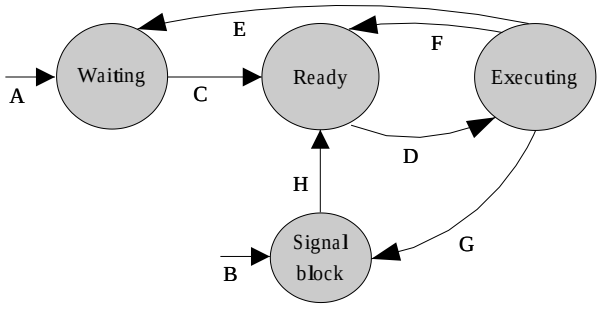

Figure 4.2: Transition in the task-model

Transition	Cause
A	Periodic task starts here
B	Aperiodic task starts here
C	Each time when a task is ready to execute in a new period.
D	When the scheduler determines that the task is the next one to execute.
E	When the task has terminated and are waiting for a new period.
F	When the task has been preempted.
G	When an executing task are blocked in order to wait for a signal.
H	When an aperiodic task is trigger by a signal.

Table 4.3: This table describes which state a task can be in under the systems life cycle

4.3 Design

There are several issues that should be addressed when designing an embedded real-time kernel. Such issues could be execution time, memory usage, jitter minimization and guarantees of tasks' timing constraints. Efficient execution time and memory usage are needed because our embedded real-time system are meant to be implemented on micro-controllers with limited computational power and RAM.

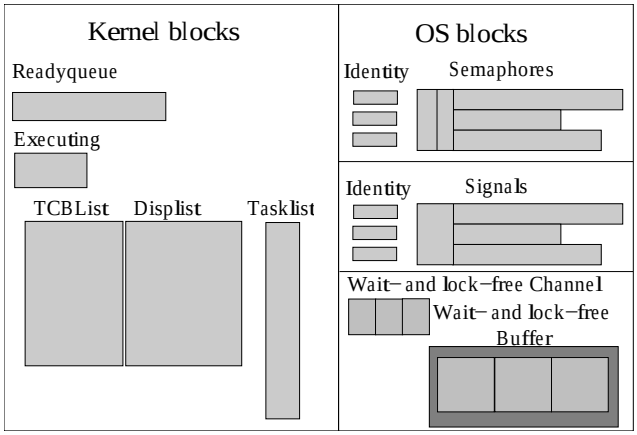

Figure 4.3: Asterix kernel showed as block diagram, both the kernel and the the extended kernel (OS)

The kernel is separated into five different blocks, see figure 4.3. This has the advantage that each of the blocks can be optimized for its own specific purpose. An explanation of the design decisions that are specific to each block are given in the section of each block.

The examples in this chapter are based on certain assumptions, if the assumptions not are valid in a example it is explicitly said. Assumptions are made that there exist an Asterix system with 50 task, the type of the tasks are of no significance. The tasks are not using any signals or semaphores.

4.3.1 Tasks

The kernel supports the following task-types: soft periodic, hard periodic, soft aperiodic and hard aperiodic.

The difference between the task types can be described as:

- soft periodic tasks are restarted every occurrence of their period and may or may not meet their timing constraints
- hard periodic tasks are restarted every occurrence of their period and must meet their timing constraints
- soft aperiodic tasks are waiting for a specific signal and may or may not meet their timing constraints
- hard aperiodic tasks are waiting for a specific signal and must meet their timing constraints

Soft periodic and soft aperiodic tasks are not guaranteed to finished before their deadline. An aperiodic task has no regular time-interval; it waits on an event to occur. On events, the kernel initiates the deadline of a task in relation to the start time of the task. Aperiodic task have yet another purpose, that is that it can be used to start the execution of interrupt service routine, see section 5.4. Tasks are only allowed to

start in either WAITING or SIGNAL BLOCK. Aperiodic tasks, waiting for a signal, starts in SIGNAL BLOCK, and all the other tasks starts in WAITING, see figure 4.2.

From the kernels perspective it is not sufficient to define a task as a user does. The internal representation of a task must contain more than just *priority* (P), *periodicity* (T), *offset* (O) and *deadline* (DL). Information about start address, parameters, error/exception handlers, stack-pointers, CPU registers, etc. is also needed. This is described in more detail in section 4.3.2.

The priority of a task must be an unique value. This is handled by the Obelix Configuration Tool by assigning appropriate priority to each task based on the user defined priority. When all tasks have unique priorities, the kernel can use these priorities as an unique identifier for each task. This simplifies the access in lookup tables and arrays, in other words no traversing of lists is required to find the task. Another benefit is that it is enough to compare the identifiers for tasks to determine which has the highest priority.

Since tasks can be preempted, each task has its own stack space. One of the big drawbacks with this solution is that it consumes a lot of RAM. An advantage is that the handling of a preempted task can be kept simple. When a task is preempted the state of the task can be saved on the stack. State means the processors registers, stack-pointer and program-counter of a task. The approach to save the state on user stack was not chosen in the Asterix kernel, why is explained in section 4.3.2.

4.3.2 Task Control Block

To design a system that allows preemption information about the state of a task must be stored. This could be saved on each tasks stack, but it will require that the stacks would be large enough to store the state in addition to the already required stack-size, given by the user. It will also be more difficult to monitor the system if a stack is used. Since the Asterix kernel is designed to support monitoring by using *deterministic replay* (DR) [17], the state needs to be stored where it is easily accessible. For each task a memory area is assigned for storage of the state. This memory area is generally called *Task Control Block* (TCB). The first approach was to create a data-structure that consisted of the necessary fields, and use this structure as the TCB. But due that Asterix supports DR, a lot of memory would be wasted because a tasks period, offset, etc. will not be altered during run-time and hence not necessary for the tasks' state. By using two structures instead of one, not only the problem with DR will be solved, it will also be very suitable for embedded systems due that the static structure can easily be put in ROM instead of RAM and automatically get a protection against overwritten memory. In the Asterix kernel, the static structure is called TCB and the dynamic structure Dispatch.

A TCB will contain a tasks *period* (T), *offset* (O), *deadline* (DL), *start-address* (SA), *input parameter* (PA), an *error-handler* (EH) and an initial *stack-pointer* (SP). The TCB is defined as:

$$TCB < T, O, DL, P, SA, PA, EH, SP > \tag{4.1}$$

A task is a function with one input parameter, a general pointer (void *). Tasks can thereby share the same function but pass different parameters for different behavior. The initial stack-pointer will not be altered during runtime. The pointer is copied into the stack-pointer in the tasks corresponding Dispatch-structure when the task is released. The error-handler will be explained in section 4.3.6.

The state of a task will be saved in the Dispatch-structure. The necessary parts that are needed to store are the *task state* (S) according to the transition-graph 4.2, *length until deadline* (LDL), *length until next period* (LPD), the current *stack-pointer* (SP), current *program-counter* (PC), *control register* (CCR) and all the *processor registers* (R0 ... Rn). The dynamic Dispatch-structure is defined as:

$$Dispatch < S, LDL, SP, PC, CCR, R0 \ldots\ Rn, LPD > \tag{4.2}$$

By introducing counters for deadlines and period, the task does not have to know the exact number of elapsed ticks. This is useful since the system timer will eventually overflow and be restarted. For instance, if the placeholder for the system ticks is just a 16-bit integer, it can only hold 65536 values. When the number of ticks have reached 65535, the next tick would be 0. To ensure that the tasks will not be affected from this overflow, they keep an intern counter instead of the system timer.

4.3.3 Task Control Block List

The *Task Control Block list* (TCBList) is a list that holds a TCB for each task. But due to that the TCB is divided into two parts, TCB and Dispatch, this list is also divided. The TCBList holds just the TCB and DispList holds all Dispatches for every task. A tasks position in the TCBList has the same position in the DispList.

By using arrays to represent the TCBList and DispList instead of linked lists, memory space equal to one pointer is saved for each task. For instance, if the system has 50 tasks, the size of (σ) a pointer is 2, and by using arrays instead of linked lists memory saved is: $50 * \sigma(pointer)$ bytes per list.
If pointer size is 4 bytes then 400 bytes (50 * 4 * 2) is saved. This technique can easily map a tasks identity and priority to an index in an array. Even though it takes more assembler instructions to resolve an indexed position in the array, it will save two positions in the TCB (ID and P). By using 'smart' coding, the penalty for resolving the index can be minimized. An example of this is shown in figure 4.4.

Since the parameters for the mapping are not known by the kernel, the actual mapping is done off-line by the Obelix Configuration Tool.

```
Original:

   for(taskidx=0;taskidx<no_tasks;taskidx++)
   {
      displist[tasklist[taskidx]].LDL++;
      displist[tasklist[taskidx]].LDP++;
      displist[tasklist[taskidx]].SP=NULL;
      displist[tasklist[taskidx]].CCR=0x0000;
   }

Smart:
   dispatch_t  *dispptr;

   for(taskidx=0;taskidx<no_tasks;taskidx++)
   {
      dispptr=&displist[tasklist[taskidx]];
      dispptr->LDL++;
      dispptr->LDP++;
      dispptr->SP=NULL;
      dispptr->CCR=0x0000;

   }
```

Figure 4.4: How smart coding can be used to minimize the penalty for using arrays.

To keep track of which task currently is running a variable, Executing, is used. The identity of the task is stored in Executing as long as the task owns the CPU. This is helpful for system-calls, to determine which task that has called the function.

4.3.4 Tasklist

The *Tasklist* is a list of references to the TCBList, and the idea is to use the Tasklist as a priority lookup-table. In order to separate soft tasks from hard, the hard task with the lowest priority must be higher than the soft task with the highest priority. A problem occur when a task is accessing a semaphore due the immediate inheritance protocol requires that the task change its priority. The solution is to create an extended list of tasks that holds references to TCBs according to position. The Tasklist holds all tasks and a placeholder, a virtual task, for the ceiling of each semaphore. The virtual task is a empty entry in which the identity of the task that currently is owning the semaphore is stored.

4.3.5 Readyqueue

The *readyqueue* is a queue of all tasks ready to execute. Traditionally the readyqueue is designed as a linked list. This classical approach has two disadvantages:

1. Each task in the queue will require at least a pointer the next task. The amount of memory needed will therefore grow with the size a pointer for each task in the system.

2. The number of tasks in the queue also vary from no task in the queue to all the tasks in the queue. The time it takes to traverse the list will depend from time to time, which in the end generates jitter.

Since the goal of the Asterix kernel was to minimize jitter and minimize the amount of memory needed, a traditional design is not suited unless some changes are made. If the array-approach is chosen, the size for 50 task will be: $50 * \sigma(taskid)$ bytes.
Assuming the identity fits in an 8 bits integer, 50 bytes would be needed for the readyqueue. This is a waste of memory, and to try to solve it with some kind of dynamic memory management is a contradiction to overhead requirements since this adds more code and hence more overhead and jitter from the kernel. The solution was to represent readyqueue as a string of bits with, at least, the size of the number of tasks in the task-list.

All tasks, including the virtual ones, must be considered when determine the amount of memory needed by the readyqueue. The sum of all 'real' tasks (n_t) and virtual tasks (n_s) divided over the available data-type (Δ)in the hardware platform gives the number of instances of Δ needed to represent the readyqueue. But theses instances must be an exact number of Δ, hence an upper bound will create the number of instances needed. A formula, R_q, that gives the number of instances needed is presented in 4.3.

$$R_q = \left\lceil \frac{n_t + n_s}{\Delta} \right\rceil \tag{4.3}$$

As in the example with 50 tasks, the amount of memory used will be 7 bytes with bits instead of bytes. This saves 43 bytes (50-7). Memory saved is far more if the next pointer also is taken into account.

By using this technique, both the problems with the traditional approach have been solved. The jitter will be minimal since the size of the readyqueue is constant, assuming that number of tasks is constant. Hence it will be possible to traverse all the positions every time and select which task that has the highest priority of those in the queue.

To determine which task that has the highest priority, each tasks *identity* (ID) is mapped to a bit position in the readyqueue. For example, the task with ID=5 will be represented by the fifth bit in the readyqueue. This means if the tasks' bit is set, the task is in the queue and ready to execute.

4.3.6 Error Handlers

Error handling is a tricky issue. Perhaps the most difficult thing is to identify different problems that can occur in the system. The kernel has deadline monitoring of tasks and protection against illegal accesses of resources (see chapter 5). All tasks will have an error handler, either by user-defined or the default handler. By letting tasks have their own error-routines, the kernel does not need to support all possible errors. When an error occur, the task causing the error will start its error routine. The routine will be executed in the kernel mode (section 4.3.9) with all interrupts disabled so

```
if( expression )
   x = 42;
else
   dummy = 42;
```

Figure 4.5: Usage of dummy code to reduce jitter in execution-time.

the error handling will not be preempted. Alternatives to this approach are discussed in the chapter 8.

4.3.7 Execution-time measurement mode

To achieve accuracy in scheduling it is important to know the execution time of both tasks and the kernel. One way to do this is to measure the execution time. Although measuring the execution time does not automatically find the *worst case execution time* (WCET), it can still be handy for the task designer. The tasks that is measured can be initialized in such a way that it will execute its WCET, but this action must be taken by the designer, not the kernel. When measuring a single task all system-calls are disabled (returning immediately). Measuring of the execution time of the kernel should be performed so that each part is measured, i.e the execution time of a task-switch should be measured in all possible cases when switching to another task and when no switch occurs. The mode also measures the different type of system-calls, so called critical sections. These sections requires that the system is complete and that the resources involved in the section remains unchanged after the measurement, otherwise the section must be measured again.

4.3.8 Jitter

Execution time jitter is an unwanted issue in a real-time system [17]. In order to minimized the jitter, several design-choices have been included in Asterix. Jitter minimization regarding kernel overhead is achieved by letting all paths in the kernel code have, as close as it can be, the same execution time. This is done by implementing dummy paths and dummy code that is equivalent to the real code. For example if-statements that have no else-branch are extended with a dummy else branch, as illustrated in figure 4.5. Another important approach is to always traverse lists until the end, even if the element needed was found in the beginning.

Further improvements of jitter minimization are given in chapter 8.

4.3.9 Kernel and User Mode

When the kernel is invoked or when system-calls are called it is appropriate to not to use space on the tasks' stacks. If the tasks' stacks are used, the amount of memory needed on each of them would increase by the size of the stack needed by the system-calls. It is better to have a separated stack for that purpose. An extra mode, kernel mode, is necessary to separate the system-calls from the ordinary code in the tasks. Kernel mode allows the system to work without use of memory from the user stacks and instead use the kernel stack. When the system-call is finished, the system changes from the kernel stack to the tasks' stack again. The execution continues in user mode until the next time the kernel is activated.

4.4 System-calls

```
void someCall( inparam_t param , ... )
{
   declaration of variables;
   statements;
   disable interrupt;
   {
      declaration of variables;
      Statements;
   }
   enable interrupts;
}
```

Figure 4.6: How interrupts are disabled within a system-call.

A system-call is a connection between a user and the kernel. When making a system-call, the kernel will require that the interrupts are disabled (if necessary). To ensure that interrupts are disabled correctly, the construction in figure 4.6 is used. This gives the possibility to enclose variable assignments, that can cause side-effects if interrupts are enabled, in a block. When a system-call is initiated, various context switches in the kernel are invoked and the system goes into kernel mode 4.3.9. A system-call is implemented as a faked interrupt so the kernel can be entered differently but only one exit for all context-switches, regardless if they are triggered by software or hardware.

4.4.1 t_return

.

When a task is about to terminate, the last thing to do is inform this to the kernel. The is done by the system-call *t_return*. The kernel makes all necessary updates and the next task in the ready-queue will start or continue to execute.

4.4.2 yield

Yield is a system-call that activates the scheduler even if no system tick have occurred. This is mainly intended for the user to be able to create a system where tasks voluntarily give other tasks the possibility to run. It has, however, only the ability to switch to a task with higher priority than the current one, just like a ordinary system tick. In a system based on FPS, a yield-call will have no effect due to that task that

```
void Task_A( void *parameter )
{
   while(1)
   {
      do_something_useful;
      ...

      yield();
   }
}
```

Figure 4.7: Traditional usage of yield.

called yield where the executing one and thereby is the highest prioritized task at the moment. A version of yield is implemented to be used in the case when an interrupt invokes yield. The only thing that differ yield from a user and a yield from an interrupt is that in the case of the user initiated yield, all other interrupts must be explicitly disabled. While a yield invoked from an interrupt automatically disables the interrupts via the hardware.

The most traditional way to use yield is shown in figure 4.7, but such a task will never terminate thus never restart its period. The task will not be able to keep the deadline since this is determined at the restart of a tasks period. Asterix still support such a design, but only as soft tasks since such a task-construction will never be able to keep their deadlines.

4.4.3 self

Sometimes it can be useful to known the identity of a task. Especially if several tasks share the same function. Self() returns the tasks identity. It is intended to be used by the user to find the identity of the task, and pass it as an argument to other system-calls. The call performs a lookup in the Tasklist and returns the value stored there.

Chapter 5

Operating system

A description of the design and implementation of the extensions of the kernel, such as interrupts, signals, semaphores and wait- and lock-free communication, are given in this chapter. These parts are separated from the kernel because they are not a necessity for the system to work. Compiled systems may or may not include these parts, depending on the users need.

5.1 Semaphores

Since semaphores can cause problems in the system 3.2.1, an algorithm called *immediate inheritance protocol* (IIP) is used to prevent dangerous situations. The IIP requires that all semaphore contains a priority ceiling, which is one step higher than the priority of the highest prioritized task of those who can access the semaphore. When a task takes a semaphore, it increases its priority to this ceiling and thereby prevents the other tasks that wants the semaphore to execute due to that they will have lower priority than the ceiling.

A semaphore in Asterix needs plenty of information that may not be the ordinary semaphore case. The definition of a semaphore is:

$$Semaphore < O, N, P >$$

The information in a semaphore is a list of owners of the semaphore (O), the number of owners (N) and the virtual task in the Tasklist (P).

All semaphores are stored in a list, an array of semaphore structures, where each semaphore id is mapped to an array index. The actual mapping is done by the *Obelix Configuration Tool* (OCT) [3] by creating a variable with the name of the semaphore and initiate it with the index to the semaphore array. In this way both the kernel and the user can easily determine which semaphore the system-call is regarding.

The implementation of semaphores is divided into two OS system-calls. One for requesting a semaphore and one for releasing a semaphore:

```
void getSemaphore( semid_t semid );
void releaseSemaphore( semid_t semid );
```

In the getSemaphore call, three controls are made to ensure that the task is legally accessing the semaphore. The first control is that the task claiming the semaphore must have passed a semaphore identity check. If an identity of a semaphore is given that is not valid, the tasks error handler is called. If the task that currently owns the semaphore and the task that is running is the same, then the call is returning. This is done to make sure that a task do not request for a semaphore twice without releasing the semaphore in between. The last check performed is to see whether the task is allowed to access the semaphore or not.

Note: When a task is owning a semaphore the task must continue to execute or be preempted by a task with a higher priority. The task that owns a semaphore is not allowed to wait for a signal or by any other method block itself, unless it is granted by an off-line analysis. It is up to the programmer of the task to ensure that this rule is not violated.

If all controls are passed then the task is moved to the semaphores virtual task in the task, which corresponds to the ceiling of the semaphore. The virtual task is moved to the readyqueue and a context switch, via yield section 4.4.2, is called. It is up to the kernel to determine if the task owning the semaphore will continue to execute or if a task with higher priority is allowed to execute. This can lead to that the task owning the semaphore can be delayed by a task with higher priority. This is no problem regarding priority inversion, deadlock and starvation. Only when responsetime is important, this can be seen as a problem.

5.2 Signals

Signals have three purposes: to serve as synchronization between periodic tasks, wake up aperiodic tasks and to trigger the Interrupt Service Routine (ISR) when an interrupt has occurred. The definition of a signal is:

$$Signal < O, N, B, L >$$

A signal is represented as a list of owners that can access the signal (O), the number of tasks in the ownerlist (N), references to tasks that is currently waiting for the signal (B) and finally the number of tasks in the waiting-list (L).

Signals are supported by two system-calls :

```
void waitSignal( sigid_t sigid );
void raiseSignal( sigid_t sigid );
```

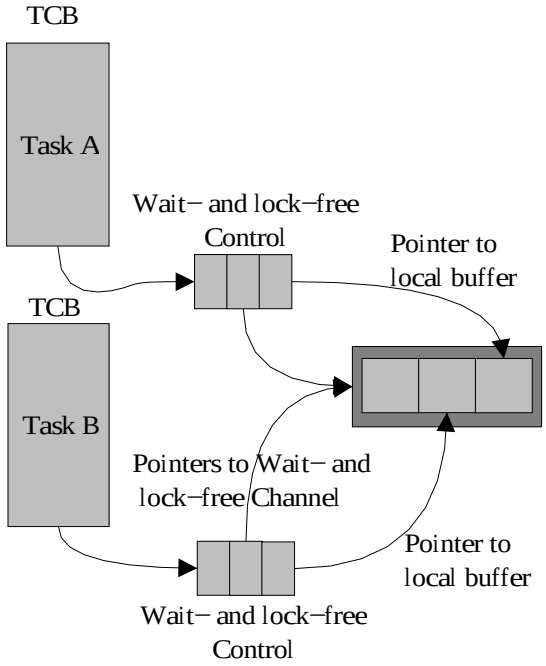

Figure 5.1: The connection between a task and its corresponding wait- and lock-free channel.

Similar to the semaphores, the signals also have protection against unauthorized accesses. Every time a task is either waiting or raising a signal, a check is done to see whether the task is allowed to use that signal or not. If an illegal access is discovered, the tasks error-handler will be invoked. Problems may still occur, since a periodic task can wait on a signal that is raised by an aperiodic task. If this aperiodic task never starts, the periodic task will be blocked until it misses its deadline which is detected by the deadline-monitor and the task error-routine starts. No control is made when a signal is sent by an interrupt since an interrupt is not scheduled as a task and hence not have an identity.

5.3 Wait- and Lock-free Communication

In the current version of Asterix kernel a simplified wait- and lock-free algorithm is implemented. The reason for this is that the responsibilities for design and implementation has altered during the work. The solution given here is not optimal in any way. There are lot of improvements that could be made, the most obvious improvement is shown in the OS analysis, section 7.2.3, regarding memory usage and CPU usage.

Wait- and lock-free communication is method to accomplish communication between tasks. The advantage with this method is that it is a non-blocking communication. Non-blocking means that if two or more

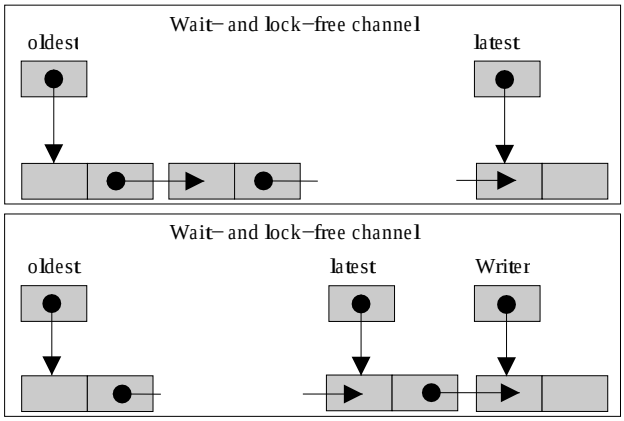

Figure 5.2: How a linked list is used to represent the buffer within a wait- and lock-free channel.

tasks wants to read from a *wait- and lock-free channel* (WLFC), no one of the readers are delayed because of another task. This is done by assigning one buffer to each reader of the WLFC, explained in Obelix Development Environment [3]. One extra buffer is added to assure that there always exists one free buffer in the WLFC.

The formula for calculating number of buffers in a WLFC is

$$n_{buffers} = n_{writers} + n_{readers} + 1 \quad (5.1)$$

Currently, only one writer is supported per WLFC, but there can be an arbitrary number of readers. So the formula to use is:

$$n_{buffers} = n_{readers} + 2 \quad (5.2)$$

System-calls to be used from the tasks:

```
buffertype_t *readWaitfree(bufferid_t id);
buffertype_t *writeWaitfree(bufferid_t id);
```

Both the read- and writeWaitfree functions returns a pointer to the buffer to operate on. Since the buffers are user-defined it is the user who is responsible for filling the buffer with data.

A WLFC contains an array of buffers (B), an array of counting semaphores (S), a pointer to the oldest value in the buffer (OLD), a pointer the the newest value in the buffer (NEW) and a list of all nodes that can use the buffers (NODES). The definition of a wait- and lock-free channel is:

$$WLFC < B, S, OLD, NEW, NODES >$$

To know which buffer that contains the most recently written value, a linked list is used in which each node has a pointer to a buffer in the wait- and lock-free channel. The reason to use a linked list is that it is easier and faster to sort than an array. This solution is not optimal from the memory and jitter constrains and needs to be analyzed and redesign to better suit the constrains. The list is sorted so that a pointer

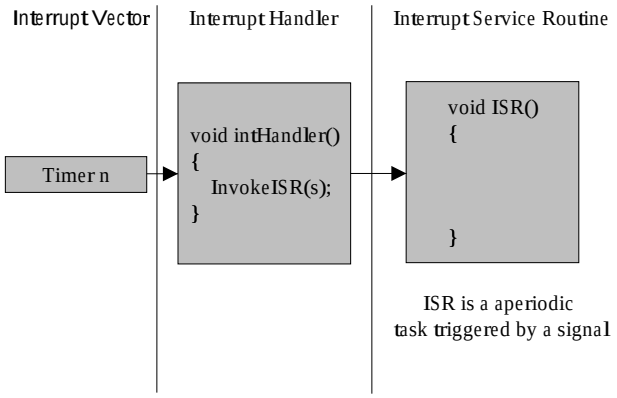

Figure 5.3: The order of execution when an interrupt has occurred.

to the newest value is in the end of the list, and the oldest in the front of the list, as shown in figure 5.3. This helps the kernel to assign the newest value to the readers and a free buffer to the writers.

Functions to be called from the kernel:

```
void updateWaitfree(tcbid_t tcbidx);
void t_returnWaitfree(tcbid_t tcbidx);
```

Every time a task is moved to the readyqueue due of a new period, it is assigned to a pointer to a buffer within the WLFC. The connection between a task and its WLFC is made by a *wait- and lock-free control-block* (WLFCB), shown in figure 5.1.

The buffer, assigned to the WLFCB, will be the most recently written if task is a reader and the first free buffer with the oldest value if the task is a writer. This assigning phase, updateWaitfree, is performed by the kernel by looking in the sorted linked list that represents a WLFC. When a task terminates, it is updating the WLFC by either just releasing it if the task is a reader, or updating and rearranging the internal linked list if the task is a writer. This update is done by the function t_returnWaitfree and it is automatically called from the t_return 4.4.1.

5.4 Interrupts

The interrupt handling has been solved by creating two code parts, one *interrupt handler* and one *Interrupt Service Routine* (ISR), as shown in figure 5.4. The interrupt handlers are executed in kernel-mode with all interrupts disabled, in contrast to the ISR that is executed in user mode. When an interrupt is triggered, an interrupt handler is invoked. This is done by letting the interrupt vector for the corresponding interrupt be set to the address of the interrupt handler. Then, the interrupt handler raises a signal which is statically assigned to an ISR which is an aperiodic task by default. It is the responsibility of the ISR-designer to make sure that the ISR does not have a negative effect on Asterix.

```
void int_handler( void )
{
   invokeISR(signal_number);
}

void ISR( void *ignore )
{
   Do_something();
}
```

Figure 5.4: Pseudo-code of the default interrupt-handler and the corresponding task. The interrupt handler raise a signal to wake up the interrupt service routine. The interrupt service routine is a aperiodic task that can be preempted.

By letting the default interrupt-handler raise a signal, the designer of a system can easily connect interrupts with tasks thus creating an aperiodic task (waiting for a signal/interrupt). Although a handler is supported by the kernel, a user can rewrite it to support their specific system (interrupt-controlled device-drivers etc.). This solution has already been implemented [15] [2] and the result is satisfactory [1].

The kernel does not fully support interrupts, since this is hardware specific and not yet supported by the Obelix Configuration Tool, see section 8. A pseudo-code of the default interrupt handler is shown in figure 5.4.

Chapter 6

Hardware

The current version of the Asterix kernel is implemented on the Lego Mindstorm system. This chapter describe the internals of this embedded system and what kind of difficulties the hardware architecture has caused.

6.1 Description

The Lego Mindstorm is a complete set of bricks which can be used to build and control robots, etc. Several different sensors and actuators exist for this system which makes it very suitable for educational purposes. The brain of the Mindstorm is the RCX-unit. The RCX consists of a LCD-display, four buttons, an infra-red transceiver and it has three inputs and outputs, that can be controlled simultaneously. The inputs are analog which allows users to create their own customized actuators. The outputs are designed to control motors designed by LEGO.

The RCX is based on the single-chip Hitachi H8/3292 [5] microcomputer which is a RISC architecture running at 16 MHz with eight 16-bit registers. This particular series has 16 kBytes of *read-only memory* (ROM) and 512 bytes of *random-access memory* (RAM) on-chip but with an additional 32 kBytes of RAM in a separate circuit. Also located on-chip are one 16-bit timer, two 8-bit timers, a watchdog-timer, a *serial communication interface* (SCI), an 8-channel 10-bit *analog-digital converter* (ADC) and 43 input/output lines. The ROM includes several functions for reading the ADC-channels, controlling the motors, display segments and numbers on the LCD-display. Additional mathematical functions are also included in the ROM.

6.2 Problems

The Mindstorm System is not really intended to be altered or modified. The designers of the RCX wanted it to be simple to work with, and not being replaced with some other application like Asterix. This cause a few problems; somehow the existing software needed to be replaced with Asterix, no debug tools were available and finally, the ROM and the hardware had to be overridden so that it not could affect Asterix in a negative way. The next sections, the focus will be on explaining these problems and how they were solved or neglected.

6.2.1 Replacing existing programs

First of all, the RCX uses an interpreting language which means that a interpretor is downloaded before the actual program. When the interpreter has been downloaded, the program is sent to the interpreter whom translates the byte-code to hardware-instructions. This solution has some similarities with a Java Virtual Machine [7] but is much more simple. But to convert a complete system into byte-code is not an option in a real-time operating system. Instead the interpreter needs to be replaced with another application, such as a compiled Asterix system. These problems had already been solved and are fairly well documented [10].

6.2.2 Debugging

A major problem is that the RCX is really an embedded system, which allows very few alternatives for debugging. The LCD and the motors were the only possibilities due that no communication with the RCX was available at that moment. The majority of the source-code to Asterix were debugged and verified on an ordinary PC-system so that only the really low-level parts of the code needed to be debugged on the RCX. The only possibility to debug the low-level code was to run it on the RCX and try to display as much information on the LCD as possible. With that information, pens, and lots of paper, the cause of the errors could be traced backwards to find the source.

6.2.3 Architecture

Perhaps the most difficult problem with the RCX is the hardware, the ROM in particular. Even though there exists a few disassemblations of the ROM [13], problems still occur due that the ROM handles interrupts and other hardware initializations such as timers, ADC and I/O-ports. A re-initialization of each device is possible, but changing the interrupt-vectors were not that simple, hence a general interrupt-handler starts for all interrupts and runs the address located in the interrupt-vector. This forces all interrupt-handlers in the interrupt-vectors to be a *return from sub-routine*

instead of a *return from exception* as in the normal case. The implementation of the context switch is an extension to the existing timer interrupt. This and the fact that the program stored in ROM is not developed to support third party solution, makes it difficult to make a kernel with efficient code. A major problem from the real-time aspect is that the timer-interrupt has lower priority than the external interrupts. The external interrupts are connected to two buttons. This means that if a button is pressed and a ISR is configured in the system, the timer interrupt will be masked off and thereby delaying updates of tasks etc.

Chapter 7

Analysis

One part of this thesis was to be able to calculate all the overhead caused by Asterix. In this chapter, analysis of the kernel and the *Operating System* (OS) are given. Since the OS and the kernel is separated, the analysis will also be separated. The analysis includes both jitter reduction and memory consumption.

7.1 Kernel

The kernel will always be included in an Asterix system, even if the application consists of only one task. In this section, formulas are presented to calculate the memory usage and the execution-time of the kernel.

7.1.1 Jitter and Execution-time

In order to achieve a predictable kernel, the first thing to do is minimize or, even better, eliminate the jitter. This requires that the programmers of the kernel writes code in such a way that the execution time of each function is identical each time the function is called, regardless of the current time and state of the system.

The kernel should also be easily ported to other hardware platforms. The decision were made to use ANSI-C (ISO/IEC 9899:1990) extended with inline assembler for low-level programming. All compiler-optimizations of the code are forbidden just to ensure the programmers that no rearrangements of the code is performed by the compiler. Another thing that the compiler must fulfill is that it is not producing any jitter when compiling, including all low-level-operations (arithmetical, logical, etc.). *The Gnu C Compiler* (GCC) [12] was chosen because it is open-source, and it is available on a large amount of hardware platforms.

One problem with the design and implementation of Asterix is that all error-routines runs in kernel-mode 4.3.9. This generates jitter every time an error occurs. This problem can be solved by letting a task run its error routine as a task instead of within kernel-mode. This technique is given more detailed in section 8.

A general formula, Ψ, is used to describe the execution-time for a task τ. Ψ adds the internal execution-time (λ) of τ with the execution-time for each task π, where $\pi \in \forall$ functions $\mathcal{F}$ called by τ.

$$\Psi(\tau) = \lambda + \sum_{\pi \in \mathcal{F}}^{\eta} \Psi(\pi) \quad (7.1)$$

With the function Ψ (7.1), along with the necessary jitter demand, each call within the kernel can be measured and summarized so that the designer of the system knows what overhead every function (task-switches, interrupts, etc.) in the kernel takes. This overhead is very useful for latency calculations of the tasks.

7.1.2 Memory

Beside the jitter minimization, the memory requirements should be reduced. A few approaches were considered to fulfill this but all have not been implemented, mostly because this demands a much more advanced configuration tool than the one that exists today [3]. To find exactly how large the kernel is, its size needs to be measured on the actual hardware platform since data-types might differ from hardware to hardware, but the source-code of kernel consists of approximately 1100 lines of C-code (all comments and file-headers included, but all initializations of tasks are excluded). In this analysis, the kernel was measured on the Hitachi-H8 hardware platform, see chapter 6, with a task-set that varied from one to two-hundred tasks.

As seen in the unoptimized part of figure 7.1, the amount of memory needed for five tasks ends up at around 3.5 kBytes (3498 to be exact). In the current implementation, the size of a post in the TCB-List 4.3.3 is 18 bytes (9*16 bits), the size of a post in Displist 4.3.3 is 26 bytes (13*16 bits) and one position in the Tasklist is 2 bytes (16 bits). These numbers can be derived directly from the definitions in section 4.1. One more thing to keep in mind is that the size of readyqueue will also change with the amount of tasks in the system, see section 4.3.5 and formula 4.3. Now, all the necessary information about the kernel-overhead for n_t task can be defined by the function Ω in formula 7.2.

$$\begin{aligned} k &= \sum_{i=0}^{n_t} \sigma(TCB) + \sigma(Dispatch) + \sigma(taskid) \\ \Omega &= \sigma(\Delta) * R_q + k \end{aligned} \quad (7.2)$$

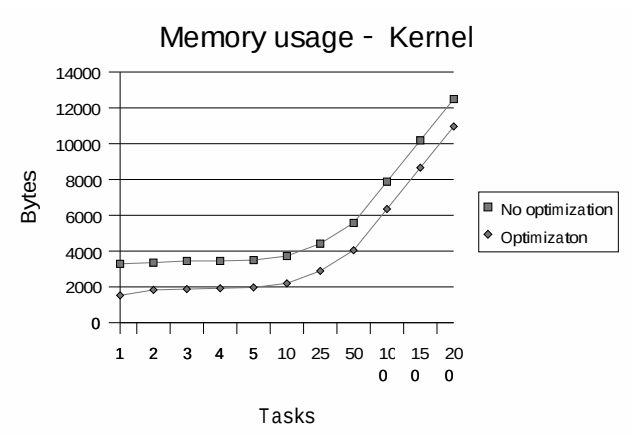

Figure 7.1: Memory usage of the kernel with various number of tasks.

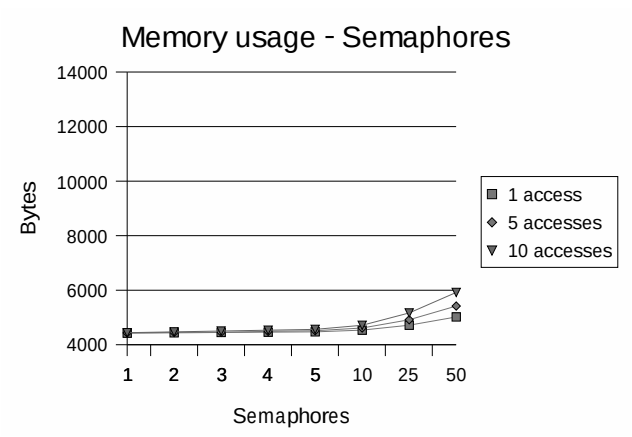

Figure 7.2: Memory usage with various sets of semaphores.

The calculated kernel-overhead in a system with five tasks is 232 bytes which indicates that the raw kernel uses 3266 bytes (3498-232). For a system consisting of two-hundred tasks, the calculated overhead would be 9226 bytes. The measured size of the entire kernel with two-hundred task is 12492 bytes. If we subtract the calculated overhead of 9226 bytes from the measured size, the result is 3266 bytes that would represent the raw kernel.

Even if the raw size of the kernel is identical regardless whether five or two-hundred tasks were used, one noticeable matter that even if GCC is forced to disable all optimizations, some optimizations are still performed. This results in that the raw size of the kernel may not be 100% accurate, but it gives the user a hint of the size. One interesting thing is that if the kernel is compiled with optimization, the amount of memory needed is about half the size of the case without any optimization. This indicates that all additional code that were added to minimize the execution-time jitter has been reduced and joined with the useful code. The problem now is that the jitter may have increased depending on the compiler.

7.2 Operating system

Most designers will require more functionality besides the basic kernel. This will of course cost in the terms of execution-time and memory usage. One major problem is that all jitter and memory overhead caused by the OS are dependent on two factors:

- The number of resources.
- The number of tasks sharing a resource.

Even if the jitter is depending on the two factors above, the design of Asterix is created in such a way that the system-calls for obtaining resources are always executing the worst case, e.g. always traverse the internal lists for that resource.

7.2.1 Semaphore

Since the Asterix kernel is using protection against unauthorized access of semaphores, the amount of memory needed from one semaphore to another differs only by the number of tasks that can access the semaphores. The definition of a semaphore, given in section5.1, and the task position added in the Tasklist would be enough to calculate the size but this is only useful from the kernels point-of-view. To make the usage of semaphores a bit more user-friendly, an additional variable that represents the identity of a semaphore is needed for each semaphore. The Obelix Configuration Tool adds extra variables to be able to generate the lists needed by a semaphore. This results that the total overhead per semaphore will be a bit larger than actually required by the kernel. The total cost of adding one semaphore with n accesses is defined by Υ in formula 7.3. The unnecessary overhead is presented within brackets.

$$\Upsilon(n) = (n * \sigma(taskid) + 2 * \sigma(taskid)) + \\ + [2 * \sigma(taskid) + \sigma(semid)]) \qquad (7.3)$$

But the needed memory can be a bit larger since every semaphore in the Asterix kernel needs to add one position to the readyqueue, see section 4.3.5. But this cost is only depending on the number of semaphores, not how may access each one of them grants, so that overhead is calculated by the kernel-overhead, see Ω in formula 7.2 and R_q in formula 4.3..

A set of 25 semaphores all with five accesses will need 500 bytes of memory (25 * $\Upsilon(5)$). If these 500 bytes are subtracted from the 4918 as given in figure7.2, the result would be the penalty for introducing semaphores in the system. But since the semaphores, in this case, also increase the size of the readyqueue, an additional 4 bytes needs to be subtracted. This results in a total of 4414 bytes. The measured 25 semaphores is part of a system with ten tasks, which already uses 3728 bytes of memory. Two bytes are removed from this due the readyqueue is not

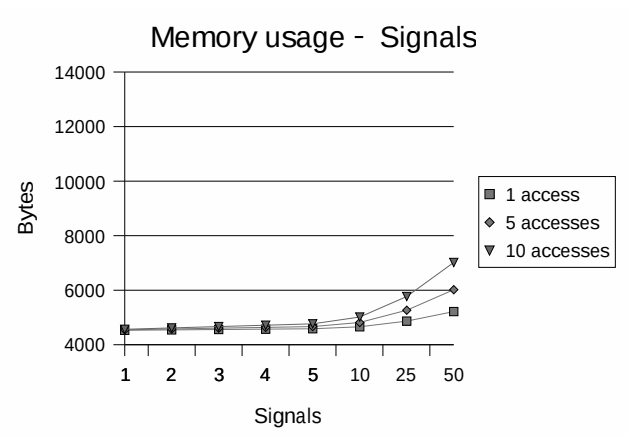

Figure 7.3: Memory usage with various sets of signals.

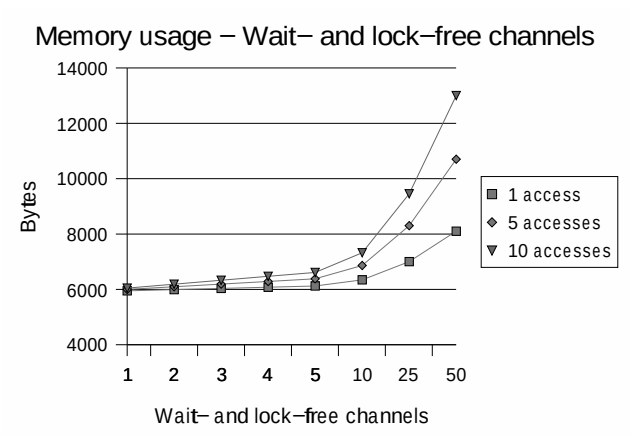

Figure 7.4: Memory usage with various sets of wait- and lock-free channels.

needed in the calculation. The penalty for introducing semaphores, regardless how many, will be 684 (4918-500-4-3728-2) bytes on the Lego Mindstorm-system.

7.2.2 Signals

The signals are very similar to the semaphores, but they differ on one point; each signal keeps record of all tasks that is currently waiting on it. On the other hand, no changes in the readyqueue or any other list is needed. This means that it is easier to calculate the amount of memory needed both for the general penalty for introducing signals, and for calculate the overhead that each signal cause. The overhead for each signal can easily be derived for its definition, see section 5.2, and thus present a formula, Γ shown in formula 7.4, that calculates the amount of memory that each signal needs for n tasks that can access the signal. As in the semaphore-case, OCT adds one reference to the signal for easier access, but it adds two extra variables to generate the owner-list and the block-list. This extra overhead is given within brackets in formula 7.4.

$$\Gamma(n) = (2 * \sigma(taskid))(n+1) + \\ + [(2 * \sigma(taskid)) + \sigma(sigid)] \qquad (7.4)$$

The ability to calculate the overhead for a single signal, along with the information about the kernel-overhead for a system with ten tasks, is enough to calculate the general penalty for introducing signals in Asterix. 25 signals with five accesses would require 750 bytes out of the 5268 bytes measured (figure7.3), leaving 4518 bytes left. Since the signals have no impact on any of the lists used by the kernel, it is enough to subtract the general kernel-overhead from the previous calculation. The one-time penalty for signals is 1252 bytes (4518-750-3266). The reason that this is larger than in the semaphore-case is because more list-operations are needed for the block-list in particular.

7.2.3 Wait-free communication

A simple form of wait-free communication is implemented in Asterix, but it is not as well designed as the signals and semaphores. The reason is that this part was a separate project but the person working on that part unfortunately chose to leave the project. Based on his work, the decision to implement a simple but very memory consuming communication were made. Since linked lists are used instead of arrays and control-blocks per task instead of per wait-free channel, it is very difficult to predict the behavior and present a general formula to calculate the amount of memory needed on pre-runtime basis. The measured values in figure7.4 confirms the large amount of memory needed to use wait-free communication in the system.

Chapter 8

Future work

This document is a description of Asterix, the real-time kernel. Since the current implementation is just a prototype, improvements can be made. Suggestions of improvements are described first in this chapter followed by some thoughts and ideas of further extension of the kernel.

8.1 Improvements

The current implementation has some limitations. In this section, a description of these limitations and how some of them can be removed is given.

8.1.1 Interrupts

Interrupts are hardware-dependent but they have the same functionality regardless of the hardware. The design of interrupts are designed well in the kernel but the initialization and how they can be accessed and altered by the user must be redesigned, both in the Asterix kernel and the Obelix Configuration Tool (OCT). The simplest thing to do would be to let the user write the interrupt-handler or use the default for all interrupts, and let OCT connect these routines with the initialization-phase in the kernel. OCT must also have access to a list of valid interrupts and thereby make the right connection to the interrupts via the kernel. Hence some sort of hardware description-file is needed.

8.1.2 Wait- and Lock-free channels

The current implementation of Wait- and Lock-free channels (WLFC) is very memory consuming due that it has not been analyzed in the same way as the other extensions of the kernel (signals, semaphores and interrupts). The reason is that linked lists are used to determine which buffer inside the channel is the oldest and latest. The linked list can be replaced with timestamps or some other mechanism that can sort elements in chronological order. Another possibility to minimize the consumption of memory is to have a control block per WLFC instead of a control block per task. The problem is to update a tasks' corresponding buffer if the task have access to multiple WLFCs.

8.1.3 Signal extension

A signal is used in several purposes in the Asterix-prototype; Synchronization, aperiodic tasks and interrupts. A slight disadvantage is that an aperiodic task cannot wait for other signals due that it always connected to its wakeup-signal and will be activated every-time this signal is raise. This means that all aperiodic tasks will never be removed from any waiting-list in the signals and hence every-time it waits for another signal, it will never be removed from that list either. This will eventually cause a system failure since memory will be overwritten when a waiting-list is full.

8.1.4 Execution-time measuring

Only execution-time of tasks are available at the moment. It would be nice if different elements (task-switches, system-calls etc.) in the kernel could be measured as well. Such a measurement requires that the system remains unchanged regarding the number of resources and tasks since the execution-time within the kernel is different depending on the tasks and resources in the system.

8.2 Next generation

In this section, some thought and ideas to improve Asterix are given. These ideas are extensions of the existing kernel and changes in both the kernel and the configuration tool might be necessary. The ideas that are presented are focused on the kernel and not the entire Asterix Framework. Future work on the framework would be to create an analysis-tool and a communication layer to support a distributed system based on the Asterix kernel.

8.2.1 Minimize memory

Since the kernel is static and compiling, the largest data-type needed for each purpose is known. This means that some extra memory can be saved if correct data-types would be chosen. For instance if no task have a *period* (T) larger than 250, it would be a waste of memory to let the data-type to hold T to be 16-bits, when a 8-bit data-type would do just fine. This requires that the configuration tool will detect and set these data-types. But this will not require any

changes in the kernel since everything is taken care of by the compiler.

8.2.2 Monitoring

The kernel is designed in such a way that an implementation of Deterministic Replay could be done relatively easy. An example of this is the TCB-list that is separated in two. The actual implementation would require to communicate with the kernel so all necessary information about task-switches etc. can be uploaded and debugged off-line.

8.2.3 Soft tasks

Soft tasks must be treated in a certain way that the hard tasks are unaffected. One major problem is that all error-handling is executed in kernel mode. This result in that a system-tick will be longer when an error has occurred. It would be better if the soft tasks error-routine executes in user-mode and thereby not affect the time-constraints in the system.

8.2.4 Execution-time jitter on tasks

The kernel can be forced to always run tasks *worst case execution time* (WCET) and thereby increase the predictability. The kernel need to be make two major changes;

- Add every tasks WCET to the task-model.
- Make sure that no task-switches are performed even if a task has terminated.

The problem is to determine how long the kernel will wait between a termination of a task and the start of the next. But this functionality can easily be derived from the system-timer although the resolution may not be enough.

Chapter 9

Conclusions

It takes a long time to develop tools, real-time kernel and operating system. This document describes the beginning of the process of developing a real-time kernel. To be successful when designing and implementing a small-sized real-time kernel, it is important to have suitable development tools. By letting the kernel and the tools collaborate so functionality usually performed by the kernel is moved to the tools. Thus optimizations can be done off-line. This is essential in embedded systems where computational power and amount of memory are limited.

The Asterix kernel supports state-of-the-art methods for mutual exclusion, interprocess communication and synchronization. The development tools have a significant role, for example when memory areas are allocated and initializing of these areas. Even more the importance of analyzing tools (not developed yet) are substantial for easy use of the Asterix Framework.

There are trade-offs to consider when implementing state-of-the-art methods. The algorithms must be as simple as possible to reduce the execution time, but yet perform what it is supposed to. The Immediate Inheritance Protocol is an example of such algorithm, another is the implemented Wait- and lock-free channel algorithm.

The kernel is designed is such way that future implementation of monitoring can be made. Specifically the Task Control Block is designed to support monitoring by Deterministic Replay. In this first version one of the major problem have been the absence of a debugging tool. During the development, a trial-and-error method have been the debugging tool. Hence the possibility to use Deterministic Replay is preferable.

The Asterix real-time kernel is designed with focus on reducing execution-time jitter and memory consumption. Low jitter and low memory consumption together with statically allocated resources makes the kernel predictable and so that its overhead can be calculated. There are trade-offs in jitter reduction and memory usage, for example reducing jitter are often solved with additional code. To reduce the memory usage, arrays are used instead of linked list and thereby the identity and priority can be combined with the index to the array to save even more memory. The most reduction of memory consumption is achieved with the design of the readyqueue. The amount of memory needed by this queue is only one bit per task. The fact that a task have its own stack has a big impact on memory usage. This can be solved but not without major changes of the design.

Jitter minimization of the kernels execution-time is difficult to achieve, since this depends on both programming skills and the behavior of the chosen compiler. For example; code optimized by a compiler can increase the jitter. Not all types of optimizations benefit the system. It is an advantage if the behavior of the compiler can be examined. Therefore a compiler shipped with source code is to be preferred.

To summarize the work, we must say that even though not all thoughts and ideas have been successfully implemented, the result have been quite satisfactory. The kernel is still a prototype but fully functional can thereby be used for demonstration and educational purposes. The design is simple and straight-forward so that any redesign can be done with little effort.

Bibliography

[1] G.C. Buttazzo. Hard real-time computing systems. *Kluwer Academic Publishers,ISBN 0-7923-9994-3*, 1997.

[2] G.C. Buttazzo and M. Di Natale. Hartik: a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution. *In proceedings of IEEE International Conference on Robotics and Automation, May 1993*, 1993.

[3] A. Davidsson and J. Lindgren. Obelix development environment. *Master Thesis, Mälardalens University, Department of Computer Engineering, Västerås June 2000.*, 2000.

[4] H. Hansson. Real-time scheduling. *Lecture notes, Mälardalen Real-Time Centre, 1998*, 1998.

[5] Hitachi. Hitachi single-chip microcomputer h8/3297 series. *Hardware manual, 3rd edition.*, 2000.

[6] M. Jones. What really happened on mars rover pathfinder. *The Risks Digest, Volume 19, Issue 49, Dec 1997*, 1997.

[7] L.Q. Li and M. Singh. Java virtual machine - present and near future. *Technology of Object-Oriented Languages 1998*, 1998.

[8] L. Lindh, T. Klevin, and J. Furunäs. Sara scalable architecture for real-time applications. *CAD and CG'99, Shanghai, China. Dec 1999*, 1999.

[9] F. Mueller, V. Rustagi, and T. P. Baker. Mihtos, a real-time micro-kernel threads operating system. *IEEE*, 1995.

[10] M.L. Noga. legos documentation and source-code 0.1.7. *legOS*, 1998.

[11] C. Norström, C. Sandström, J. Mäki-Turja, H. Hansson, and H. Thane. Robusta realtidssystem. *Mälardalen Real-Time Research Centre*, 1999.

[12] GNU's not UNIX. The gcc homepage. *http://www.gnu.org/software/gcc/gcc.html*, 2000.

[13] K. Proudfood. Rcx internals. *Stanford University, Computer Graphics Lab*, 1999.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.*, 1990.

[15] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE*, 1991.

[16] H. Hansson H. Lawson O. Birdal C. Eriksson S. Larsson H. Lon M. Stromberg. Basement: An architecture and methodology for distributed automotive real-time systems. *IEEE Transactions on Computers, vol. 48 9*, 1997.

[17] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. *In proceedings of the 12th Euromicro Conference on real-time systems , Stockholm, June 2000.*, 2000.

[18] H. Neugass G. Espin H. Nunoe R. Thomas D. Wilner. Vxworks: an interactive development environment and real-time kernel for gmicro. *TRON Symposium, Proc. Eighth, pp. 196-207*, 1995.

[19] K. M. Zuberi and K. G. Shin. Emeralds: A microkernel for embedded real-time systems. *IEEE*, 1996.