Master's thesis

Analysable Hard Real-Time Systems

Thomas Hedemand Nielsen Jens Christian Schwarzer

September 3, 2001

Informatics and Mathematical Modelling
Technical University of Denmark

This document was created with the LaTeX 2_ε text formatting system

Abstract

The concept of hard real-time systems is introduced, and an informal reference model for hard real-time systems is defined. The theory of priority-driven schedulers is presented, and the problem of validating a schedule is addressed. Synchronisation protocols are introduced to control blocking due to resource contention. A commercially available real-time operating system is investigated. Based on scheduling theory a computational model is defined, and implemented upon the particular operating system. A development process for hard real-time systems is proposed. The design phases of the proposed development process are applied to a case study.

Keywords

Hard real-time development process, analysable hard real-time systems design, computational model, fixed-priority scheduling, schedulability analysis.

Preface

This master's thesis is the result of a project at Informatics and Mathematical Modelling, Technical University of Denmark. The work was carried out in the period March 5 to September 3, 2001, under supervision of associate professor Hans Rischel.

Acknowledgements

We would like to thank our supervisor, Hans Rischel, for his help and support during the project. We especially appreciate the many constructive and entertaining discussions we had during our meetings.

We would also like to thank Torben Hoffmann for providing concise and relevant comments on our thesis.

Furthermore, we would like to thank Anders Thurah Nielsen, Critical ApS, for the inspiration and ideas that formed the initial project definition. Additionally, we would like to thank Anders Thurah Nielsen and Lars Vange Jørgensen, Critical ApS, for providing our execution environment, consisting of the Nucleus Plus operating system and the Infineon C164 microcontroller, and for readily answering all our questions.

Finally, we would like to thank Bente Kuhlmann for her readings of our thesis.

Kgs. Lyngby, September 3, 2001

Thomas Hedemand Nielsen & Jens Christian Schwarzer

Contents

List of Figures

List of Tables

1 Introduction

A *hard real-time system* is a system whose correctness depends not only on the logical result of computation, but also on the time at which the results are produced: it must satisfy explicit response-time constraints or risk severe consequences, including failure [Stankovic and Ramamritham, 1988]. Thus, computations in a hard real-time system are required to complete before a *deadline*.

Real-time systems span from safety-critical systems controlling large power plants, over signal processing systems like radar systems, to multimedia applications decoding video streams and displaying them on a screen.

A digital *control system*, as illustrated in figure 1.1, is an example of a conceptually simple real-time system. The control system receives a *reference input* indicating a desired state of the *controlled system*. The control system may monitor the state of the controlled system by means of *sensors* and the control system can change the state of the controlled system by means of *actuators*.

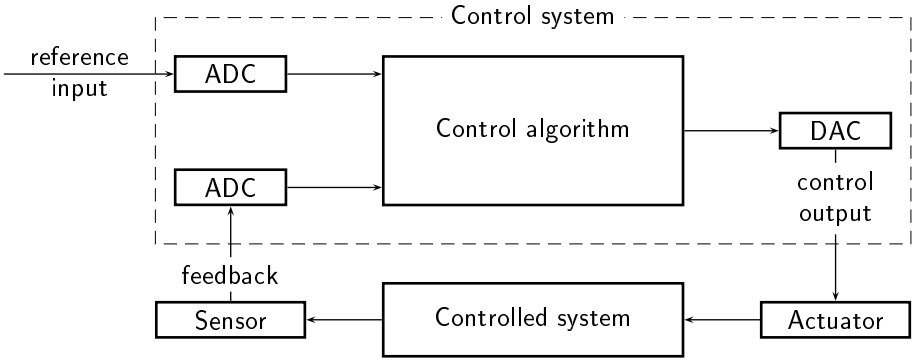

Figure 1.1: *A digital control system.*

For a control system, timing constraints may be derived from requirements on the responsiveness of the sensors and actuators used for monitoring and control. Thus, for the control system, deadlines may be imposed on the computations reading the reference input and system state, the computation of a new actuation value according to a control algorithm and the computation outputting the value to the actuator. If it is possible at all for computations to meet their deadlines, it is crucial that they are scheduled in a way which guarantees that their deadlines are met.

To develop hard real-time systems efficiently, a development process that supports timing constraints must be used. The development process, in particular the design phase, must continuously assess that the system is able to meet its timing constraints. In this way the development process shall trace the development of a system that will meet all its deadlines, once built.

Given a design of a hard real-time system it must be possible to validate that the

system meets its timing constraints. The requirement for validation places many restrictions on the design and implementation of hard real-time systems. An objective of this thesis is to investigate how to restrict a design of hard real-time systems to facilitate validation.

1.1 Thesis Objectives

The primary objective of this thesis is to investigate how to build hard real-time systems guaranteed to meet their deadlines. This includes:

- The construction of an informal *reference model* of real-time systems defining and designating concepts used in the further discussion of real-time systems.
- How to design real-time systems. Based on a study of the vast amount of real-time scheduling theory we present a *computational model* that ensures the predictability necessary for validating a design. We sketch a *development process* that traces a feasible design through the individual phases of the development process.
- How to *validate* the timing constraints of real-time systems. Based on the computational model we present validation algorithms which may be used for demonstrating that timing constraints for a design are feasible.
- How to *implement* the computational model with a real-time operating system. This includes a study of the operating system with respect to its suitability for hard real-time systems.
- The presentation of a case study illustrating the use of the concepts and techniques introduced in this thesis. The case study presents a design of a motor control system of industrial size and relevance.

1.2 Thesis Outline

Chapter 2 defines the reference model for real-time systems. The definitions and designations in the model allows us to concentrate on the essential characteristics of real-time systems.

Chapter 3 presents common approaches for scheduling real-time applications consisting of independent tasks on a single processor. The validation of timing constraints for independent tasks is the topic of chapter 4.

In many real-time applications resources are shared between tasks making those tasks interdependent. This may introduce uncontrolled blocking of tasks resulting in missed deadlines. In chapter 5 we consider interdependent tasks, provide methods to control blocking and extend validation algorithms to handle blocking.

In chapter 6 we define a computational model that ensures the predictability necessary for validating a design. We explore a commercially available real-time operating system in chapter 7 and present an implementation of the defined computational model in chapter 8.

An integration of scheduling theory and a traditional development process is suggested in chapter 9. In chapter 10 the proposed methods are tested in a real world case study.

Finally, in chapter 11, we present our conclusions.

Parts of the implementation of the computational model defined in chapter 6 can be found in appendix A. Appendix B contains the source code for an application used for measuring the overhead imposed by the Nucleus Plus operating system. Appendix C features a list of acronyms used in this thesis.

2 A Reference Model for Real-Time Systems

For the purpose of describing real-time systems we form an abstract model of real-time systems. The model presents designations and definitions [Jackson, 1995], which are used in the descriptions of real-time systems in this thesis. In this thesis we will refer to the abstract model as the *reference model*. The model is based on concepts presented in [Løvengreen, 1997], [Rischel et al., 1987], [Hoare, 1985], [Liu, 2000], and [Klein et al., 1993].

The primary benefit of an abstract model is that it allows us to concentrate on the essential characteristics of real-time systems, hence we are independent of the various application areas. Having described a system in terms of the reference model, we are allowed to analyse and simulate the real-time behaviour of the system, e.g. produce accurate estimates of the real-time performance and overhead for the system.

2.1 Real-Time Systems

In the presentation of the reference model, we first define the abstract concept of a system and then a real-time system in particular. A *system* is a structured collection of *entities* of a given universe to be considered as a whole. The system boundary or the system *interface* divides the universe into those entities which are part of the system and those which belong to the rest of the universe, denoted the *environment* of the system. System entities are denoted *components*.

Interaction may occur between entities of the universe. However, when modelling a real-time system we only focus on the interaction occurring between the system components and between system components and environment entities. Interactions between entities in the environment are not considered. The interaction between components and the entities in the environment defines the *externally observable behaviour* of the system.

Timing behaviour is important to all computing systems. However, including timing behaviour in the specification of the system, and thereby in the definition of the correctness of that system, distinguishes real-time systems from other types of systems. In this thesis we shall adopt the following definition of a real-time system [Klein et al., 1993] [Stankovic and Ramamritham, 1988].

Definition 2.1 (Real-Time System) *A* hard real-time system *is a system whose correctness depends not only of the logical result of computation, but also on the time at which the results are produced.*

Many real-time systems responds to external stimuli over time, and is typically placed in its environment with the purpose of monitoring and controlling some as-

pects of the environment. There are usually timing requirements associated with monitoring and controlling each aspect. There are also work in the form of computations that must be performed in order to control or monitor the environment. Thus, a typical scenario is that the real-time system waits for a stimulus, and starts some computation in response to its occurrence. The computation must often be completed by a *deadline* that is relative to the time, the stimulus occurred.

2.2 Events

We use events to model external stimulus of real-time systems. An *event* is an abstraction of an interaction, i.e. an activity involving more than one entity or component. An event is considered to be atomic: either the activity results in an overall effect, or there is no effect at all [Løvengreen, 1997] [Hoare, 1985].

As in [Rischel et al., 1987] we let the term event denote a class of uniform single events. Thus, we assume that a system processes a collection of single events which can be divided into a number of classes. A specific event occurring in a system is a member of an event class.

Events may be partitioned into three groups:

Input events carry information across the system boundary from the environment to the system.

Output events carry information across the system boundary from the system to the environment.

Internal events carry information between system components.

Additionally, events are characterised by their *arrival pattern*, that is the pattern of occurrence as a function of time. In this thesis we consider the following kinds of event arrival patterns:

Periodic events are characterised by a constant time interval between the arrival between two consecutive events. The length of the interval is denoted the *period*.

Aperiodic events which can be subdivided even further [Klein et al., 1993]:

Irregular events	Events arrives with known intervals that are not constant.
Bounded events	Events have a minimum inter-arrival interval or a maximum arrival rate.
Bursty events	Events do not exceed a specific *event density*, which consists of a *burst interval* and a *burst size*.
Unbounded event	Event arrivals are described in terms of probability distribution functions.

2.3 Jobs and Tasks

To motivate the abstract notions of a job and a task in the reference model, we reconsider the control system in figure 1.1. The input data, i.e. reference input and system state, is obtained from the environment. For the system to operate properly, the input data acquisition and execution of the control algorithm must be performed repeatedly. Hence, both computations could be performed in response to two timer events with a periodic arrival pattern.

To distinguish a single execution instance of the computational work performed in response to an event from the system function performed by the repeated executions of the computational work, we introduce the notions of jobs and tasks.

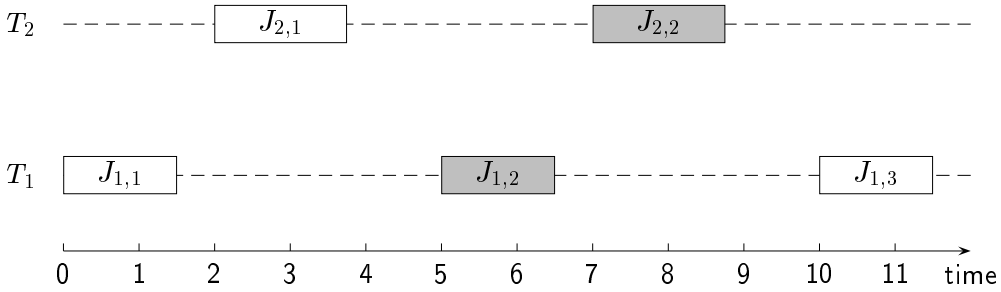

Figure 2.1: *A job is an execution instance of a task. The execution of individual jobs in two tasks T_1 and T_2, may be illustrated with a time line diagram. Consecutive jobs in a task is shaded differently.*

A *job* is an abstraction of the computational work that is performed in a response to a single occurrence of an event. The term *task* denotes a set of related jobs which jointly performs some system function.

Over time a task is a sequence of jobs that are performed in response to an event. For the control system we may illustrate the situation with a *time line diagram*. In figure 2.1 the task of acquiring input data is denoted T_1, and the task of executing the control algorithm is denoted T_2. For each task a time line is placed above the time-axis. When a job is being executed an indication in the form of a rectangle is drawn on the time line of the task. Consecutive jobs in a task is shaded differently. We only consider a single processor, hence only one job can execute at any time.

At the time an event arrives, the corresponding job is *released* and becomes *eligible* for execution. The arrival time of the event is the *release time* for the job.

We shall denote the set of tasks in a system $\mathbf{T}$. For the cardinality n, where $\text{card}(\mathbf{T}) = n$, the task set contains the tasks $T_1, T_2, \ldots, T_n$. The individual jobs of task T_i is denoted $J_{i,1}, J_{i,2}$ and so on, $J_{i,k}$ being the kth job in T_i. The naming convention is illustrated in figure 2.1. When we are discussing properties of individual jobs but are not interested in the task to which they belong, we simply denote the jobs J_1, J_2, and so on. Using the naming convention of jobs, the release time of the kth job in T_i is denoted $r_{i,k}$ and r_k. The *inter-release time* is defined as the time interval between the release of two consecutive jobs in a task.

A job J_i is characterised by its *execution time*, e_i, i.e. the amount of time required to complete the job, when it executes alone without having to wait for any required

resources. The execution time of a job may vary for different reasons, e.g. it may depend on the job's input data. For a task T_i the execution time is the maximum execution time of all jobs in it.

2.4 Timing Constraints

A constraint imposed on the timing behaviour of a job is a *timing constraint*. A timing constraint considered in this thesis shall be specified only in terms of the release and completion times of a job.

The release time $r_{i,1}$ of the first job $J_{i,1}$ in task T_i is the *phase* of the task, and is assigned the symbol ϕ_i. Thus, for task T_i we have $\phi_i = r_{i,1}$. Different tasks may have different phases. Two tasks are said to be *in phase* if they have the same phase.

A job J_i released at time r_i must complete D_i units of time after r_i. Thus, D_i is the *relative deadline* of the task T_i. Thus, the worst-case response time W_i of the job is D_i. The *absolute deadline* of the job is $d_i = r_i + D_i$.

A deadline may be characterised as:

Hard deadline A job with a hard deadline must always complete its execution before its deadline.

Soft deadline A deadline is soft when it shall only be respected on *average*.

No deadline This is included in the reference model as some tasks may exist in a real-time system that, although they affect the timing of other tasks in the system, have no timing constraint of their own. Such a task is denoted a *background task*. Though a background task have no deadline, the task must must be allowed to execute eventually.

In this thesis we shall only consider tasks with hard deadlines.

A periodic task may require that a job released in response to a periodic event is performed without *jitter*. Such a job is usually related to an input or an output operation. Jitter is a measure of deviation between the desired time for the operation and the actual time the operation is performed. Thus, jitter is related to the completion times of the jobs in a periodic task.

Jitter is a consequence of the fact that a job may not start to execute immediately upon its release. Hard deadlines may be used for controlling jitter. In the extreme case of a relative deadline equal to the constant execution time of a periodic task, $D_i = e_i$, no jitter is allowed.

2.5 The Periodic Task Model

The *periodic task model* is a well known deterministic workload model proposed in 1973 by Liu and Layland [Liu and Layland, 1973] [Liu, 2000]. In its original form, the model was restricted to the strictly periodic case, where all tasks are responses

to periodic events. Over the years the model has been extended so that it is now capable of characterising many traditional hard real-time applications, including those with aperiodic activities.

Periodic Tasks

In the periodic task model, each task T_i is a sequence of jobs performed in response to an event with a periodic arrival pattern. In the following we shall simply use the term *periodic task* to denote such a task. The *period* p_i of the task is the length of the time interval between the release times for two consecutive jobs in the task. Thus, for all jobs in a periodic task the inter-release time between the job and the consecutive job is equal to the period of the task.

At any time, the periodic task model assumes that the period and execution time of all tasks in a system are known.

The *utilisation* u_i of a task T_i is the fraction of time a strictly periodic task with execution time e_i and period p_i occupies a processor.

$$u_i = \frac{e_i}{p_i} \tag{2.1}$$

The *total utilisation* U of the tasks in a system consisting of n tasks is the sum of the utilisations of the individual tasks.

$$U = \sum_{i=1}^{n} u_i \tag{2.2}$$

We will often assume that for every task a job is released and becomes ready at the beginning of each period and must complete before or by the end of the period. Thus, $D_i \leq p_i$ for all tasks. This restriction actually states a throughput requirement: the system shall keep up with all the work demanded of it at all times. In general, the relative deadline can have an arbitrary value.

Aperiodic Tasks

In the periodic task model, the workload generated in response to aperiodic events is modelled by *aperiodic tasks*. Each aperiodic task is a sequence of jobs, where each job is the computational work performed in response to an event with an aperiodic arrival pattern.

Like a periodic task, an aperiodic task is characterised by its execution time, phase, and relative deadline. However, due to the aperiodic arrival pattern of the event to which the jobs in an aperiodic task respond, the inter-release intervals for a sequence of jobs can vary. In particular the inter-release times may be arbitrarily small, thus increasing the total utilisation of the system.

2.6 Processors and Resources

A number of system resources may be available for the tasks of a real-time system. All resources are divided into two major types: *processors* and *resources*.

A processor P_i is an abstraction of a central processing unit (CPU) or a microcontroller, and it is the most common shared resource of a software system. Processors carry out instructions and move data. Every job requires a processor to execute and make progress toward completion. An attribute of a processor is its *speed*. Though this attribute is rarely mentioned, it is implicitly assumed that the rate of progress a job makes toward its completion is a function of the speed of processor, on which it executes.

The decisions on how a processor is assigned to individual jobs are performed by a *scheduler*. We return to schedulers in section 2.7, and give a thorough presentation of schedulers in chapter 3. In thesis we only consider systems with a single processor.

As opposed to processors resources R_i are passive. A job may need some resources in addition to a processor in order to make progress. Examples of resources are data objects and peripherals. Often, resources are critical regions that must be accessed in a mutual exclusive manner.

Synchronisation Protocols

When resources are shared in a mutually exclusive fashion between concurrent tasks, a task can be blocked by another task.

In the value-domain the blocking introduced by requirements for mutual exclusion ensures the integrity of an application. However, in the time-domain the integrity of a hard real-time application may be compromised by blocking.

The access of a resource is controlled by a *synchronisation protocol*. If resources are shared in a hard real-time application, a synchronisation protocol providing a predictable blocking delay shall be applied. An in-depth introduction to synchronisation protocols is given in chapter 5.

2.7 Scheduling

Every task $T_i \in \mathbf{T}$ is associated with an unique priority π_i.

$$\forall T_i, T_j \in \mathbf{T} \cdot T_i \neq T_j \Leftrightarrow \pi_i \neq \pi_j$$

All the jobs in a task T_i are assigned the priority π_i of the task. In this thesis priorities are represented by integers; the smaller the integer, the higher the priority. We introduce a new operator, $\succ$, to indicate this relation. For example, if a task T_1 has higher priority than a task T_2, i.e. $\pi_1 = 1$ and $\pi_2 = 2$, we have that $\pi_1 \succ \pi_2$. In general we have

$$\pi_i < \pi_j \Leftrightarrow \pi_i \succ \pi_j.$$

Concurrent activities, i.e. jobs, are competing for the same resources, in particular for the processor. In this section we shall introduce the terminology associated with the allocation of the processor and the resources to jobs.

Among the eligible jobs a job is chosen for execution and allocated resources according to a chosen *scheduling algorithm* and a set of synchronisation protocols

associated with the resources. An implementation of the scheduling algorithm is denoted a *scheduler*.

A scheduler may be either *preemptive* or *non-preemptive*. Preemptive means that the execution of any job can be interrupted by the execution of a higher priority job. When preemption is allowed, a scheduling decision is performed whenever a job is released or completed.

We say a job is *scheduled* in a time interval on a processor, if the processor is assigned to the job in that interval. A *schedule* is an assignment of all the jobs in the system to the processor. A schedule is produced by a scheduler.

As in [Liu, 2000] we shall not question the correctness of the schedule. Thus, we assume the scheduling algorithm and the scheduler are correct and produce only *valid schedules*. A valid schedule satisfies the requirements below:

- Every processor is assigned to at most one job at any time.
- Every job is assigned at most one processor at any time.
- No job is scheduled before its release time.
- Depending on the scheduling algorithm used, the total amount of processor time assigned to every job is equal to its maximum or actual execution time.
- All the precedence and resource usage constraints are satisfied.

For brevity, we shall often use the term scheduling algorithm or simply algorithm to denote the scheduler implementing the scheduling algorithm.

Feasibility of Schedules

A valid schedule is *feasible* if every job scheduled in it meets its deadline. A set of jobs is said to be *schedulable* according to a scheduling algorithm, if that algorithm always produces a feasible schedule for the set.

Optimality of Schedulers

The main criterion for evaluating a scheduling algorithm is its ability to find feasible schedules for a given set of tasks, whenever such schedules exist.

A scheduling algorithm is said to be *optimal*, if and only if the scheduling algorithm always produces a feasible schedule for a set of tasks, when a feasible schedule exists. Thus, if an optimal scheduler fails to find a feasible schedule for a given set of tasks, we can conclude that no feasible schedule exists for it, and that the set of tasks cannot be scheduled by any other algorithm.

2.8 Representing Real-Time Situations

The definitions and designations of the reference model allows us to describe real-time systems with a standard terminology. To facilitate a very compact representation of the timing aspects of a real-time system we introduce the concept of a *situation table*.

A situation table represents the essence of a real-time situation. It associates each task in a real-time system with is parameters in a tabular form. Rather than defining multiple instances of situation tables, we take a very pragmatic approach with respect to the contents of situation tables. A situation table simply summarises the real-time related information that is relevant in the particular situation.

An example of a situation table is given in table 2.1. The situation table describes the simple control system introduced in figure 1.1 on page 1. The table presents the parameters of three periodic tasks.

Task name	Period	Phase	Exec. time	Deadline	Resources usage
Sensor	5.0	0.0	1.50	2.0	R_1
Control	5.0	2.0	1.75	2.0	R_1, R_2
Actuator	5.0	4.0	0.50	1.0	R_2

Table 2.1: *A situation table describing the timing parameters of a real-time system.*

In chapter 9 we introduce notation for describing the functional aspects of a real-time system.

Symbol	Legend
$\mathbf{T}$	A set of tasks
T_i	The ith task of a given set of tasks
$J_{i,k}$	The kth job in task T_i
J_k	The kth job of a task. The task index is omitted, as we are not interested in identifying a specific task
p_i	The period of the periodic task T_i is the length of the time interval between the release times for two consecutive jobs in task T_i
$r_{i,k}$	The release time of the kth job in task T_i
r_k	The release time of the kth job of a task. The task index is omitted, as are not interested in identifying a specific task
ϕ_i	The phase of the task T_i, where $\phi_i = r_{i,1}$
e_i	The execution time of task T_i
b_i	The worst-case blocking delay imposed on a job in task T_i due to resource contention
W_i	The worst-case response time of task T_i
D_i	The relative deadline of the task T_i
d_i	The absolute deadline of the job J_i with the release time r_i is $d_i = r_i + D_i$
π_i	The priority of task T_i
P_i	A processor. In this thesis we only consider real-time systems with a single processor. A job needs the processor to execute
R_i	A resource. Tasks may share resources
u_i	The utilisation of a task T_i is the fraction of time a strictly periodic task with execution time e_i and period p_i occupies a processor P_i
U	The total utilisation of the tasks in a system consisting of n tasks is the sum of the utilisations of the individual tasks

Table 2.2: *The notation used in the reference model.*

3 Priority-Driven Scheduling

In this chapter we introduce the priority-driven approach for scheduling hard real-time systems on a single processor. A taxonomy of hard real-time scheduling algorithms is given in figure 3.1.

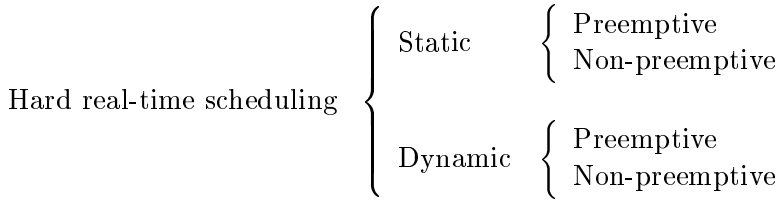

Figure 3.1: *The taxonomy of hard real-time scheduling algorithms.*

A scheduler is said to be *static* when it uses single schedule that has been calculated *off-line*, that is before the system is started. The interleaving of all jobs are predetermined, as all scheduling decisions have been made before system start. The schedule is repeated over and over until the system is restarted.

The concept of preemption was introduced in section 2.7. A scheduler may be either *preemptive* or *non-preemptive*. Preemptive means that the execution of any job can be interrupted by the execution of a higher priority job. In this thesis only preemptive schedulers are considered.

A *cyclic executive* is the implementation of a system, which is scheduled by a static scheduler. The implementation may be organised as a main loop calling a number of procedures. This architecture requires that a *harmonic* relationship exist between all periodic tasks in the system. If preemption is allowed, the static schedule is resumed at the end of a preemption instance. We will not consider static schedulers and their implementations any further in this thesis.

The priority-driven approach is an example of *dynamic* scheduling. The approach allows dynamic scheduling decisions at run-time contrary to the static scheduling approach. Dynamic scheduling is also referred to as *on-line* scheduling. When preemption is allowed a scheduling decision is performed whenever a job is released or completed.

Priority-driven scheduling algorithms are distinguished from each other in the way priorities are assigned to jobs. A *fixed-priority* scheduling algorithm assigns the same precomputed priority to all jobs in a task, whereas a *dynamic-priority* algorithm assigns different priorities to each job in a task.

In dynamic scheduling our main concern is the coupling of three components:

- An *optimal scheduling algorithm.*
- A *feasibility test*, which can decide whether a schedule produced by a scheduling algorithm is indeed feasible.
- A *worst-case response time computation* for a job, which depends on the specific scheduling algorithm.

In this chapter we investigate different priority-driven schedulers. We will consider an optimal dynamic-priority scheduler, and three optimal fixed-priority schedulers.

Feasibility tests and worst-case response time computations are very closely related. These topics are investigated in chapter 4.

3.1 The Simplified Periodic Task Model

In this chapter we restrict ourselves to scheduling systems characterised by a simplified version of the periodic task model introduced in section 2.5.

The *periodic task model* is a well known deterministic work-load model proposed in 1973 by Liu and Layland [Liu and Layland, 1973] [Liu, 2000]. The model imposes the following restrictions on a task set:

- All tasks are *periodic*, with a constant interval p_i between the instants the task is submitted for execution, i.e. released r_i.
- All tasks have a fixed *relative deadline* D_i and *phase* ϕ_i.
- All tasks are *independent*, in the sense that the release of a task does not depend on the initiation or completion of other tasks.
- All tasks have a fixed *execution time* e_i. All computational overhead is assumed to be included in the computation time.
- No task may voluntarily *suspend* itself.

In the periodic task model defined above no restrictions have been placed on the phases of the tasks in a task set. Similarly, no relations have been assumed between the period of a task and its relative deadline. In the following presentation of specific priority-driven schedulers, restrictions are imposed on phases and relative deadlines.

In chapter 5.6 we will extend the periodic task model to include shared resources.

3.2 Dynamic-Priority Scheduling

An example of a dynamic-priority scheduler is the *earliest deadline first (EDF)* scheduling algorithm.

3.2.1 Earliest Deadline First Scheduling

The EDF algorithm dynamically assigns priorities to jobs based on their absolute deadlines. Thus, the dynamic scheduling decision of the EDF algorithm may be defined as follows: at any time the eligible job with the earliest absolute deadline is scheduled. If there are no eligible jobs, the processor is idle.

Because of the optimality of the EDF algorithm, stated by theorem 3.1, it is considered theoretically superior to the class of fixed-priority schedulers.

Theorem 3.1 *When preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule for a task set* **T** *with arbitrary release times and deadlines on a single processor if and only if* **T** *has feasible schedules.*

A proof for the theorem is given in [Liu, 2000]. The proof is based on the fact that any feasible schedule of a task set **T** can be systematically transformed into a schedule produced by the EDF algorithm.

Despite its optimality, only few available real-time operating systems support the EDF scheduling algorithm. Fixed-priority schedulers were devised for easy and efficient implementations. Compared to an implementation for a fixed-priority scheduler, an implementation of an EDF scheduler will have a large overhead, due to the deadline management for the jobs that have been released but not yet completed.

The theory of dynamic-priority schedulers is well developed. Thus, feasibility conditions and worst-case response time computations have been developed. We shall not consider EDF anymore in this thesis.

3.3 Fixed-Priority Scheduling

In the following discussion of fixed-priority schedulers, we divide a fixed-priority scheduler into two components:

- The algorithm producing the precomputed priority assignment used in fixed-priority scheduling.
- The dynamic scheduling decision.

The fixed-priority schedulers considered in this section all use the highest priority first (HPF) dynamic scheduling decision. The HPF scheduling decision uses a precomputed priority assignment. Thus, the dynamic scheduling decision is defined as follows: at any time the eligible job with the highest priority is scheduled. If there are no eligible jobs, the processor is idle. The scheduling decision is performed when a job is released or completes.

In this section three different algorithms for computing priority assignments are discussed:

- The rate monotonic algorithm

- The deadline monotonic algorithm
- Audsley's algorithm

For each combination of a priority assignment algorithm and the HPF dynamic scheduling decision we will discuss the optimality of the resulting scheduler. For fixed-priority scheduling we will discuss optimality in the sense that if a task set can be scheduled by any fixed-priority scheduler, it can also be scheduled by the particular scheduler.

For the rate monotonic and deadline monotonic algorithms to be optimal it is required, in addition to the requirements to task sets given by the periodic task model in section 3.1, that all tasks in the set have the same phase ϕ. Hence, the first job in every task has the same release time r.

When tasks are allowed to have arbitrary phases a common release time between jobs in all tasks in a task set may not exist. If it does not exist, the rate monotonic and deadline monotonic algorithms are no longer optimal. Audsley's algorithm removes the requirement for identical phases. Thus, the algorithm is optimal for arbitrary phases. In chapter 4 we present an algorithm for determining if a common release time exists for a given task set.

3.3.1 Rate Monotonic Algorithm

The *rate monotonic (RM)* is a well known priority assignment algorithm. The algorithm assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. Thus, the task with the highest *rate*, i.e. the shortest period, is assigned the highest priority.

For a task set $\mathbf{T}$ where priorities are assigned using the rate monotonic approach, the relationship between periods and priorities is expressed by the following formula:

$$\forall T_i, T_j \in \mathbf{T} \cdot p_i > p_j \Leftrightarrow \pi_i \prec \pi_j \tag{3.1}$$

If several tasks have the same rate given by a common period p an arbitrary priority assignment among those tasks is selected.

The RM priority assignment corresponds to a simple ordering of tasks with respect to periods. Hence, the complexity of the RM priority assignment algorithm is $O(n \log_2 n)$ in the cardinality n of the task set.

For a set of tasks $\mathbf{T}$ which is in phase with relative deadlines equal to periods $D_i = p_i$ RM is an optimal priority assignment [Liu and Layland, 1973].

Theorem 3.2 *When preemption is allowed and jobs do not contend for resources, the RM algorithm can produce a feasible schedule for a set of tasks* $\mathbf{T}$*, when the tasks are in phase and the relative deadline of each task equals its deadline, if and only if* $\mathbf{T}$ *has a feasible schedule.*

3.3.2 Deadline Monotonic Algorithm

Another fixed-priority algorithm is the *deadline monotonic (DM)*. This algorithm assigns priorities to tasks according to their relative deadlines: the shorter the relative deadline the higher the priority. Hence, the task with the shortest relative deadline is assigned the highest priority.

For a task set $\mathbf{T}$ where priorities are assigned using the deadline monotonic approach, the relationship between periods and priorities is expressed by the following formula:

$$\forall T_i, T_j \in \mathbf{T} \cdot D_i > D_j \Leftrightarrow \pi_i \prec \pi_j \tag{3.2}$$

If several tasks have the same relative deadline D, an arbitrary priority assignment among those tasks is selected.

The DM priority assignment corresponds to a simple ordering of the tasks with respect to their deadlines. Thus, the complexity of the DM priority assignment algorithm is $O(n \log_2 n)$ in the cardinality n of the task set.

The DM algorithm relaxes the restriction of the RM algorithm that the deadline of each task in a task set must equal its period. The relaxation of this restriction is essential for jitter control as discussed in section 2.3. For task sets with relative deadlines less than or equal to periods, $D_i \leq p_i$, DM is an optimal priority ordering [Leung and Whitehead, 1982] [Audsley et al., 1991] [Audsley et al., 1993] [Liu, 2000].

Theorem 3.3 *When preemption is allowed and jobs do not contend for resources, the DM algorithm can produce a feasible schedule for a task set* $\mathbf{T}$ *when the tasks are in phase and relative deadline of each task is less than or equal to its period, if and only if* $\mathbf{T}$ *has feasible schedules.*

When the relative deadlines of every task are proportional to its period $D_i = \delta p_i$ for some constant δ, the RM and DM are identical. Thus, a corollary to the optimality of DM is that RM is optimal when the relative deadlines of every task is proportional to its period.

When the relative deadlines are arbitrary, the DM algorithm performs better than the RM in the sense that it can sometimes produce a feasible schedule when RM fails to do so. In the case of arbitrary relative deadlines the RM algorithm always fails when the DM algorithm fails [Liu, 2000].

3.3.3 Audsley's Algorithm

The RM and DM algorithms imposes the restriction on the periodic task model that all tasks share a common release time, i.e. all tasks have the same phase. This assumption simplifies response time analysis, but it is seldom a requirement that jobs must be released simultaneously. By assigning different phases to a set of tasks their work is spread out. In this case there may not be a common release time for the tasks, and RM and DM algorithms are no longer optimal.

Audsley's algorithm [Audsley, 1991] is an optimal algorithm for assigning priorities to task sets when phases may be arbitrary and deadlines may be less than or equal

to the periods. Thus, the algorithm removes the restriction imposed by the RM and DM algorithms.

Unlike the RM and DM algorithms, Audsley's priority assignment algorithm requires the availability of a feasibility test, which is applicable to a task set, where phases may be arbitrary and deadlines are less than or equal to periods. The test must be able to determine whether or not a priority assignment results in a schedule that is feasible. Such a test is known as a *necessary and sufficient* feasibility test. A priority assignment resulting in a feasible test is said to be feasible. We investigate feasibility tests in chapter 4, and discuss such a feasibility test in section 4.4.

In this section we consider a task set $\mathbf{T}$ of cardinality n, where phases may be arbitrary and deadlines are less than or equal to periods. When assigning unique priorities to the tasks in $\mathbf{T}$, there are $n!$ distinct priority assignments. A naive but optimal priority assignment algorithm would test the feasibility of all distinct priority assignments, and if feasible priority assignments were found arbitrarily select one. Audsley's algorithm improves this naive but optimal priority assignment algorithm by reducing the number of priority assignments that must be tested for feasibility.

In the periodic task model a task T_i in $\mathbf{T}$ is *feasible* if and only if

$$e_i + i_i \leq D_i,$$

where i_i represents the *inference* for T_i, i.e. the requirement for execution time of higher priority tasks in the interval defined by the relative deadline D_i of the task T_i.

If T_i is not feasible, and it is not possible to reduce the execution time e_i or increase the relative deadline D_i, the only way to make T_i feasible is to decrease i_i. This may be done for the given priority assignment by switching priorities between a higher priority task T_j and T_i. It may also be possible to reduce i_i by switching the priority of two tasks T_h and T_j where

$$\pi_h \prec \pi_i \prec \pi_j.$$

Approaching the problem in a more structured manner, we consider the task T, which is the task assigned the lowest priority n in a task set of cardinality n. Audsley proves the following two results about such a priority assignment [Audsley, 1991].

Theorem 3.4 *If T is assigned the lowest priority, n, and is infeasible, no priority assignment that assigns T priority level n is a feasible assignment.*

Theorem 3.5 *If T is assigned the lowest priority, n, and is feasible, then if a feasible priority assignment for $\mathbf{T}$ exists, a feasible assignment with T assigned the lowest priority exists.*

The theorems follow from the fact that the time-demand of all higher-priority tasks are constant for all priority assignments where T is assigned priority level n. The two theorems above are now generalised into considering an arbitrary priority assignment Ψ for a task set of cardinality n.

Theorem 3.6 *Let the tasks assigned priority levels $i, i+1, \ldots, n$ be feasible under that priority assignment, Ψ. If a feasible priority assignment for $\mathbf{T}$ exists, then there exists a feasible priority assignment that assigns the same tasks to levels $i, \ldots, n$ as Ψ.*

Theorem 3.6 is proved by induction in [Audsley, 1991]. It is shown that a feasible priority assignment can be transformed into a feasible priority assignment, where the tasks at level $i, i+1, \ldots, n$ are the tasks at the same levels in Ψ.

The above theorems form the basis for the optimal priority assignment algorithm, which assigns priorities $n, n-1, \ldots, 1$ in order. The priority assignment algorithm only proceeds to level $i-1$ if a feasible priority assignment can be made at priority level i. The algorithm is summarised in the steps below.

The algorithm is structured into two nested loops:

- The outermost loop iterates through the priority levels starting at the lowest priority n.
  - The innermost loop iterates through the subset of tasks, which have not yet been assigned priorities, searching for a task that is feasible at that priority.
    * If no feasible task is identified no priority assignment exists for the task set by theorem 3.4, and the algorithm terminates.
    * If a feasible task is found, the task is assigned the current priority. Theorem 3.6 ensures that if a feasible assignment exists one will exist with the feasible task assigned the current priority. The innermost loop exits.

If for a priority level several tasks are feasible, a feasible task is arbitrarily selected by Theorem 3.6. The algorithm is optimal by the theorems 3.4, 3.5 and 3.6.

The complexity of Audsleys's algorithm can be expressed in the number of tasks for which the algorithm performs a feasibility test, that is the number of different priority assignments considered. Assuming the cardinality of the task set is n, the number of feasibility tests performed when searching for a feasible task at priority level $i \leq n$ is i. Considering all the n priority levels the maximum number of feasibility tests can be expressed as n terms:

$$n + (n-1) + (n-2) + \ldots + (n-(n-1)) = \sum_{i=1}^{n} i$$

The sum of integers between 1 and n is an Archimedes series, which has the value

$$\sum_{i=1}^{n} i = \frac{1}{2} n(n+1)$$

giving us the maximum number feasibility tests.

Thus, the complexity of Audsley's priority assignment algorithm is $O(n^2 + n)$ in the number of tasks for which the algorithm performs a feasibility test. This is clearly better than performing a feasibility test for each of the $n!$ possible priority assignments.

4 Fixed-Priority Scheduling of Periodic Tasks

Dynamic scheduling was introduced in the previous chapter, including dynamic-priority as well as fixed-priority scheduling. We stated, that a general concern in dynamic scheduling is the coupling of three components:

- An *optimal scheduling algorithm.*
- A *feasibility test*, which can decide whether a schedule produced by a scheduling algorithm is indeed feasible.
- A *worst-case response time computation* for a job, which depends on the specific scheduling algorithm.

In this chapter we focus on fixed-priority scheduling. We addressed optimal scheduling algorithms for fixed-priority scheduling in section 3.3, where three different algorithms were presented. In this chapter we shall address the remaining two components.

Thus, we will consider the problem of determining if a schedule given by a precomputed priority assignment is indeed a feasible schedule. We will present techniques, based on the computation of the worst-case response times, which test if a task in a set of fixed-priority periodic tasks will meet its deadlines.

4.1 The Validation Problem

In this chapter we consider the *validation problem.* In its general form the problem may be stated as follows: Given a set of tasks, a set of resources available to the tasks, the scheduling algorithm allocating processors to tasks, and synchronisation protocols allocating resources to tasks, determine whether all jobs in the tasks will meet their deadlines [Liu, 2000].

The techniques for solving the validation problem, which is presented in this chapter, are only applicable to a restricted problem: Given a set of independent periodic tasks, and a preemptive fixed-priority scheduler, determine if all jobs in the tasks will meet their deadlines.

In chapter 5.6 we will consider a more general form of the validation problem, which includes shared resources and their synchronisation protocols. Hence, we may restate the validation problem: Given a set of periodic tasks, a set of shared resources, a fixed-priority scheduling algorithm allocating processors to tasks, and synchronisation protocols allocating resources to tasks, determine whether all jobs in the tasks will meet their deadlines.

4.1.1 Validating Timing Constraints in Priority-Driven Systems

A feasibility test is said to be *correct* if it never declares that all deadlines are met, when some deadlines are in fact missed. A feasibility test is based on a workload model. When the algorithm is applied to a system, the conclusion of the algorithm is only correct if the assumptions of the workload model is observed by the system.

Feasibility tests are divided into two classes:

Sufficient feasibility tests can determine if a task set is feasible. However, if a sufficient feasibility test cannot determine that a feasible schedule exists for the task set, a feasible schedule may still exist.

Necessary and sufficient feasibility tests can determine whether or not a feasible schedule exists for a given task set. Thus, if a necessary and sufficient algorithm determines that a set of tasks has a feasible schedule then all jobs in the tasks will meet their deadlines. If the algorithm determines that the task set is infeasible, then no feasible schedule exist for the task set.

While sufficient feasibility tests cannot dismiss a task set as infeasible, they are still useful tools in the design of hard real-time systems. In general, the complexity of sufficient tests are lower than that of necessary and sufficient tests. Thus, sufficient tests can often be performed by hand whereas the complexity of necessary and sufficient tests require the support of tools.

4.2 A Feasibility Test for Fixed-Priority Tasks with Short Response Times

The response time of a task is said to be *short*, if it is less than or equal to the period of the task. This section presents a feasibility test that can be applied to a task in a set of independent fixed-priority tasks, where the deadline of each task is less than or equal to its period. It is a requirement that priorities are unique, but it is not required that priorities are assigned by the rate monotonic or deadline monotonic algorithms.

The feasibility test for a task T_i is performed on the basis of information about periods p_i and execution times e_i of the tasks in a task set $\mathbf{T}$.

4.2.1 Critical Instants

To determine the worst-case response time of any job $J_{i,j}$ in task T_i in the task set $\mathbf{T}$, the worst-case combination of release times for $J_{i,j}$ and all higher priority jobs in $\mathbf{T}$ must be found. For this combination of release times the job $J_{i,j}$ faces the most contention for the processor.

The worst-case combination of release times for a task T_i is denoted a *critical instant* of T_i. The response time of a job in T_i released at a critical instant is the *worst-case response time* of the task and is denoted by W_i. The following theorem identifies the worst-case combination of release times.

Theorem 4.1 *In a task set* $\mathbf{T}$ *of fixed-priority tasks, where every task has a deadline which is less than or equal to its period, a critical instant of any task* T_i *in* $\mathbf{T}$ *occurs, when one of its jobs* $J_{i,j}$ *is released at the same time as a job in every higher priority task, hence,* $r_{i,j} = r_{k,l_k}$ *for some* l_k *for every* $k = 1, 2, \ldots, i-1$.

The notion of a critical instant was first introduced by [Liu and Layland, 1973] in the presentation of the RM scheduling algorithm. A proof of theorem 4.1 is given in [Liu, 2000].

Feasibility Intervals

Feasibility testing requires the definition of an interval over which the testing shall occur. This interval is denoted the *feasibility interval*.

For a system given by a set of n periodic tasks, the least common multiple of p_i for $i = 1, 2, \ldots, n$ defines the *hyper-period*, H, of the system. Thus, the execution of a the system of periodic tasks is periodic with a period equal to H. In general, when testing the feasibility of a set of periodic tasks that shares a critical instant, it is not necessary to consider a feasibility interval of length H.

By theorem 4.1, the feasibility of a task set of cardinality n sharing a critical instant t_0, can be determined by examining whether the jobs released at the critical instant have completed by their deadline. Thus, the feasibility interval may be restricted to the interval starting at the critical instant and ending at the absolute deadline for the lowest priority job $J_{n,i}$ released at the critical instant, $[t_0, t_0 + D_n]$.

4.2.2 Drawing a Time Line Diagram

A simple way of testing the feasibility of a task set $\mathbf{T}$ is to draw a time line digram over the feasibility interval of the task set. Time line diagrams were introduced in section 2.3. When the time line diagram is complete, it is inspected to see if all jobs in the feasibility interval meet their deadlines. If all jobs meet their deadlines the task set is feasible.

This feasibility test is necessary and sufficient. However, for large task sets this manual technique is laborious and error prone. For simple situations, the technique is appropriate for providing insight into the schedulability of the system.

We now consider a task set $\mathbf{T}$ consisting of four tasks. All tasks have a deadline equal to its period, and priorities have been assigned by the rate monotonic algorithm. All tasks share a critical instant as they share the same phase. The characteristics of the tasks is summarised in situation table 4.1.

We draw the time line diagram by drawing a time line for each task in the task set in priority order, starting with the highest priority. The time line for the four tasks $T_1, \ldots, T_4$ is presented in figure 4.1 on the next page. After the time line diagram is complete we inspect it to see if any job misses its deadline. In figure 4.1 on the following page we see that the lowest priority task T_4 misses its deadline at time 9.0. Hence, the task set is not feasible. As we only consider the feasibility interval the preemption of task T_4 at time 9.0 is left out in the time line diagram.

The schedulability tests in the following sections are the mathematical counterparts

Task name	Period	Phase	Exec. time	Deadline	Priority
T_1	3.00	0.0	1.00	3.00	1
T_2	5.00	0.0	1.50	5.00	2
T_3	7.00	0.0	1.25	7.00	3
T_4	9.00	0.0	0.75	9.00	4

Table 4.1: *A task set sharing a critical instant.*

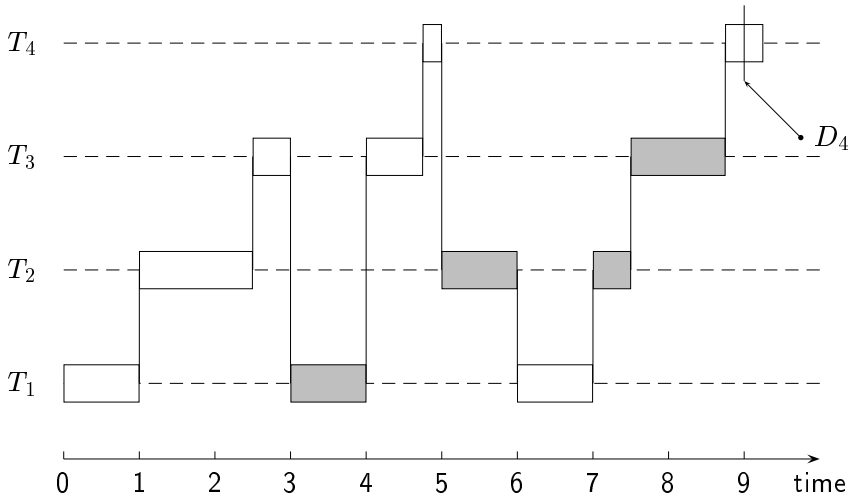

Figure 4.1: *Testing the feasibility of the task set summarised in situation table 4.1 by drawing a time line diagram. The task set is infeasible as task T_4 misses its deadline at time 9.0.*

to this technique.

4.2.3 Time-Demand Analysis

Because of theorem 4.1, when tasks are in phase a worst-case response time of a job in a task may be calculated. If the job $J_{i,j}$ in a task T_i is released at a critical instant at time t_0, then at time $t_0 + t$, where $0 \leq t \leq p_i$, the time-demand of the job $J_{i,j}$ and the jobs of higher priority tasks released in the interval $[t_0, t]$ is expressed by the *time-demand function*

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \text{for } 0 \leq t \leq p_i. \tag{4.1}$$

The second term of the time-demand function defines the interference on $J_{i,j}$, i.e. the time-demand of the jobs released in the $i-1$ higher priority tasks in the interval $[t_0, t_o + t]$.

The job $J_{i,j}$ will meet its deadline at $t_0 + D_i$ if and only if $w_i(t) \leq D_i$ for some $0 \leq t \leq D_i$. By theorem 4.1, the job $J_{i,j}$ released at a critical instant has the worst-case response time of all jobs in T_i. Hence, if the job meets its deadline, then all jobs in the task will meet their deadlines, and the task T_i is feasible. If the job does not complete within its deadline the task T_i is infeasible and the task set cannot be scheduled by the particular fixed-priority scheduling algorithm.

To investigate the nature of the time-demand function we return to the task set summarised in situation table 4.1 on the preceding page. All tasks have a deadline equal to its period, hence priorities were assigned using the rate monotonic algorithm. Additionally, all tasks share a critical instant as they share the same phase. In figure 4.2 on the following page the solid lines show the time-demand functions of the individual tasks. The dotted lines show the total contribution of higher-priority tasks to each lower priority task.

The time available for scheduling tasks is given by the supply function $y(t) = t$. Thus, when $w_i(t) > t$ the time-demand is greater than the time supply, hence the task is not feasible by the particular scheduling algorithm. When $w_i(t) \leq t$ the task can be feasibly scheduled as the time-demand is less than or equal to the time supply.

The worst-case response time of a task T_i is the least solution to the equation $w_i(t) = t$. Thus, a dot at the intersection between $w_i(t)$ and $y(t)$ marks the instant where the job $J_{i,j}$ released at the critical instant of T_i completes.

From figure 4.2 we see that the tasks T_1, T_2, and T_3 are schedulable. However, the time-demand function for task T_4 lies entirely above the supply function from 0 to 9, hence it misses its deadline at time 9. The figure illustrates the staircase nature of the time-demand function. An increase in the function occurs when a job in one of the higher priority tasks is released. Releases occur at integer multiples of the periods of higher priority tasks. After a release the time-demand is constant until the next release. Hence, at the right-most point on each plateau the shortage of processor time $w_i(t) - t$ is the smallest, and it may be negative if the supply is greater than the demand.

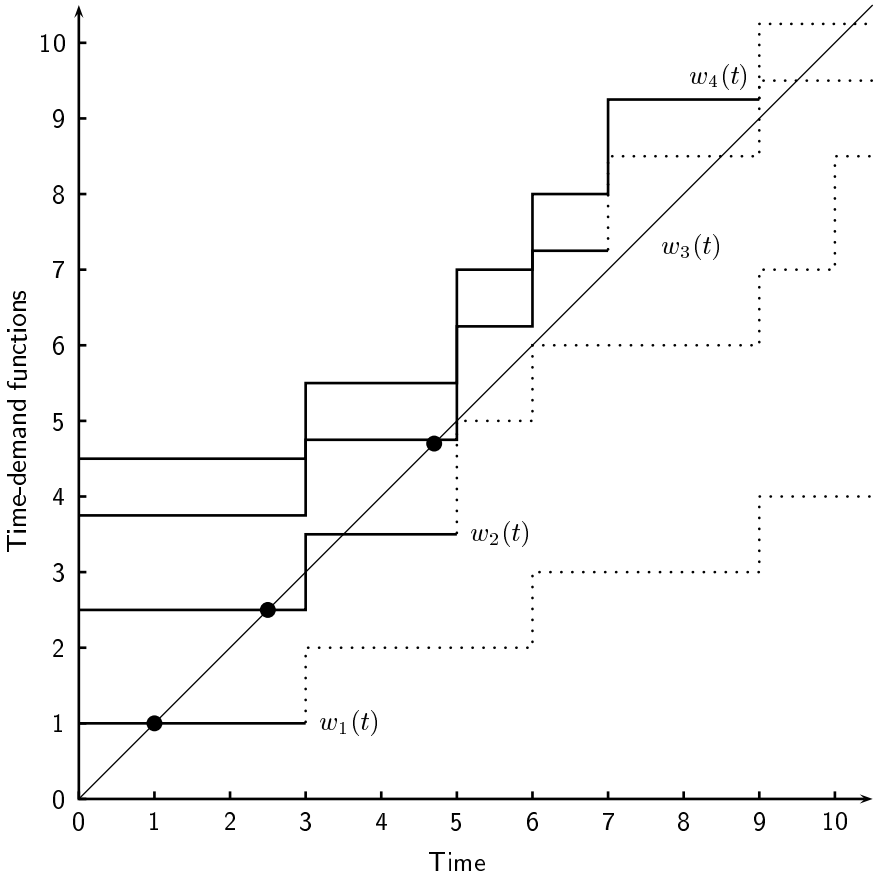

Figure 4.2: *Feasibility test of the task set summarised in situation table 4.1 by time-demand analysis. The task set is infeasible as task T_4 misses its deadline at time 9.0.*

Hence, to test if a task T_i in a task set $\mathbf{T}$ is feasible the *time-demand analysis method* requires us to:

1. Compute the time-demand function $w_i(t)$ according to equation (4.1)
2. Test if the inequality
$$w_i(t) \leq t \tag{4.2}$$
is satisfied for a release time t in one of the k highest priority tasks in $\mathbf{T}$, thus
$$t \in \{t' \mid t' = j_k p_k \wedge k = 1, 2, \ldots, i\}$$
where for each of the tasks T_k we must consider the following releases of a job in the task
$$j_k = 1, 2, \ldots, \left\lfloor \frac{\min(p_i, D_i)}{p_k} \right\rfloor$$
If the equality is satisfied for any of these instants the task T_i is feasible.

   The worst-case response time of the task is the value of the time-demand function for the least solution to the inequality 4.2.

Given the time-demand analysis method, the feasibility of a task set may be tested by successive application of the method to each task in the set from the highest priority task to the lowest priority task. The task set is feasible if and only if all the individual tasks are feasible.

We now return to the task set summarised in situation table 4.1 on page 26 to illustrate the application of the time-demand analysis method. We apply the method to the tasks in decreasing priority order. For task T_1 we must test the inequality $w_1(t) \leq t$ for the single value of 3. We find that the inequality is satisfied, hence the task is feasible. For T_2 we must test the values 3 and 5. The inequality is satisfied for the value 3. For the task T_3 the inequality must be tested for the values 3, 5, 6, and 7. We find that the inequality is not satisfied by 3 but is satisfied by 5. For the task T_4 we must test the following values 3, 5, 6, 7, and 9. The inequality $w_4(t) \leq t$ is not satisfied by any of the values, hence the task T_4 is infeasible. The time-demand functions of the individual tasks were illustrated in figure 4.2 on the facing page.

We may use the time-demand analysis method with task sets, where the tasks have arbitrary phases. In this case a critical instant may not exist, hence the worst case situation might not occur. For such task sets we simply ignore the phases. Thus, we assume the worst-case behaviour of the system when using the time-demand analysis method.

By ignoring phases for a task set in which the tasks will never share a critical instant, we reduce the necessary and sufficient feasibility test to a sufficient test. Hence, if a task set is deemed infeasible it may still be feasible, as the worst-case situation might not occur. However, if a task determined to be s feasible, it is indeed feasible.

4.2.4 Worst-Case Response Time Computation

The previous section discussed how it was possible to test the feasibility of a task with a deadline less than or equal to its period. For a task the feasibility was tested

by computing the worst-case response time for integer multiples of the periods of higher priority tasks and the period of the task itself.

This section presents an alternative algorithm that computes the worst-case response time of a task T_i from a task set $\mathbf{T}$ in an iterative manner. It is a requirement of the algorithm that $\mathbf{T}$ scheduled by the RM scheduling algorithm.

To compute the worst-case response time we must find the least solution with respect to t in equation (4.3).

$$t = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k \tag{4.3}$$

Equation (4.3) may be solved in an iterative manner [Klein et al., 1993]. The following three steps defines an algorithm for computing the worst-case response time for a task T_i.

1. Compute the initial approximation to the worst-case response time W_i.

$$t^0 = \sum_{k=1}^{i} e_i$$

2. Use the approximation t^l to compute the next approximation t^{l+1} in the formula below

$$t^{l+1} = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t^l}{p_k} \right\rceil e_k$$

3. Determine if the approximation is the answer.
   - If t^{l+1} is less than or equal to D_i, and t^{l+1} is not equal to t^l, step 2 is repeated.
   - If t^{l+1} is greater than D_i the worst-case response time is greater than the relative deadline, hence the task is infeasible.
   - If $t^{l+1} = t^l$ the algorithm terminates and t^l is the worst-case response time of the task.

The algorithm may provide an erroneous result if the task set is not scheduled by the RM scheduling algorithm [Briand and Roy, 1999].

4.3 Feasibility Test for Fixed-Priority Tasks with Arbitrary Response Times

The response time of a task is *arbitrary* if it may be larger than the period of the task. In this section we sketch a feasibility test for determining the feasibility of a task with a deadline, which may be larger than the period. We refer to [Liu, 2000] for the details of the test.

As the response time is allowed to be larger than the period a task may have more than one job ready for execution at any time. It is a requirement that ready jobs in the same task is scheduled in the order they were released.

When jobs are allowed to execute beyond the end of its period, we must define a new feasibility interval over which we shall perform the feasibility test. A *level-π_i busy interval* is an interval during which the processor is assigned to jobs in tasks with priorities greater than or equal to π_i. At the end of the interval there are no ready jobs in tasks of priorities greater than or equal to π_i.

A level-π_i busy interval is said to be *in phase* if the first job of all tasks, that have priorities higher than or equal to π_i and are executed in the interval, have the same release time. When determining the schedulability of a task T_i in a task set $\mathbf{T}$, in which the response times of jobs can be larger than their respective periods, it is sufficient to consider the special case, where tasks are in phase.

As response times are arbitrary, the first job $J_{i,1}$ may not have the largest response time among all jobs in T_i. Thus, to test the feasibility of the task T_i, we must examine all jobs of T_i in the first level-π_i busy interval, which is in phase. The right hand side of the time-demand function (4.1) is still valid for the individual jobs in task T_i within a level-π_i busy interval.

Hence, to test the feasibility of a task set $\mathbf{T}$ where deadlines are arbitrary, we test the feasibility one task at a time from the highest priority task to the lowest priority task.

4.4 Feasibility Test for Fixed-Priority Tasks with Short Response Times and Arbitrary Phasing

When tasks are permitted to have arbitrary release phases, a common release time between the jobs of all tasks in the task set may not exist.

In [Audsley, 1991] Audsley presents a method for determining if a common release time exists. If a common release time exists we may use one of the feasibility tests discussed in the previous two sections.

If a common release time does not exist Audsley's feasibility test is a necessary and sufficient feasibility test [Audsley, 1991]. Due to the complexity of the Audsley's feasibility test we shall simply refer to the original paper for further details.

5 Synchronisation Protocols

In real-time systems resources may be shared between tasks making those tasks interdependent, i.e. we have a *resource contention* or a *race condition* between the tasks. To control this resource contention a *synchronisation protocol* or a *resource access-control protocol* must be used. In this thesis we shall only consider resources which shall be allocated in a mutual exclusive manner, i.e. a critical section.

Using a traditional *fixed priority* synchronisation protocol a task can make a *request* on a resource. If the resource is available it becomes *allocated* by the task. If the request fails, i.e. the resource is already allocated by another task, the task may abort the request or wait until the resource becomes available. A task *releases* a resource when it has finished using it.

The failure to request a resource may result in a task being blocked by another task of lower priority. This phenomenon is called *priority inversion*, i.e. the blocked task indirectly a gets lower priority than the blocking task. We define the *blocking*, b_i, for a task, T_i, as the worst possible amount of time the task can be blocked by a task of lower priority. Thus, the task of lowest priority can never be blocked.

Unfortunately, there are situations were a task can be blocked for uncontrolled and unacceptable times. Occasionally there may even be mutual deadlocks. Uncontrolled blockings and mutual deadlocks are not acceptable in hard real-time systems. In this chapter we present alternatives to the fixed priority synchronisation protocol. These protocols deal with the problems introduced in the next section.

5.1 Uncontrolled Blocking and Mutual Deadlock

Table 5.1 on the next page features a task set which results in uncontrolled blocking of the task T_1. The situation is illustrated in figure 5.1 on the following page. The problem is that the intervening and resource independent task T_2 might run for a very long time. The situation becomes even worse if more intervening tasks are added. The intervening tasks are said to be *chain blocking* or to be *transitive blocking* task T_1.

When designing real-time systems the amount of blocking should be at an absolute minimum. Even a good design cannot avoid blocking totally, but if chain blocking can be eliminated, the blocking of tasks becomes controllable and thus might be acceptable.

When tasks allocate more than one resource at the same time, the possibility for mutual deadlock may occur. Table 5.2 on the next page features a task set which results in mutual deadlock. The situation is illustrated in figure 5.2 on the following page.

Task	Priority	Phase	Resource usage
T_3	3	0	1.7
T_2	2	2	-
T_1	1	3	0.8

Table 5.1: *A task set which results in uncontrolled blocking.*

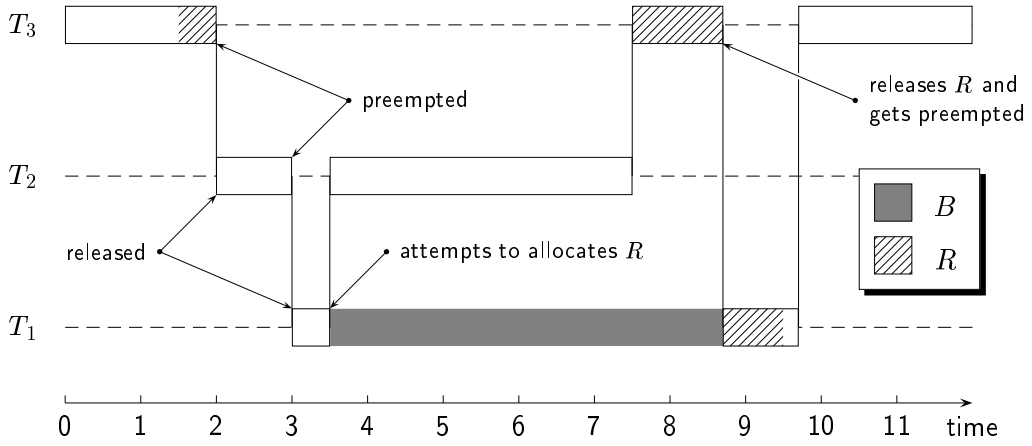

Figure 5.1: *Example of an uncontrolled blocking situation. The intervening task T_2 is chain blocking the task T_1 which causes the uncontrolled blocking.*

			Resource usage	
Task	Priority	Phase	R_1	R_2
T_2	2	0	2.0	1.5
T_1	1	2	0.5	3.0

Table 5.2: *A task set which results in deadlock.*

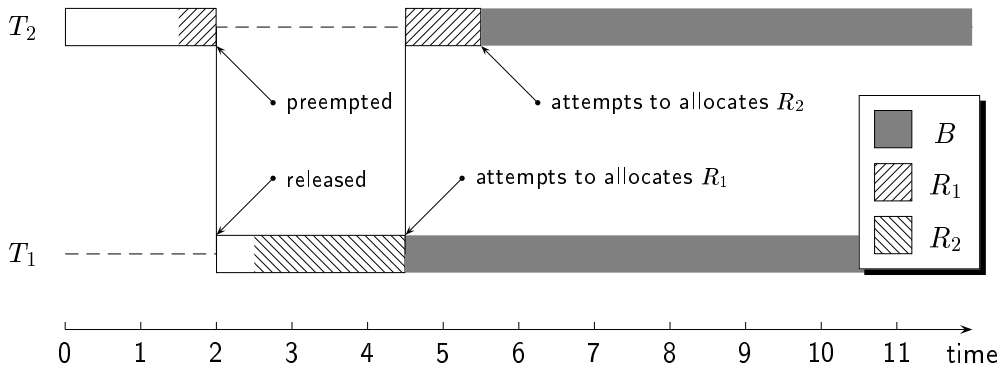

Figure 5.2: *Example of a mutual deadlock situation.*

A deadlock is fatal. A simple way of avoiding deadlocks is to forbid tasks to request more than one resource at the same time or to let all task request the resources in a predefined sequence, e.g. R_1, R_2, etc. Methods to detect deadlocks at the application design level are a subject of concurrent programming theory and is not treated in this thesis.

A good application design cannot in general eliminate the problems of chain blocking and mutual deadlock. Thus, we need a synchronisation protocol that can do that. A number of protocols can eliminate chain blocking and/or deadlock. The most known are listed below:

- priority inheritance protocol (PIP)
- priority ceiling protocol (PCP)
- non-preemptive critical section protocol (NPCS)
- highest locker protocol (HL)

The PIP can decrease the blocking duration but is not able to control blocking in general. Furthermore, it does not avoid deadlock. The PCP can both control blocking and avoid deadlock.

The PIP and PCP basically works using *priority inheritance*, e.g. if a task T_1 requests a resource allocated by a task T_3 of lower priority, T_3 dynamically inherits the priority of T_1 until is has released the resource. The PIP protocol and especially the PCP protocol are difficult to implement and introduces a large overhead. In this thesis we shall only consider the NPCS and the HL. For a more detailed description of the PIP and PCP protocols see [Klein et al., 1993], [Briand and Roy, 1999], and [Liu, 2000].

Section 5.2 and section 5.3 present the non-preemptive critical section protocol (NPCS) and the highest locker protocol (HL), respectively. In section 5.4 we see how to compute the amount of blocking for tasks using either the NPCS or HL. Furthermore, in section 5.5 we present an alternative to synchronisation protocols for periodic tasks were their timing behaviour are used to synchronise tasks.

5.2 The Non-Preemptive Critical Section Protocol

The *non-preemptive critical section protocol (NPCS)* makes a task non-preemptable when the task requests a resource. Thus, a task always succeeds in allocating a resource. No other task can preempt the task and make a request for the same resource while the resource is allocated. When the resource is released, the task is made preemptable again. The effect of the protocol is similar to giving the task the highest priority of all tasks.

The protocol is very easy to implement and may be implemented in two variants:

Task level Makes a task non-preemptable by disabling the scheduler, i.e. informs the scheduler that the task is now non-preemptable.

Interrupt level Makes a task non-preemptable by disabling interrupts.

The protocol prevents both chain blocking and deadlock, i.e. no other task can execute as long as the task is non-preemptable. The new behaviours of task sets presented in table 5.1 to 5.2 on page 34, using the NPCS, are illustrated in figure 5.3 to 5.4 on this page.

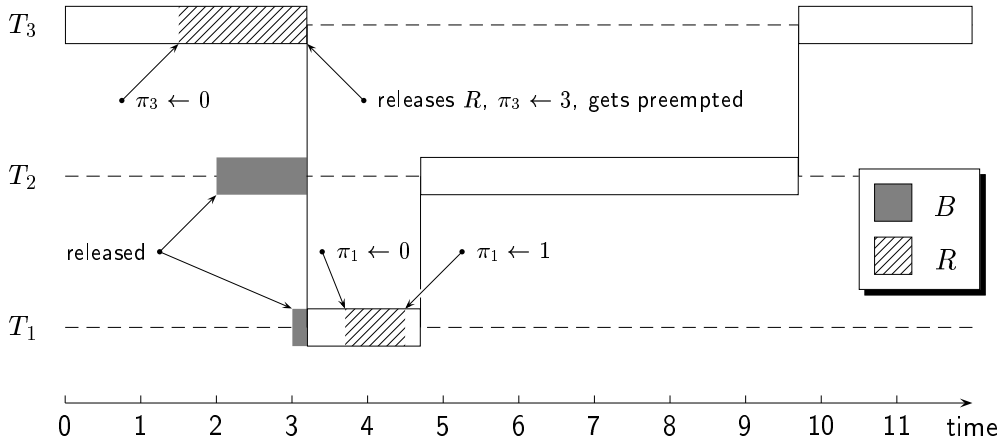

Figure 5.3: *Example of uncontrolled blocking avoidance using the non-preemptive critical section protocol or the highest locker protocol. The intervening task T_2 is now push-through blocked of T_3.*

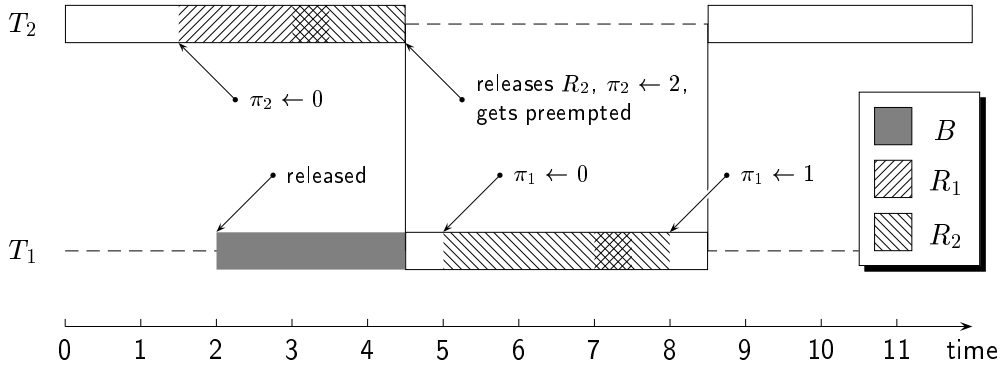

Figure 5.4: *Example of mutual deadlock avoidance using the non-preemptive critical section protocol or the highest locker protocol.*

The protocol fails to work

- on multiprocessor platforms where a task running in parallel can also request the same resource.
- if the non-preemptable task choose to self-suspend thereby yielding control to another task which can also request the same resource.

A task using the protocol will block *all* tasks of higher priority. Even tasks that never request the resource are blocked. The latter phenomenon is called *push-through blocking*, i.e. the tasks are blocked by an event they do not take part of. In the next section a synchronisation protocol which is more moderate in terms of push-through blocking is presented.

The NPCS is a good and simple protocol when the critical sections are small and when most of the tasks conflict with each other.

5.3 The Highest Locker Protocol

The *highest locker protocol (HL)* uses priority manipulation to synchronise tasks. It uses the following rules [Klein et al., 1993, p. 5-47]:

- Each shared resource has a *ceiling priority* defined, prior to run-time, as the highest priority of all the tasks that requests the resource.
- When a task requests a resource, its priority is set to one level higher than the defined resource ceiling priority. If a task is requesting more than one resource at the same time, its priority is set to one level higher than the highest ceiling priority of all the requested resources. Thus, a task always succeeds in allocating a resource.
- When a task has released all resources, its normal priority is restored.

No other task competing for the same resources can preempt the task which allocated the resources and thus the protocol prevents both chain blocking and deadlock.

If all resource ceiling priorities are equal to the priority of the highest priority task the behaviour of the HL is identical to the NPCS. Thus, the examples situated by table 5.1 to 5.2 on page 34, and illustrated in figure 5.3 to 5.4 on the facing page gives the same behaviour for NPCS and HL.

Consider the example situated in table 5.1 and illustrated in figure 5.3. If a task of higher priority were added to this set of tasks, the task would have been push-through blocked using the NPCS but not by using the HL. Thus, using the HL, tasks with higher priority than the ceiling priority of a resource cannot be push-through blocked by request on that resource.

Like the NPCS this protocol fails to work:

- on multiprocessor platforms where a task running in parallel can also request the same resource.
- if the non-preemptable task choose to self-suspend thereby yielding control to another task which can also request the same resource.

The protocol is easy to implement. The use of the protocol requires knowledge prior to compilation of all tasks that uses a resource in order to define the ceiling priority of each resource.

5.4 Computing Blocking Times

With the presented synchronisation protocols we have eliminated the possibility for chain blocking and mutual deadlock. Unfortunately, these protocols introduces

another blocking factor: push-through blocking. Fortunately, push-through blocking is more controllable than chain blocking.

We repeat the definitions of direct blocking and push-through blocking:

Direct blocking A high priority job can be blocked by a low priority job in a direct resource contention.

Push-through blocking A medium priority job can be blocked by a low priority job which inherits the priority of a high priority job.

For the HL a task is subject to push-through blocking if its priority is between the priorities of at least two task sharing a resource. The factor of push-through blocking is the duration of the resource allocation of the push-through blocking task. If a task is push-through blocked by several tasks the factor is the maximum duration of the resource allocations of the push-through blocking tasks. A task may be direct blocked, at most, for the duration of the longest resource allocation by a resource it uses.

Both protocols has the property that a job can be blocked for at most one duration of the longest continuous resource allocation made by tasks of lower priority.

Table 5.3 situates a task set of four tasks. The two tasks, T_2 and T_4, has a resource contention for the resource R. The table shows the obtained blockings with either of the two protocols. The situation presented is quite simple. Structured methods to determine blockings are needed in more complex situations. Examples of these can be found in [Klein et al., 1993], [Liu, 2000], and [Briand and Roy, 1999].

			Blocking	
Task	Priority	Resource R usage	NPCS	HL
T_1	1	-	$9^{\ddagger}$	0
T_2	3	9	$7^{\dagger}$	$7^{\dagger}$
T_3	4	-	$7^{\ddagger}$	$7^{\ddagger}$
T_4	5	7	-	-

Table 5.3: *Example of how synchronisation protocols affect the blocking. The blockings marked with a dagger are direct blockings, and blockings marked with a double dagger are push-through blockings.*

5.5 Synchronisation with Phasing and Deadlines

In this section we presents an alternative to the presented synchronisation protocols.By the use of timing constraints we can obtain mutual exclusive access of a resource. We illustrate this by an example. The alternative can be used for periodic tasks in certain situations only.

Consider the situation presented in table 5.4 on the facing page. Here two tasks with same period, T_1 and T_2, shares a resource R. By creating a controlled interleaving of tasks by introducing a phase displacement and deadline we ensure that the task

of higher priority cannot preempt the task of lower priority. Thus, the task cannot be in direct contention for the resource. The situation is illustrated in figure 5.5.

Task	Priority	Phase	Period	Exec. time	Rel. deadline	Resource usage
T_2	2	2.0	10.0	6.0	8.0	1.0
T_1	1	0.0	10.0	1.5	2.0	1.0

Table 5.4: *Situation table for alternative synchronisation protocol example.*

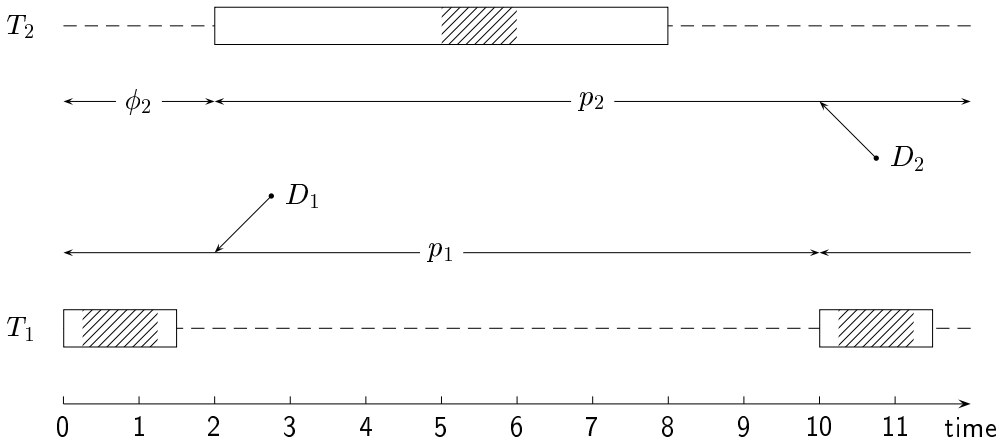

Figure 5.5: *Example of mutual exclusive access using precedence constraints.*

Using this alternative it is *very* important that the two tasks are released by the same timer. Otherwise, the periods of the tasks may drift from each other causing a direct resource contention with no mutual exclusion access.

5.6 Shared Resources in the Periodic Task Model

In the previous sections we investigated the extra complexity introduced by sharing resources in a real-time system. To ensure mutual exclusive access to a shared resource a job may be blocked. In particular, we explained how shared resources may cause scheduling abnormalities such as priority inversion, leading to uncontrolled blocking.

To control the blocking the concept of synchronisation protocols was introduced. We gave a detailed description of the two synchronisation protocols: NPCS and HL. NPCS ensures mutual exclusion access and controls the blocking by turning critical sections into non-preemptable sections. Though the synchronisation protocol succeeded in controlling the blocking, it introduced a new form of blocking, push-through blocking, which will affect all but the lowest priority task in the system. HL ensures mutual exclusion access and controls blocking by priority manipulations, which turns the critical section into a non-preemptable for task within a given range of priorities.

The delay due to blocking may cause a job in a high priority task to miss its deadline. Thus, when testing the feasibility of a task we must consider the interference of all

higher priority tasks, but also the amount of blocking introduced by lower priority tasks.

The time-demand function (4.1) in section 4.2.3 did not consider the amount of blocking introduced by lower priority tasks. With the introduction of synchronisation protocols, we argued that a job is only blocked if it is released when a lower priority task is in a non-preemptive critical section. Thus, we may see blocking as extra execution time added to the start of the job.

Fixed-Priority Tasks with Short Response Times

The feasibility test for fixed-priority tasks with short response times are now extended to properly consider blocking. Assume we have computed the blocking time b_i for the task T_i as described in section 5.4. If a job is to meet its deadline, the time-demand of all higher priority tasks, the task itself, and the blocking time imposed by lower priority tasks must be met by the time supply.

Thus, we may add the blocking time b_i to the time-demand function $w_i(t)$ for the task T_i. Hence, when using the following time-demand function

$$w_i(t) = e_i + b_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \text{for } 0 \leq t \leq p_i. \tag{5.1}$$

instead of the function (4.1) in the inequality (4.2) of the feasibility test, we properly consider the effects of blocking when testing the feasibility of a task T_i.

Fixed-Priority Tasks with Arbitrary Response Times

In the feasibility test for a task which has arbitrary response times T_i the effects of blocking is captured as in the feasibility test for tasks with short response times. We refer to [Liu, 2000] for further details.

6 Defining a Computational Model

In this chapter we define a *computational model* which will restrict the possible design of a hard real-time system. The restrictions ensure a schedulability analysis of a design is possible.

In the case study of this thesis we shall adhere to this computational model in order to analyse the resulting designs.

The restrictions are derived from the theory of the previous chapters and is stated below without any further explanation.

Processors A system consists of one and only one CPU.

Dynamic scheduling decision A fixed priority, preemptive scheduler is required. The dynamic scheduling decision shall work in a HPF manner.

Priorities The priorities of tasks must be fixed using either the DM approach or the Audsley approach. Thus, all tasks are assigned unique priorities.

Task behaviour All tasks must have a truly periodic behaviour. Tasks are not allowed to self-suspend and to alter their priorities themselves.

Execution times It must be possible to compute the worst-case execution time (WCET) for a task.

Deadlines A job in a task may have a deadline less or equal to its period.

Phases Tasks are allowed to have different phases.

Task synchronisation For ensuring mutual exclusive access of a critical section either the NPCS or the HL must be used.

In special situations mutual exclusive access of a critical section may be obtained using precedence constraints, cf. section 5.5.

Tasks shall only interact using critical sections. Resources to be requested in a mutual exclusive manner shall be treated as a critical section.

7 The Nucleus Plus Real-Time Operating System

Nucleus Plus (NP), by Accelerated Technology Incorporated (ATI), is a real-time, preemptive, multitasking kernel designed for time-critical embedded applications. Nearly 95% of Nucleus Plus (NP) is written in ANSI C, making it portable for different processor architectures and compilers.

NP is implemented as a C library. Applications are linked together with this library, resulting in one binary object which may be loaded onto a target platform. NP comes with complete source code, which promotes greater understanding and permits application-specific modifications.

This chapter provides an overview of the features in NP and also highlights relevant details for this thesis. Finally, we discuss pros and cons for using NP in hard real-time systems.

7.1 Provided Functionality

NP features a large set of functionality, e.g. tasks, dynamic memory, inter-process communication and semaphores, which is described in [ATI, 2000b]. This is summarised in this section.

Nucleus Plus Tasks

NP tasks are semi-independent programs with dedicated purposes. Managing the execution of competing tasks is the main purpose of NP, cf. section 7.3. A task is always in one of following five states:

Executing Task is currently running

Ready Task is ready to run, but another task is running

Suspended Task is dormant while waiting for a service request, cf. section 7.3. When the request is complete, the task is moved to the ready state

Terminated Task was killed

Finished Task finished its processing

If a task enters one of the two latter states, it has to be reset in order to execute again. If reset the task becomes suspended and must be explicitly resumed in order to execute again. Figure 7.1 on the following page shows an automaton of the states of a NP task. Processing time required for managing task is constant.

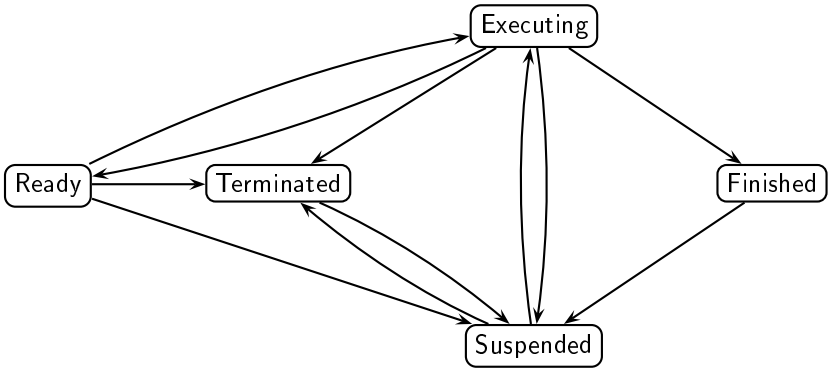

Figure 7.1: *Automaton for the states of a Nucleus Plus task.*

Partition Memory Pools

A partition memory pool specifies an area of memory from where it is possible to allocate/deallocate *fixed-sized* memory blocks. The block-size is specified when the partition memory pool is created. A property of partition memory pools is that a request for a block of memory can always be granted by a non-empty pool. In general, this property does not hold for dynamic memory pools due to fragmentation, cf. section 7.1. A task is subject to suspension if it attempts to allocate memory from a partition memory pool that is currently empty. Processing time required for allocating and deallocating memory partitions is constant.

Dynamic Memory Pools

A dynamic memory pool specifies an area of memory from where it is possible to allocate/deallocate *variable-sized* memory blocks. NP uses a first-fit algorithm, which is basically a linear search, to see if a memory block of the wanted size is available. A task is subject to suspension if it attempts to allocate a memory block of a size bigger than currently available. Processing time required when allocating memory from a dynamic memory pool is upper-bound by the total size of the dynamic memory pool and the number of fragments to look through.

Mailboxes

A mailbox provides a mechanism to simple asynchronous inter-process communication. It is capable of holding one message at the time only. A message has a fixed size of 16 bytes and is sent and received by value. A task is subject to suspension if it attempts

- to receive a message from an empty mailbox; or
- to send a message to a full mailbox.

It is also possible to broadcast a message to all the tasks waiting for a message. Processing time required for sending and receiving a single messages is constant.

Queues and Pipes

Queues and pipes are conceptually similar constructs which provides a mechanism to asynchronous inter-process communication. They are very similar to mailboxes,

except that they can hold several messages at the time. Furthermore, messages can be either fixed-sized or variable-sized of arbitrary length. Messages may be placed either at the front or the back of the queue/pipe. The only difference between queues and pipes is that

- a queue message consists of one or more 32-bit words; and
- a pipe message consists of one or more bytes.

It is also possible to broadcast a message to all the tasks waiting for a message. Basic processing time required for sending and receiving messages is constant. However, the time required to copy a message is linear in the size of the message. Thus, a broadcast operation is linear in the number of waiting tasks.

Semaphores

A semaphore provides a mechanism to control access of critical regions and to manage shared resources. The implementation provides counting semaphores that ranges in value from 0 to 4,294,967,294. A task is subject to suspension if it attempts to obtain a semaphore whose count is currently zero. Processing time required for obtaining and releasing a semaphore is constant.

Event Groups

Event groups provides a mechanism to indicate that a certain system event has occurred. An event is represented by a single bit, called an *event flag*. An event group consists of 32 event flags. Event flags are synchronous by nature. A task does not recognise that event flags are present until a specific request is made. A task is subject to suspension if it tries to receive a combination of event flags that is currently not present. Processing time required for receiving event flags is constant. However, the processing time required to set an event flag is linear in the number of tasks currently suspended on the event group to whom the event flag belongs.

Signals

Signals are somehow similar to event groups. However, there are significant differences in operation. As mentioned in section 7.1, event groups are synchronous by nature. Signals, on the other hand, operate in an asynchronous manner. When a signal is present, a special signal handling routine is executed when the task is resumed. A task is capable of handling 32 different signals. A task's signal handling routine must be supplied before any signal can be processed. Processing time required to send and receive signals is constant. The time required by the signal handling routine is application specific.

Timers

Most real-time applications require processing on periodic time intervals. Each task in NP has a built-in timer which is used to provide task sleeping and time-outs on system calls which might suspend the task. It is also possible to read and set the system clock.

NP provides programmable *application timers*. When they expire a specific user-

supplied routine executes. They can be set up in two ways: as an *one-shot* timer or as a *periodic* timer. Periodic timers continuous to expire until disabled. Application timers execute as HISRs, cf. section 7.1. Therefore, self-suspension are not allowed and processing should be kept at a minimum. Processing time required to manage application timers is constant. However, time required by the user-supplied routines depends on the routines themselves and the number of timers that expired simultaneously.

Interrupts

When an interrupt occurs, the processor suspends the current path of execution and transfers control to the appropriate interrupt service routine (ISR). In NP an interrupt service routine may be divided into two parts:

Low-level interrupt service routine (LISR) A LISR is a regular ISR. A LISR have very limited access to the system services of NP, e.g. only six simple system services are available. If a LISR needs to access other system services, this must be indirectly done by activating a *high-level interrupt service routine (HISR)*. It is not possible to release a semaphore from a LISR.

In NP there are two kinds of LISRs: *managed* and *unmanaged*. With the managed LISR NP takes care of saving and restoring the context. This might introduce some extra overhead because NP saves all registers whether they are to be altered or not. With the unmanaged LISR, the routine itself must take care of everything self. This makes it possible to make a tailored ISR with a minimum of overhead.

High-level interrupt service routine (HISR) A HISR can only be activated by a LISR and thus form the second part of a NP ISR. A HISR is scheduled by NP, cf. section 7.3 and may access most of the NP system services. However, *only* non-blocking system services may be called. Thus, if a HISR needs to call a system service which may block, this must be done indirectly by starting a regular NP task.

NP offers system services to disable and enable interrupts. By disabling all interrupts a task can not be preempted at all, thus ensuring mutual exclusion access of critical regions.

System Diagnostics

Nucleus Plus provides facilities to improve examinations of problems in the system:

Error management If a system error occurs, processing control is transferred to a common error handling routine, cf. section 7.2. By default, this routine prepares an error message and halts the system. However, it is possible to add additional error processing to this routine.

System history System activities can be logged into a circular log. Additionally, it is possible for tasks and HISRs to make entries in this log. Every log-entry is time-stamped and is provided with information about the service and the caller. In order to enable system history, the NP library must be compiled with this option enabled.

Version information It is possible to retrieve information about the underlaying version and release of the NP operating system.

License information It is possible to get information about the customer's license of NP, e.g. the customer's serial number.

I/O Drivers

Many applications require input and output from various peripherals. The control of peripherals is usually accomplished through a *device driver*.

NP provides a standard interface to attend request for initialisation, assignment, releasing, input, output, status and termination. It is possible to protect the internal data structures of a driver from simultaneous access. In this case a task may be subject to suspension.

7.2 States of Execution

A NP application is always in one of the following six possible states of execution:

Initialisation is the systems first state of execution. All NP components are initialised by this function. After system initialisation is complete, application specific initialisation is carried out. After all initialisation is complete, the system timer is started and the control is transferred to the scheduling loop.

System error is entered if a system error occurs. System errors are at most detected during initialisation. However, stack overflow conditions are detected during task and HISR execution. By default, system errors are fatal and therefore this is a terminal state.

Scheduling loop is responsible for transferring control to the HISR or task of highest priority, ready for execution. Control stays inside this loop if no HISR or task is ready to execute.

Tasks express most of the application processing. A task has it own stack and have full access to NP services.

Signal handlers are associated with tasks and executes on top (using the same stack) of a task. A signal handler have limited access to NP services, e.g. self-suspension is not allowed.

HISR forms the second part of an interrupt service routine. HISRs are scheduled in a manner similar to tasks, cf. section 7.3. Most non-blocking system calls may be performed by a HISR.

7.3 HISR and Task Management

NP has a preemptive scheduler that runs in a HPF manner. Tasks and HISRs are both managed by the scheduler. A ready HISR is always executed before any task,

i.e. HISRs have higher priorities than tasks. HISR priorities can be either 0, 1 or 2, where priority 0 is the highest. Task priorities ranges from 0 to 255, where priority 0 is the highest. The priority of a HISR or a task is specified at the creation time. The priority of a task may be changed dynamically.

Tasks at the same priority level may or may not run in a weighted round-robin (WRR) fashion. The size of the time-slice is specified in timer ticks at task creation time and can be changed dynamically. A task can also relinquish control to other tasks at the same priority level.

It is possible for a task to disable preemption and thus not become preempted of other tasks. This can be done at task creation time and can also be done dynamically. If preemption is disabled time-slicing is also disabled for tasks at the same priority level. If a task has disabled preemption a HISR can still preempt a task.

Suspension of tasks

When a task invokes a system call which could start a race condition for a shared resource, the task is subject to *implicit suspension*. In this case the task has the option to

- suspend unconditionally;
- suspend with a time-out; or
- not to suspend.

The type of suspension is specified when the task invokes such types of system calls. If the task chooses to suspend unconditionally, it will be resumed when the resource becomes available for that task. If it chooses to suspend with a time-out and the resource does not become available before the time-out, the task continues with other jobs when the time-out occurs. Finally, if the task chooses not to suspend, the task continues immediately with other jobs.

If the task becomes suspended it can be resumed in either

- first-in, first-out (FIFO) order; or
- highest priority first (HPF) order.

Which policy to use is specified when the shared resource is created. If tasks are suspended in first-in, first-out (FIFO) order, they are resumed in the order they were suspended. Otherwise, if tasks are suspended in HPF order, they are resumed from high priority to low priority.

When race conditions occurs, tasks are subject to priority inversion and to mutual deadlocks. NP does *not* implement resource allocation protocols to bound priority inversion and to avoid mutual deadlocks.

A task can also be *explicitly suspended* e.g. by another task or HISR. If this happens, the suspended task must also be explicitly resumed again. Finally, a task can also go to sleep (self-suspension). The amount of time to sleep is specified in timer ticks.

7.4 Determinism

Most of the system services in NP are performed in constant time. However, there are some cases were this is not true:

- When a task is suspended in priority order. The processing time required to suspend the task is affected by the number of task currently suspended on that particular resource.
- When allocating memory from a dynamic memory pool. This is due to possible memory fragmentation. NP uses a first-fit algorithm, which is basically a linear search, and as a result the worst-case performance depends on the amount of fragmentation.
- When broadcasting a message over a mailbox, a queue or a pipe. The processing time is linear to the number of the tasks waiting for message.
- When sending/receiving a message over a queue or a pipe. The processing time required to copy a message is linear to the size of the message.
- When setting an event flag in an event group. The processing time required depends on the number of tasks currently suspended on that event group.
- When a signal-handler invokes. The processing time depends on the signal-handling routines themselves.
- When an application timer executes a user-supplied expiration routine. The processing time depends on the expiration routines themselves and the number of timers that expires simultaneously.

7.5 Kernel Internals

In the beginning of our investigations of the Nucleus Plus operating system we were told that it was believed that when using a NP semaphore as synchronisation primitive, resources protected by this were requested in a way similar to the priority inheritance protocol (PIP). Though we were not able to find anything about this in the documentation of Nucleus Plus [ATI, 2000a] [ATI, 2000b].

At the Nucleus Plus web-site (www.atinucleus.com) we found the following question and answer in the pages of Nucleus Plus frequently asked questions (FAQ):

> Q: Does Accelerated Technology do anything about priority inversion?
>
> A: We (ATI) have a protection mechanism that works like a software monitor. It uses a scheme comparable to priority inheritance when a higher priority task needs a resource that is in the exclusive possession of a lower priority task. The lower priority task is allowed to run just far enough to release the resource, and then control returns to the higher priority task, which is then able to continue, with exclusive access to the resource it needed.

There were no further explanations, e.g. how to utilise this protection mechanism. The answer did neither confirm nor disprove that NP semaphores were implemented in a PIP-like way. We decided to investigate the source code of NP by ourselves in order to find whether there was a special protection mechanism or not.

By studying relevant parts of the NP source code we found that NP semaphores did only implement a traditional fixed priority synchronisation protocol.

Nevertheless, we found that *kernel data structures* of the Nucleus Plus was in fact protected by a PIP-like mechanism, i.e. in the way described in the Nucleus Plus FAQ above.

This means that a task operating inside the kernel can become preempted. If the preemptor invokes a system call that requests for the same kernel data structures, the preempted task is 'pushed through' the critical section of the kernel. As soon as the preempted task has left the critical section the preemptor executes again and allocates the requested internal data structures.

We believe that this mechanism is only supposed to be used by the system services of kernel, i.e. the mechanism was not described in details anywhere. We think that the reason why to let a task become preempted during a system call is an attempt to reduce interrupt latency.

7.6 Summary and Discussion

We have examined the functionality and concepts of a commercial real-time operating system. The operating system was introduced to us by Critical ApS which uses the particular one in their development of embedded software systems.

By investigating parts of the Nucleus Plus source code we have obtained insight into the internals of the kernel. During this journey we have not been further impressed of the implementation.

We have to criticise that the equipped documentation did not mention the kernel synchronisation mechanism as discussed in section 7.5. Even though it was somehow mentioned on the web-site of ATI we had to investigate large parts of the source code in order locate the mechanism and furthermore to understand how the mechanism worked. The protection mechanism was complex and used widely in all system services increasing the operating system overhead. Based on our investigations of the source code we believe that the mechanism is created as a stop-gap solution to an unforeseen problem discovered in the kernel — the possibility for chain blocking.

We think that the HISR concept is indeed superfluous. HISRs are scheduled by the same scheduler as tasks. HISRs can be considered as tasks of higher priority that may not suspend. If it was possible for a LISR to release a semaphore for a waiting task a similar effect could be obtained.

From the previous sections it should appear that Nucleus Plus provides a very large set of different functionalities and concepts. It seems that the creators of NP tries to satisfy everybody by providing all kinds of constructs. We think that a lot of the functionality is overlapping which makes it hard to figure out which mechanism to use, e.g. which mechanism is the fastest etc.

Even though the source code is provided it is way too difficult to understand the most essential parts of the operating systems, e.g. the scheduler and internal protection mechanism.

8 Implementing the Computational Model

In chapter 6 a computational model was defined. In this chapter we show an implementation of the computational model in the Nucleus Plus (NP) operating system, i.e. we make it possible for a hard real-time application to comply the computational model when implemented upon the particular operating system.

In section 8.1 we describe a way of creating periodic tasks in NP. In section 8.2 we show implementations of the non-preemptive critical section protocol (NPCS) and the highest locker protocol (HL).

8.1 Periodic Tasks

Nucleus Plus has a system call which let a task sleep for a number of timer ticks. This can be used in the following scenario to create a periodic task:

1. Execute a job
2. Read the system clock
3. Compute the amount of time to sleep
4. Go to sleep
5. Resume when timer expires and go to step one

There is a problem with the presented scenario. After the system clock is read in step two, the system clock may tick before the task goes to sleep. E.g. if the task is preempted right after step three and is resumed two ticks afterwards the computed amount of time to sleep is now incorrect. This jitter may accumulate over time.

The problem is that the NP sleep system call uses a relative timer. Thus, we found the use of NP application timers more suitable in the creation of periodic tasks, cf. section 7.1. Application timers can be set up to expire periodically. The first time of expiration can be used to phase the periodic expirations. E.g. with a phase of seven and a period of ten the timer can have the absolute expiration pattern: 7, 17, 27, 37, etc.

We now illustrate by example the creation of a periodic task in Nucleus Plus using application timers. We emphasise that the relative deadline of the task must be less or equal to the period of the task, i.e. $D_i \leq p_i$, cf. section 6. The example shows the creation of a periodic task $T_1 = \{\phi_1 = 5, p_1 = 10, \pi_1 = 3\}$.

A periodic task must perform a job within every period. When a job is completed the NP task must be suspended until its release at the beginning of a new period. In NP this could be expressed with:

```
NU_TASK TaskPtr;

/* Definition of functional behaviour of a periodic task. */
VOID Task(UNSIGNED argc, VOID *argv) {
  while (1) {                                          /* Infinite loop. */

    /* Job start. */
           .
           .
           .
    /* Job end. */

    NU_Suspend_Task(&TaskPtr);                       /* Self-suspension. */
  }
}

/* Function to be called when a new period begins. */
VOID ResumeTask(UNSIGNED id) { NU_Resume_Task(&TaskPtr); }
```

First, a task pointer is defined. Next, the functional behaviour of the task is defined within the infinite loop of the `Task` function. The loop ends with a self-suspension. Finally, each time a new period begins the NP task must be resumed again with a call to the `ResumeTask` function. We return to the caller of the `ResumeTask` function.

We now define the phase, the period, and the priority of the task:

```
#define TASK_PHASE 5
#define TASK_PERIOD 10
#define TASK_PRIORITY 3
```

In the initialisation part of a NP application a task is created with the system call `NU_Create_Task` [ATI, 2000b, p. 24]:

```
NU_Create_Task(&TaskPtr,                  /* Pointer to task           */
               "Task T1",                 /* Name tag                  */
               Task,                      /* Pointer to function       */
               0,                         /* Argument count            */
               NU_NULL,                   /* Pointer to arguments      */
               pointer,                   /* Pointer to stack          */
               2000,                      /* Stack size in bytes       */
               TASK_PRIORITY,             /* Priority level            */
               0,                         /* Time slice                */
               NU_PREEMPT,                /* Task preemption allowed   */
#if TASK_PHASE==0                         /* If the phase == 0 release */
               NU_START                   /*   the task right away     */
#else                                     /* If the phase != 0 suspend */
               NU_NO_START                /*   the task right away     */
#endif
               );
```

The task is created with the previously defined task pointer, function, priority, etc. The preprocessor directive makes sure to start the task right away if the phase is zero. If the phase is non-zero the task is suspended from the beginning. We do not promote this kind of C preprocessor use, but in this example it perfectly illustrates the differences in the creation of a periodic task. In the example we have left out the allocation of memory for the task stack.

A periodic timer is also created in the initialisation part with the `NU_Create_Timer` system call [ATI, 2000b, p. 204]. Every time the timer expires it invokes the previously defined function, `ResumeTask`, in order to resume the task.

```
NU_Create_Timer(&TimePtr,              /* Pointer to timer             */
                "Timer T1",            /* Name tag                     */
                ResumeTask,            /* Pointer to function          */
                42,                    /* Unique timer id (not used)   */
#if TASK_PHASE==0                      /* If the phase == 0 expire the */
                TASK_PERIOD,           /*   timer at phase == period   */
#else                                  /* If the phase != 0 expire the */
                TASK_PHASE,            /*   timer at phase == phase    */
#endif
                TASK_PERIOD,           /* Period                       */
                NU_ENABLE_TIMER);      /* Enable timer right away      */
```

If the phase is non-zero the timer is set to expire for the first time when the phasing is over. If the phase is zero the timer is set to expire for the first time when the second period begins. After the first timer expiration the timer expires periodically with the defined period.

8.2 Synchronisation Protocols

This section presents implementations of the non-preemptive critical section protocol (NPCS) and the highest locker protocol (HL) in the Nucleus Plus operating system. Both protocols can ensure that a task gets mutual exclusive access to critical regions. The protocols can be used in the construction of more structured synchronisation mechanisms like a monitor.

Section 8.2.1 presents the implementation of the NPCS and section 8.2.2 presents the implementation of the HL. The overhead of the implemented protocols and the built-in NP semaphore is compared in section 8.2.3.

8.2.1 Non-Preemptive Critical Section Protocol

As mentioned in section 5.2 there exists two variants of this protocol:

Task level Here inter-task preemption is not possible, e.g. a normal task of higher priority cannot preempt a task of lower priority. A task can still be preempted by an interrupt service routine. This variant is sufficient if the protocol is used to protect a inter-task critical region.

Interrupt level Here preemption is not possible at all, hence a normal task cannot be preempted by an interrupt service routine. This solution ensures mutual exclusive access for both tasks and interrupt service routines.

Task Level Implementation

An implementation of the NPCS task level variant in NP is sketched below. The implementation uses a NP system call, `NU_Change_Preemption`, which enables or disables the possibility to preempt the running task [ATI, 2000b, p. 20].

```
/* Outside critical region. */

NU_Change_Preemption(NU_NO_PREEMPT); /* Disable task level preemption. */

/* Inside critical region. */

NU_Change_Preemption(NU_PREEMPT);     /* Enable task level preemption. */

/* Outside critical region. */
```

Interrupt Level Implementation

An implementation of the NPCS interrupt level variant in NP is sketched below. The implementation uses a NP system call, `NU_Control_Interrupts`, which enables or disables all interrupts [ATI, 2000b, p. 224].

```
/* Outside critical region. */

NU_Control_Interrupts(NU_DISABLE_INTERRUPTS);   /* Disable interrupts. */

/* Inside critical region. */

NU_Control_Interrupts(NU_ENABLE_INTERRUPTS);     /* Enable interrupts. */

/* Outside critical region. */
```

8.2.2 Highest Locker Protocol

This section presents two implementations of the highest locker protocol (HL) in the NP operating system. The first one is simple as a task is limited to use a single resource at the same time. The second one is general and implements all to the rules presented in section 5.3.

Priority Manipulation in Nucleus Plus

As mentioned in section 5.3 the HL uses only priority manipulation to synchronize tasks. In NP the following system calls are needed to change the priority of a task:

`OPTION` `NU_Change_Priority(`**`NU_TASK`** `*task,` **`OPTION`** `new_priority)`

> This function changes the priority of the specified task. It returns the old priority to the caller. If the new priority necessitates a context switch, control is transferred back to the system. [ATI, 2000b, p. 21]

`NU_TASK` `*NU_Current_Task_Pointer(`**`VOID`**`)`

> This function returns a pointer of the calling task. [ATI, 2000b, p. 26]

The result of the latter is required in the call to `NU_Change_Priority`. Both functions are constant time operations.

Simple Implementation

By limiting a task to use only one shared resource at the same time, it is possible to keep the implementation simple. When a task enters a critical region its priority is

saved when it inherits the ceiling priority of the resource. When it leaves the critical region it restores the saved priority. A data structure to maintain this scenario is:

```
typedef struct critical_region_control_block {
  OPTION priority;     /* Must be initialised to the ceiling priority. */
  OPTION saved_priority;
} CRCB;
```

Every shared resource must have an instance of the data structure **CRCB** attached. The field `priority` must be initialised to the ceiling priority of the resource. The field `saved_priority` stores the original task priority while the task is inside the critical region.

Two functions are needed. One must be called when the task enters the critical region and one must be called when it leaves the critical region:

```
VOID enter_crit(CRCB *region) {
  region->saved_priority =
    NU_Change_Priority(NU_Current_Task_Pointer(), region->priority);
}

VOID leave_crit(CRCB *region) {
  NU_Change_Priority(NU_Current_Task_Pointer(), region->saved_priority);
}
```

The function `enter_crit` immediately sets the priority of task to the ceiling priority of the shared resource. Now the task has exclusive write access to the data structure and then saves the old priority here. The function `leave_crit` restores the original priority of the task.

The complete interface and implementation of the simple implementation can be found as listing A.1 and listing A.2 in appendix A.

The simple implementation is sufficient if a task accesses a single resource at the same time. Nevertheless, it can be used to let a task request more than one resource at the same time *if* the policy below is followed:

- Resources must be requested in increasing ceiling priority order.
- Resources must be released in reverse order they were requested.

This policy may be difficult to follow in general.

General Implementation

To support the request for an arbitrary number of resources and an arbitrary request and release order, a list of inherited priority ceilings must be associated to every task. Then, every time a resource is requested, its priority ceiling is added to this list. When the resource is released again, its priority ceiling is removed from the list. The priority of the task must always be equal to the greatest priority in the list. To maintain this scenario the following data structures are needed:

```
typedef struct critical_region_control_block CRCB;

typedef struct critical_region_control_block {
  OPTION priority;     /* Must be initialised to the ceiling priority. */
```

```
  CRCB *next, *prev;
};
```

As in the simple implementation every shared resource must have an instance of the data structure **CRCB** attached. The priority field must be initialised to the ceiling priority of the resource. The `next` and `prev` fields are pointers linking the **CRCB**s for the regions allocated by a task into a list.

```
typedef struct thread_control_block_extension {
  OPTION normal_priority;    /* Must be initialised with the priority. */
  OPTION current_priority;   /* Must be initialised with the priority. */
  CRCB *region_list;         /* Must be initialised with null pointer. */
} TCBE;
```

Every task should have an instance of the data structure **TCBE** attached. The field `priority` and `current_priority` must be initialised with the normal priority of the task. The field `region_list` is a pointer to the list of regions the task is currently inside.

Figure 8.1 shows how the data structures are maintained for a task, T_1, which allocates the resources, R_1, R_2, R_3, and deallocates the resource R_2. The fields `region_list`, `next` and `prev` are illustrated as arrows in the figure.

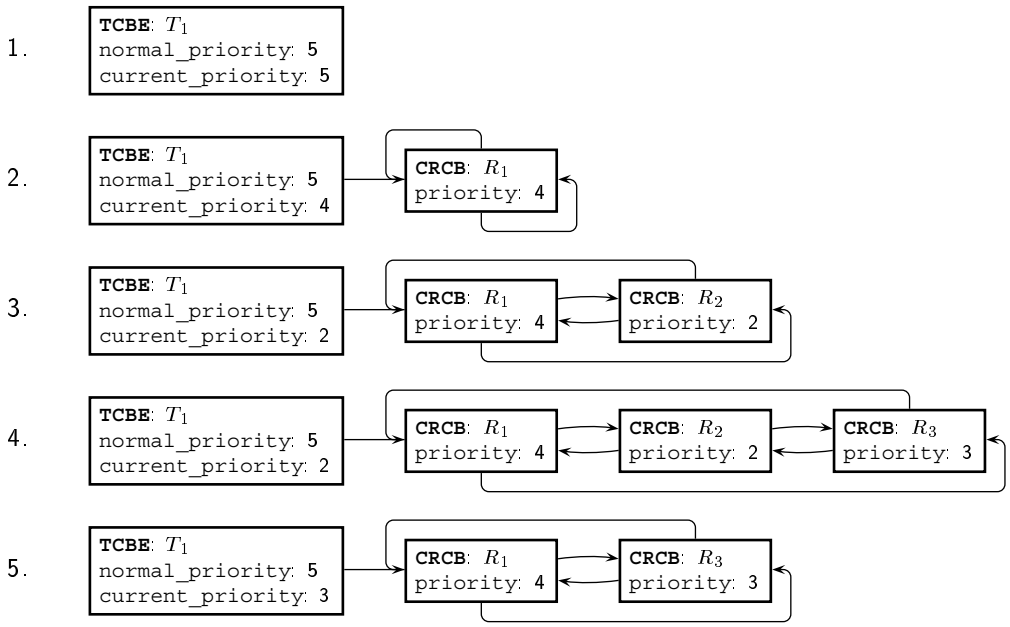

Figure 8.1: *Data structures of the general implementation of the highest locker protocol. The figure shows how the data structures are maintained for a task which allocates and deallocates resources.*

The two functions, `enter_crit` and `leave_crit`, are in the general implementation more complicated:

```
VOID enter_crit(CRCB *region, TCBE *thread) {
  if (region->priority > thread->current_priority) {
    NU_Change_Priority(NU_Current_Task_Pointer(), region->priority);
    thread->current_priority = region->priority;
  }
```

```
  insert(thread->region_list, region);
}
```

The function `enter_crit` first checks if the critical region has a higher ceiling priority than the current task priority. If this is the case the task inherits the ceiling priority of the region. Now the task have exclusive write access. It updates the field `current_priority` to its new priority. Finally, the new ceiling priority region is inserted into the region list by a call to the function `insert`.

```
VOID leave_crit(CRCB *region, TCBE *thread) {
  remove(thread->region_list, region);
  thread->current_priority =
    find_greatest_priority(thread->region_list,
                           thread->normal_priority);
  NU_Change_Priority(NU_Current_Task_Pointer(),
                     thread->current_priority);
}
```

The function `leave_crit` first removes the ceiling priority from the region list for the region just left with a call to the function `remove`. Next, it has to find out what the priority should be: the highest priority of all the regions in the list. To do this the function `find_greatest_priority` is called which returns the greatest priority of all the elements in the list. If the list is empty it returns the normal priority of the task. Finally, the new priority of the task is set.

The implementation of the `insert`, `remove` and `find_greatest_priority` functions are not listed in this section. The complete interface and implementation of the general implementation, including the list operations, can be found as listing A.3 and listing A.4 in appendix A.

The list functions `insert` and `remove` are implemented as constant time operations, while the latter, `find_greatest_priority`, is linear in the number of regions the tasks is currently inside. All the functions can be implemented as constant time operations, but this will lead to another data structure which requires more memory. The memory required is linear in the number of priority levels. A way to do this can be found in [Labrosse, 1998].

In the general HL implementation every task and resource that uses the protocol have a data structure attached. This makes the protocol more awkward to use. For tasks it would be suitable to extend their thread control block (TCB) with the list of resources they possess [Liu, 2000, section 12.3.1].

The general implementation has not been developed further since the simple implementation is sufficient enough for this project.

8.2.3 Overhead Comparison

In this section we compare the overhead of the built-in NP semaphore and the implementations of the NPCS and the HL. Obviously, the overhead is dependent of the execution environment. In this project the execution environment is given by Critical ApS.

At Critical ApS they develop applications for the *Infineon C164* family of microcontrollers. Together with the Nucleus Plus operating system this microcontroller forms

our execution environment. In this section we will not go into great details about the microcontroller, but refer to a more detailed description given in section 10.4.1.

Critical ApS uses a C164 family C cross-compiler, cross-assembler, debugger etc. by *Tasking Incorporated* [Tasking, 2000a] [Tasking, 2000b] [Tasking, 2000c]. The debugger, called *CrossView Pro*, can measure the number of *clock cycles* for every executed machine instruction. For example, it is possible to count the number of clock cycles between two breakpoints. With this feature at disposal very precise execution times for the system services can be found. Note that for a C164 it takes two clock cycles to perform one machine cycle [Infineon, 1999].

We define the overhead of a synchronisation protocol as the number of machine cycles it takes for a task only to enter and to leave a critical region using the protocol. To measure the overhead of the two HL implementations, a test application were created, cf. listing A.5 in appendix A. To measure the overhead of the other synchronisation protocols. we combined the results from table B.1 in appendix B. The final result can be found in table 8.1.

Synchronisation protocol	Machine cycles	Relative
Built-in NP semaphore (BCET)	355	32
Built-in NP semaphore (WCET)	1,102	100
Non-preemptive critical section (task level)	490	44
Non-preemptive critical section (interrupt level)	52	5
Highest locker protocol (simple)	1,178	107
Highest locker protocol (general)	1,299	118
Highest locker protocol (raw)	1,052	95

Table 8.1: *Overhead of the implemented synchronisation protocols.*

Discussion

There are both a best-case execution time (BCET) and a WCET for the NP semaphore. The BCET is where the task obtains and releases the semaphore without producing a context switch. The WCET is where the task produces a context switch obtaining and releasing the semaphore. Semaphores are considered a traditional way of protecting a critical region. Because of this the WCET is given a relative index of 100 in table 8.1. The semaphore BCET is relative small compared to most of the other protocols, but a BCET cannot be used in hard real-time systems. More important is that the semaphore do not provide uncontrolled blocking times — more overhead is better than uncontrolled blocking times.

The implementation of the task level variant of NPCS produces less than half the overhead of the semaphore WCET. Note that the number of machine cycles in this result is based on a worst-case situation were the task gets preempted as soon as it leaves the critical region.

The implementation of the interrupt level variant of NPCS is very fast. It takes fifty-two machine cycles in total to disable and enable all interrupts using the NP system services. By studying the instruction set of the microcontroller, we find that it will take five machine cycles in total to disable and enable interrupts using

pure assembly code [Infineon, 2001]. The operating system overhead is considerably larger compared to this.

The simple implementation of the HL gives only seven percent more overhead than the semaphore. The general implementation of the HL introduces eighteen percent more overhead than the semaphore. Note that this is a best-case result, i.e. the task enters only one critical region.

Finally, we have a 'raw' implementation of the HL. By raw we mean the minimum of system calls and computations needed to obtain the same behaviour as the HL, i.e. two priority change system calls. The programmer must now take care of setting the correct priority levels when entering and leaving critical regions. We discover that the raw implementation has less overhead than the NP semaphore.

We find there is a large variation in overhead of the synchronisation protocols. Of course, another execution environment will give different measurements. Below we give some considerations about protocol overhead:

Atomic critical regions Most CPUs has atomic read/write operations of simple data types, e.g. a 16-bit word on the Infineon C164, thus no synchronisation protocol is needed and thus no protocol overhead. The programmer has to be careful, e.g. if a 16-bit word is put on an odd memory address, the operation might *not* be performed in one atomic operation, but in two operations which may be interrupted. If the application has to be ported to another execution environment, this environment has to be studied carefully to ensure atomicy is kept.

Small critical regions If a task spends ten machine cycles inside a critical region, but it takes a thousand machine cycles to enter and to leave the region, the protocol overhead is relative big compared to the time spend inside the region.

High frequency used critical regions The overhead accumulates faster.

9 The Development Process

In this chapter a development process well suited for the development of hard real-time systems is presented. The objective of the *hard real-time development process* is to trace a *feasible* design from requirements to deployment. A design said to be feasible, if and only if a feasible schedule exists for the task set given by the processes of the design. Thus, during development process the feasibility of a design is repeatedly assessed using the theory of schedulability analysis.

In the discussion of the development process we focus on the phase of architectural design. A design method that addresses the architectural design phase is introduced.

9.1 The Design Phase

We adopt the following view of the design phase of the development process. The view was formulated by Alan Burns and Andy Wellings [Burns and Wellings, 1995].

We take a constructive view of the design phase, by describing the phase of system design as a progression of increasingly specific *commitments*. These commitments define properties of the system design, which designers operating at a more detailed level are not allowed to change.

The aspects of a design to which no commitment is made at some particular level in the design hierarchy are *obligations* that lower levels of the design must address. The initial obligations in the design phase are given by the requirements defined for the system. The requirements may also contain commitments to the structure of the system, in terms of interface definitions and processes. However, the detailed behaviour of the defined processes remains the subject of obligations which must be met during further design.

The process of developing and implementing a design, i.e. transforming obligations into commitments, is often subject to *constraints* imposed primarily by the *execution environment*. The execution environment is the set of hardware and software components, e.g. processors and real-time operating system, upon which the system is built. It may impose both resource constraints, e.g. processor speed, timing constraints, e.g. the period with which a value must be written to a register of a peripheral, and constrains of mechanism, e.g. interrupt priorities and mutual exclusive access to critical sections. For a given execution environment these constraints shall be considered fixed.

When the design phase is complete all obligations have been transformed into commitments under the constraints imposed by the execution environment. Thus, the collection of commitments made during the design process constitutes the design of a system.

9.2 A Design Method

When considering design methods, it is a general misconception that a method will provide a design to the designer if he simply performs the individual steps of the method.

A method cannot make commitments for the designer. The design phase is a creative phase of the development process. Thus, it is the designer's creative ability, ability to abstract, and experience that will enable him to produce the design. A method simply provides a systematic and focused approach for reaching a design, by making commitments in a systematic manner.

This section briefly introduces the design method described in [Løvengreen, 1997] and [Rischel et al., 1987]. The method defines the following steps:

System partition decomposes the system into a number of *subsystems*.

Interface identification finds the *events* linking the system and its environment, i.e. input and output events. Events are typed to indicate the information they carry.

Event structuring characterises the behaviour of the system. This is done by identifying the temporal ordering of the input and output events. The result is a number of *event sequences*. The event sequences are all put in parallel, hence they identify the processes of the system.

Program structuring where the event sequences are transformed into *abstract programs*, which are expressed in a CSP-like process language where processes communicate over synchronous channels corresponding to the events of the system. Shared resources are identified by a *data flow analysis* and are represented by processes denoted *state monitors*.

9.2.1 Design Documentation

An important part of the design phase is the documentation of the design, that is the documentation of how obligations are transformed into commitments. As a method provides a structured approach to making commitments, a method could be seen as a structured approach to documenting the design.

For the method introduced in this section, the final design documentation is the program structure given by the abstract programs for the system processes and state monitors. The abstract programs express all the commitments that has been made through the design process.

The first three steps of the design method produces a series of temporary documentation: a list of identified subsystems, a list of identified input and output events, and the event sequences. The temporary documentation captures the progression of increasingly specific commitments, which formed the basis for the program structure.

9.3 The Traditional Development Process

Most traditional software design methods incorporate a development process in which the following activities of system development are recognised:

Requirements definition produces a specification of the functional as well as non-functional requirements to the behaviour of the system. It is outside the scope of this thesis to discuss requirements specification and analysis.

Architectural design where the top-level structure of the system is defined.

Detailed design completes the system design based on the architectural design.

Implementation of a system according to the detailed design.

Testing of a system implementation. Testing can be prohibitive time consuming if done exhaustively, but it is unreliable if the coverage of the test is incomplete.

The development process is iterative. Thus, we may revert to a previous activity to solve a problem identified in the current activity. For example, we may return to the detailed design activity if we identify a problem in the implementation activity.

9.4 The Hard Real-Time Development Process

For hard real-time systems the traditional development process has the disadvantage that timing problems will only be recognised during testing, or worse after deployment. The following observation is found in [Burns and Wellings, 1995]:

> ... it has been common practise of system developers to consider non-functional requirements comparatively late in the development process. Often timing requirements are viewed simply in terms of performance of the completed system. Failure to meet the required performance often lead to ad hoc changes to the system. This is not a cost effective development process.

To avoid this problem we decompose the architectural design activity of the traditional development process into two activities:

Logical design where the commitments are made that are independently of the constraints imposed by the execution environment. The objective is to satisfy the functional requirements of the system.

Physical design maps the logical design onto the execution environment within the imposed constraints. Hence, the functional as well as non-functional requirements are to be satisfied by the physical design.

The resulting hard real-time development process is illustrated in figure 9.1 on the following page. The individual phases of the process is addressed in the remainder of this chapter.

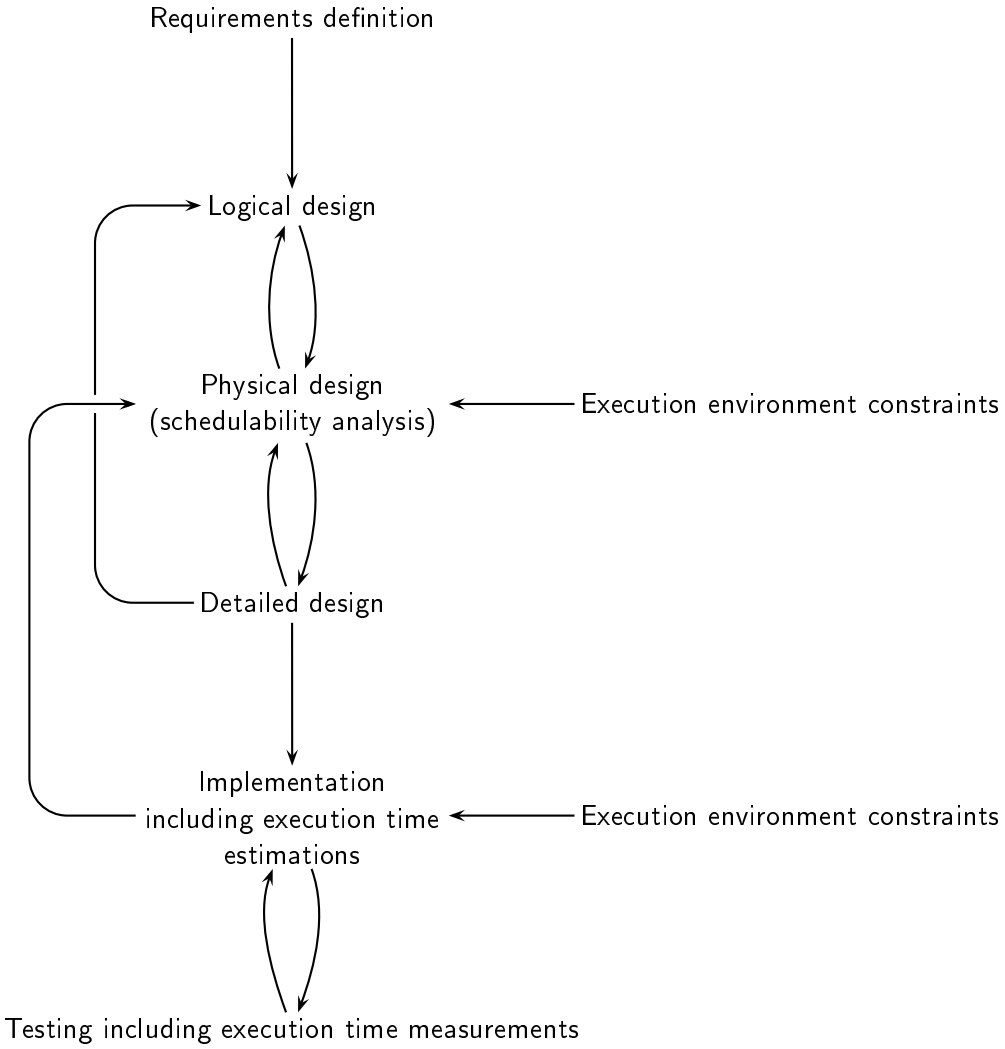

Figure 9.1: *The hard real-time development process.*

9.5 Logical Design

The logical design is the set of commitments which can be made independently of the execution environment. Thus, the product of the activity shall be a design satisfying the functional requirements. The existence of timing requirements may, however, strongly influence the development of the logical design.

We will base our development of the logical design on the method introduced in section 9.2. From the logical design activity we proceed to the physical design activity.

9.6 Physical Design

The physical design is developed from the logical design, however its development will usually be an iterative and concurrent process, in which both the logical and physical designs are modified.

The physical design maps the logical design onto the execution environment while taking the constraints of the execution environment into account. The method used for creating the logical design is also used in the physical design phase.

In the abstract programs defined in the physical design are assigned their real-time attributes, i.e. priority, period, phase, deadline, and an initial execution time estimate.

Hence, the final documentation of the physical design is the program structure given by abstract programs for processes and state monitors, and a situation table documenting the real-time attributes of the abstract programs. A situation table were introduced in section 2.8.

As a part of the physical design phase the schedulability of the physical design is analysed. We may proceed from the physical design phase when the schedulability analysis concludes that the design is feasible, otherwise we must return to the logical design phase. If the timing requirements for the system are too restrictive it may prove impossible to develop a feasible design for the given execution environment.

Thus, the product of the physical design activity shall be a design that satisfies the functional requirements and the timing requirements.

Mapping Logical Design to Execution Environment

Mapping the logical design onto the execution environment to produce a physical design may not be straightforward. In particular, there may be several ways to do this.

The execution environment may provide specialised hardware features, which may be used to increase the performance of a system. Such features could be hardware shortcuts for performing certain multiprogramming operations, e.g. scheduling of tasks, monitor operations, or direct memory access (DMA).

The software components of the execution environment may also provide several possibilities for mapping the logical design into software components, e.g. different

types of tasks and interrupt handlers.

Not all events necessary for modelling the behaviour of components of the execution environment were considered in the logical design. Thus, in the process of mapping the logical design onto the execution environment, additional events may be introduced if they are necessary for modelling the behaviour of components in the execution environment, onto which the logical design is mapped.

Additional events of the execution environment are incorporated into the logical design by reapplying the method used for creating the logical design. Thus, the additional events shall be temporally ordered with the events of the event sequence for the abstract program, which is being mapped onto the execution environment. Following the formation of new event sequences, new abstract programs and state monitors are formed for the physical design.

Schedulability Analysis

The physical design forms the basis for assessing whether non-functional requirements of the application are met, once the detailed design and implementation have taken place. The physical design addresses the timing constraints and the necessary schedulability analysis that can validate that the final system will meet its deadlines.

As the process of developing the physical design is an iterative and concurrent process, the schedulability analysis should be applied to the physical design as early as possible. As several alternative implementations may be possible, the schedulability analysis shall be used in the assessment of the alternatives for the purpose of selecting an implementation.

To undertake the schedulability analysis, the time dependent behaviour of the target processor and other aspects of the execution environment must be available, e.g. estimates for the overhead imposed by the operating system.

Additionally, it is necessary to make some initial estimations of the resource requirements of the physical design, e.g. estimates of the execution time of an implementation of the abstract programs of the physical design. The initial resource requirements are subject to modification and revision as the physical design is developed, implemented, and tested. In this way a *feasible* design is traced from requirements to deployment.

9.7 Detailed Design, Implementation, and Testing

Once the logical and physical design activities are complete, the detailed design activity can begin. When detailed design completes new execution time estimates must be produced and the schedulability of the detailed design analysed. If the detailed design is feasible we can proceed to the implementation phase. If the detailed design is infeasible or design problems are discovered we must return to the logical or physical design phase to create an improved design.

The detailed design phase is followed by the implementation phase, where the code for the system is written. When the code has been written the execution times of the implementations for the abstract programs must be measured. Given the measured

execution times the schedulability of the system must again be assessed. If the schedulability analysis concludes that the implementation is feasible, we proceed to the testing phase. If the implementation is infeasible, i.e. the execution time measurements deviate from the estimates of the detailed design phase, it may suffice to return to the detailed design phase if the deviations are small, otherwise we must return to either the logical or physical design phase to improve the design.

In general, the detailed design phase of the hard real-time development process should be performed as in the traditional development process, with the addition of the execution time estimation.

The execution time measurement performed in the implementation phase is a complex issue. In particular, it is necessary to constrain the way code is written so that analysis of execution time for the final system is not too pessimistic. For example, all loops must be bounded. An introduction to program analysis is out of the scope of this thesis.

9.8 Summary

In this chapter we introduced a development process which is better suited for the development of hard real-time systems than the traditional development process.

The hard real-time development process divides the architectural design phase into a logical design phase and a physical design phase. The physical design phase addresses the schedulability of the physical design.

In the following phases of the development process: detailed design, implementation, and testing the schedulability is repeatedly reassessed as the execution time estimates are revised. Thus, the hard real-time development process allows a feasible design to be developed and traced from requirements to deployment.

In the discussion of the hard real-time development process we have carefully avoided the term *refinement*. When a design is developed or modified it is not guaranteed that the behaviour of the new design is a refinement of the previous design, e.g. when mapping the logical design onto the execution environment.

10 Case Study: Motor Control System

To illustrate the use of the concepts and techniques introduced in this thesis, we now return to the example of a control system as presented in figure 1.1 in the introduction. Two main control principles are used in such control systems: *closed-loop* and *open-loop* control. A closed-loop control system reads the current state of the controlled system. This input, denoted *feedback*, and the reference input are used in a *control algorithm* for computing a new *control output*. This output is based on the deviation between the current state and the desired state. The output is used in the activation of the actuators, which brings the controlled system closer to its desired state. In an open-loop control system no feedback from the controlled system is used in the control algorithm.

In this case study we shall study a closed-loop control system, which controls a variable speed alternating current (AC) motor. This kind of motor is used in many kinds of electrical machinery such as washing machines, ventilators and electrical power steerings.

A three-phase, line-to-line, induction motor is a simple, robust, and inexpensive type of electrical motor. A *frequency converter* is the key component in the implementation of an variable speed induction motor. In this case study we shall investigate how a frequency converter for controlling this kind of motors may be constructed using a microcontroller.

The main focus in this case study will be on analysing the timing behaviour of the system which turns out to be very time-critical.

10.1 The Frequency Converter

In a three-phase induction motor the rotor follows the magnetic field rotating in the gap between the rotor and the stator [Danfoss, 1998]. Hence, the wanted number of revolutions per minute of the motor can be obtained by controlling the frequencies of the three phases of the motor's supply voltages. The controlling of the frequencies is implemented by a frequency converter.

Frequency converters have experienced a rapid development since the end of the 1960's. Especially the advances in semi-conductors and microprocessors have lead to progress in this area. However, the basic structure and principles of frequency converters remain the same. The key component in a frequency converter is the *inverter*; a circuit converting a direct current (DC) voltage into a three-phase AC voltage. The DC voltage supply may either be variable or constant. In this case study we assume the DC voltage supply to be constant. Figure 10.1 on the next page gives a structural overview of the frequency converter.

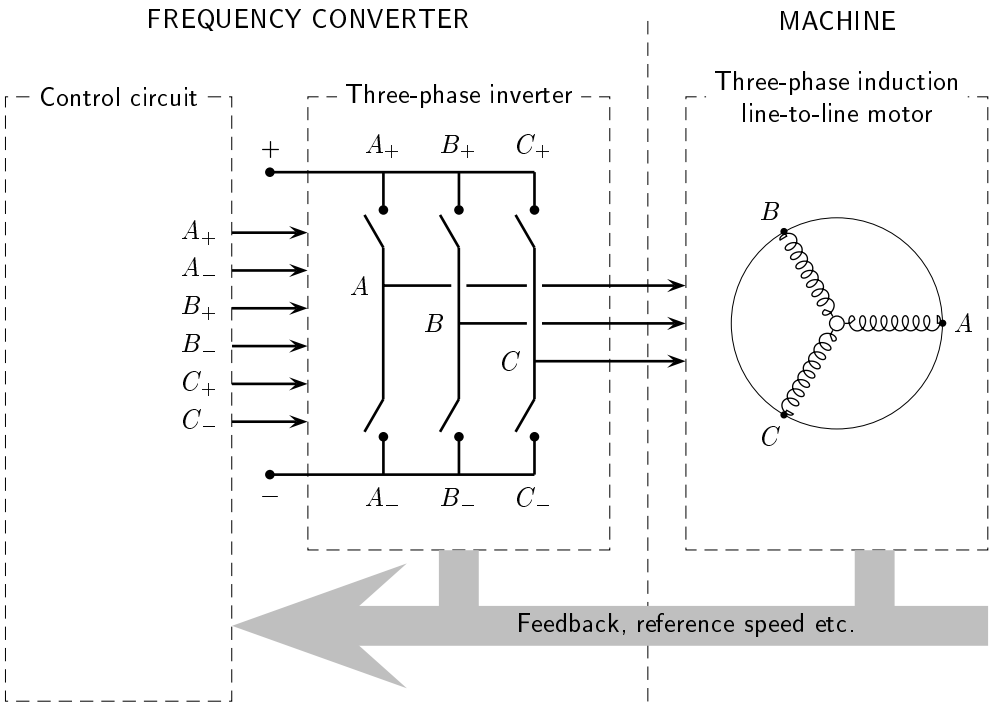

Figure 10.1: *A frequency converter consists of a control circuit and an inverter. The control circuit gets feedback from the inverter and motor, e.g. voltage drop, temperature etc. Furthermore, reference input like a motor-speed is also treated by the control circuit.*

The Inverter

The three-phase AC output voltage is generated in the inverter. Though different inverters exist, their structure is always the same [Danfoss, 1998]. The structure is illustrated in figure 10.1. The main components in an inverter are six switches grouped in three pairs. The switches connect or disconnect each of the three points A, B and C on the rotor to the positive or negative pole on the DC power supply. This enables or disables the flow of current through the rotor; the change in state is considered instantaneous. Today, transistors are used as switches.

The transistors are operated by the control circuit of the frequency converter. In this case study the transistors are operated according to a principle known as *pulse-width modulation (PWM)*. This principle is discussed in section 10.1.

Pulse-Width Modulation

A common way to generate sinusoidal voltages through a three-phase inverter is *pulse-width modulation (PWM)* [Copeland, 1999] [Danfoss, 1998]. In this case study, the inverter consists of six switches grouped in three pairs, one for each phase, cf. figure 10.1.

The three-phase voltages for an induction motor are generated by delivering a DC voltage to the motor in pulses of varying width. Varying the width of the pulses causes different characteristics, e.g. voltage amplitude and frequency.

The two switches in a pair has the property that when e.g. switch A_+ is *on* then

switch A_- is *off* and vice versa. The time interval where a switch is on is called *pulse-width* or *duty cycle*. This is illustrated in figure 10.2. The sum of the positive- and the negative pulse-width equals a *switching period*. The positive pulse-width and negative pulse-width within one switching period forms a *PWM pattern*. A PWM pattern is needed for each switch-pair.

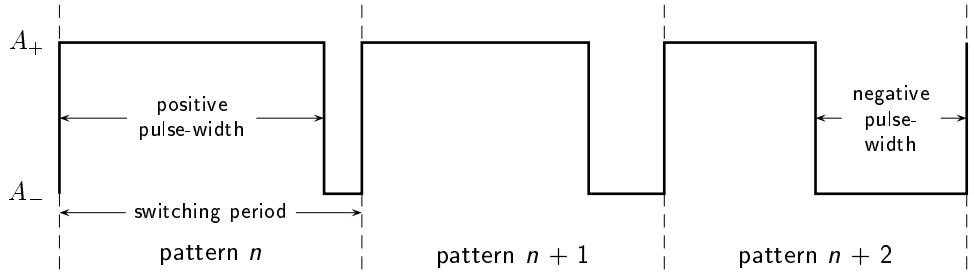

Figure 10.2: *An example of three PWM patterns for a single switch-pair. The sum of the positive pulse-width and the negative pulse-width equals the switching period. The width of the switching period is derived from the switching frequency.*

In practice, to prevent a short circuit of the power source through the inverter, a small period of time where both switches are off, must be inserted when the switches are changed. This is called *dead-time* or *dead-band*. The length of this period depends on how the inverter is built. In this case study dead-time is not discussed any further.

When connecting the inverter to a three-phased induction motor, it is assumed that the inductance of the motor will filter the PWM patterns into a smooth signal [Copeland, 1999]. Figure 10.3 illustrates this. The higher *switching frequency* (rate of PWM patterns), the better approximation of a sinusoidal. The better approximation of a sinusoidal, the cleaner drive of the motor is achieved and thus less heat is produced in the motor. However, a higher switching frequency produces more heat in the inverter. Thus, these parameters must be weighted against each other.

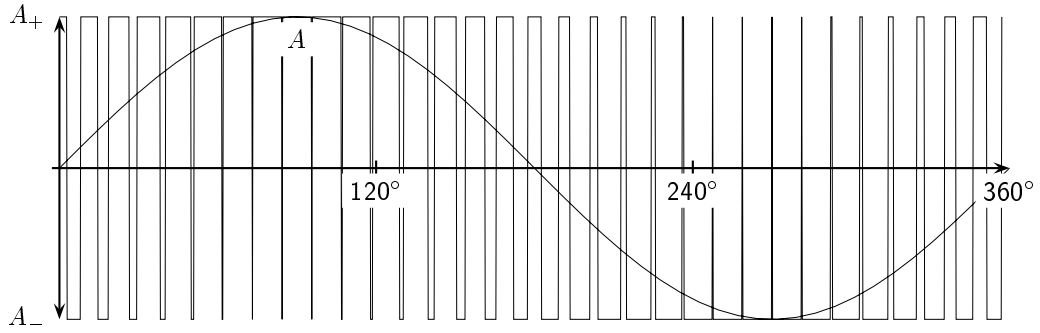

Figure 10.3: *An example of PWM patterns for a single switch-pair showed with an ideal sine wave.*

There are several possibilities for making PWM patterns for the purpose of generating sinusoidal voltages. The *sinusoidally weighted pulse-width modulation (SW-PWM)* is a simple and traditional way to do this [Danfoss, 1998]. The obtained output voltages using SWPWM are illustrated in figure 10.4 on the following page. The figure reveals that the effective voltage is only 87% of the inverter rail voltage. The greater voltage drops obtained, the better utilisation of the power source. Im-

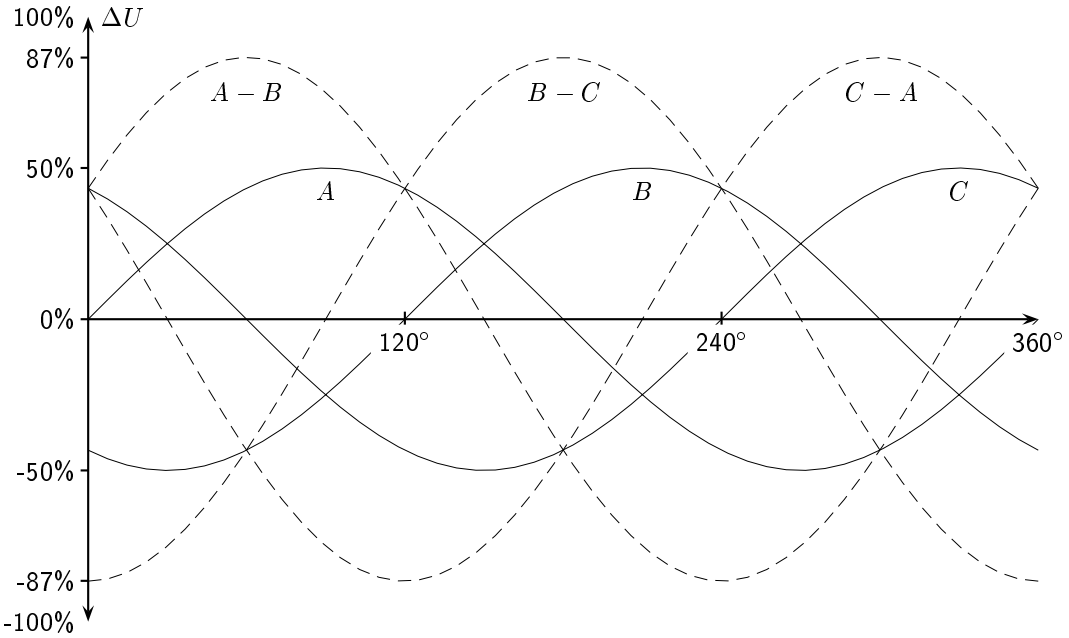

Figure 10.4: *Output voltages in percentage of the inverter rail voltage (ΔU) using sinusoidally weighted pulse-width modulation. The solid lines are the phase voltage. The dashed lines are the line-to-line combined voltages.*

proved methods like *space vector modulation (SVM)* combined with *overmodulation* give voltages up 112% of the inverter rail voltage [Copeland, 1999].

A PWM pattern is generated from a voltage frequency and amplitude, plus a *phase-angle* from the previous PWM pattern. The generation of PWM patterns is not discussed any further in this case study.

10.2 Requirements Specification

In this section we present the requirements for the motor control system. We present functional requirements as well as non-functional requirements.

Functional Requirements

Functional requirements do not relate to time. The motor control system shall be based on the closed-loop control paradigm. The motor shall be controlled on the basis of the following inputs:

- The reference input, which is the desired motor speed, is given by a DC voltage.
- The state of the motor controlled by the motor control system can be deduced from the following feedback:
  - inverter output voltage
  - inverter output current
  - inverter temperature

The output from the motor control system shall be the PWM signals for each of the three phases of the AC motor. Additional details of the control computation is discussed in section 10.3.

All inputs to the motor control system are analog. Hence, an analog digital converter (ADC) is needed to read the analog data and convert it to digital values. This conversion is considered unreliable. To increase the reliability, it is common practice to perform a filtering, combining digital values from several readings. For the motor control system four readings per input is required.

The motor control system will be based on a custom hardware platform developed by the manufacturer of the motor control system. The hardware platform features the Infineon C164CI microcontroller. The hardware platform also includes the sensors providing the inputs to the motor control system. However, in this case study we will abstract from the these, and restrict our attention to the microcontroller. In section 10.4.1 we discuss the details of the microcontroller.

The software of the motor control system shall be based on the Nucleus Plus operating system. The operating system implements a preemptive fixed-priority scheduler. A brief summary of the components of the operating system which are important for the development of the motor control system is given in section 10.4.2.

Non-Functional Requirements

The non-functional requirements of the motor control system are the requirements related to the timely behaviour of the system.

It is estimated that the responsiveness to changes in the input values is sufficient, if the inputs are read periodically with a period of 10 milliseconds. Jitter on the reading of input values are allowed, however the input operation must complete within the 10 millisecond period.

Finally, to ensure a suitable approximation of a sinusoidal, the switching frequency of the motor control system shall equal 10 kHz. No jitter is allowed on the output of PWM signals.

10.3 Control Computations

This section gives a description of the computations performed for a closed-loop motor control system in this case study. This is used as a foundation for the design of the control system.

Based on a reference speed for the motor and feedbacks, the control algorithm must produce two values to be used in the generation of PWM pulses: a voltage frequency and a voltage amplitude. The reference speed and the feedbacks will change over time and the computations must hence be repeated regularly.

At a given reference speed, the motor control system must, at variable motor-loads, maintain a constant motor-speed. By reading the inverter output voltage and current, it is possible to determine the actual speed of the motor and then compensate for a variable motor-load.

Furthermore, the control system must monitor the inverter temperature. If this temperature exceeds certain levels, the control system must interfere, resulting in a stepwise degradation of the dynamic performance. If the inverter temperature becomes too high, a system shutdown should be carried out. We do not consider this situation in this case study.

In this case study, computations performed by the control algorithm are not described. Emphasis is put on the input used by the algorithm, and the computation time used to produce a voltage frequency and amplitude value.

- reference motor-speed (indicated by a voltage to the system)
- inverter output voltage
- inverter output current
- inverter temperature

All inputs are analog. Therefore, an ADC is needed to read the analog data and convert it to digital values. Sometimes this conversion is considered unreliable and might result in a faulty result. Therefore, it is common practice to perform a filtering combining digital values from several readings in order to get a reliable value. In this case study four readings per input are considered suitable.

Based on the output from the control algorithm, the motor control system is supposed to drive the motor by applying PWM signals to the inverter. Figure 10.5 shows a system diagram for the motor control system.

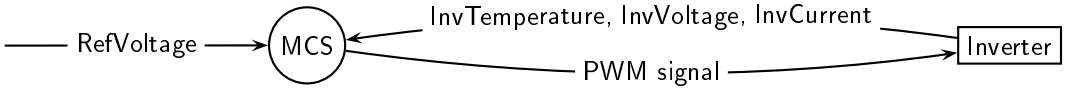

Figure 10.5: *System diagram for the motor control system.*

10.4 Execution Environment

In section 9.1 an execution environment was defined as the set of hardware and software components upon which the system is built. In this section we describe the execution environment of the motor control system, which consists of two components:

- Infineon C164CI microcontroller
- Nucleus Plus real-time operating system

10.4.1 The Infineon C164CI Microcontroller

Several microcontrollers, tailored for motor control systems, exist on the market today. In this project, Critical ApS has introduced us to the *Infineon C164CI*

microcontroller. This single-chip microcontroller has a built-in interface to control a three-phase inverter with PWM signals. Furthermore, it is cheap and supported by the Nucleus Plus operating system.

The Infineon C164CI is a 16-bit single-chip microcontroller, a member of the Infineon C164 class of microcontrollers [Infineon, 1999]. It combines high CPU performance with high peripheral functionality. Some of the relevant key features are highlighted here:

- high performance 16-bit, RISC concept, CPU with a four-stage pipeline
- integrated memory
- 16 priority-level interrupt system
- 8-channel peripheral event controller (PEC)
- integrated peripherals, controlled via special registers:
  - 8-channel 10-bit ADC
  - real-time clock
  - capture/compare unit

In the following sections we summarise relevant features of the microcontroller.

10.4.1.1 The Analog Digital Converter

The C164 family features an 8-channel multiplexed, 10-bit resolution, analog digital converter (ADC) [Infineon, 1999, ch. 18]. The ADC has several modes of operation. In this case study the ADC is operated in *auto scan continuous conversion* mode. Here the ADC converts a set of channels repeatedly. It is only possible to sample from one of the eight channels at the same time. Thus, in auto scan conversion mode the selected set of channels are converted in turns using a multiplexer and *not* in parallel.

10.4.1.2 The Real Time Clock

The C164 family features a real time clock (RTC) to serve for different purposes [Infineon, 1999, ch. 14]:

- system clock to keep hold of the current date and time
- to produce periodic time based interrupts
- 48-bit resolution for long term time measurements

10.4.1.3 The CAPCOM6 Capture/Compare Unit

The *capture/compare unit 6 (CAPCOM6)* of the C164CI supports generation and control of timing sequences with a minimum of software intervention [Infineon, 1999, ch. 17]. The CAPCOM6 unit provides three 16-bit capture/compare channels. As the name implies this capture/compare unit can operate in two modes:

Capture mode Here the contents of a 16-bit timer may be captured into any one of three 16-bit registers on specific external or internal events. In this mode the CAPCOM6 unit can be used for measuring the duty cycle of a PWM signal.

Compare mode In this mode the unit can compare up to three 16-bit values with a 16-bit timer. In case of a match the unit can modify the signals on output pins associated with the CAPCOM6 unit. Below it is explained how the unit in this mode may be used for the generation of PWM signals for controlling a three-phase induction motor.

In compare mode the CAPCOM6 unit provides two output signals for each of the three 16-bit channels. The two signals may have identical or inverted polarity. With inverted polarity the unit is well suited to drive the kind of inverter used in this case study. For example, the first output signal is used to control the positive pulse-width and the second and inverted output signal is used to control the negative pulse-width, cf. figure 10.1 on page 72.

The CAPCOM6 unit contains *shadow latches* which allow the next PWM patterns to be set for later use without disturbing the use of the current PWM patterns. When the current patterns are used, the content of the shadow latches are transfered immediately into the real CAPCOM6 registers and an interrupt is generated. The interrupt indicates that new PWM patterns can be written to the shadow latches. As PWM patterns may be written to the shadow latches before use, the shadow latches provides an simple way of eliminating PWM output jitter.

10.4.1.4 Interrupts Mechanisms

The C164 family supports several mechanisms for fast and flexible response to service requests that can be generated from various sources internal or external to the microcontroller [Infineon, 1999, ch. 5]. In this case study we apply:

Normal interrupt processing This suspends the current path of execution to service an interrupt requesting device. The current program status is saved on the internal system stack. There are sixteen interrupt priority levels.

Interrupt processing using the peripheral event controller (PEC) This is faster alternative to the normal interrupt processing is servicing an interrupt requesting device with the integrated PEC. Triggered by an interrupt it performs a single byte or word data transfer between two memory locations through one of eight programmable PEC service channels. During a PEC transfer the normal program execution is halted for one instruction cycle only. No program status has to be saved. Section 10.4.1.5 goes into details of the PEC.

10.4.1.5 The Peripheral Event Controller

This section describes the peripheral event controller (PEC) in greater detail. As mentioned in section 10.4.1.4, if the response to an interrupt is to transfer data, the PEC can service such an interrupt very fast, with no software intervention.

During normal interrupt processing an ISR takes care of an interrupt request from a peripheral. Here, the current path of execution is interrupted to service the peripheral. In this operation the associated ISR has to save the currently used registers in order to resume the interrupted path of execution after the termination of the ISR. Furthermore, some data processing is performed by the ISR. Transferring data between a peripheral and a memory address is the most common kind of data processing. An ISR triggered with a high frequency introduces a significant CPU overhead. Therefore, the C164 family provides the PEC as a faster alternative to normal interrupt processing.

Triggered by an interrupt request the PEC performs a single byte or word transfer between two memory addresses, e.g. a peripheral register and a memory location. During this transfer the CPU is halted for one instruction cycle only, which reduces CPU overhead dramatically for high frequency interrupt requests. This technique is somehow similar to *direct memory access (DMA)* where some peripheral accesses the memory directly without using the CPU.

There are eight PEC *service channels* available for data transfers. When a service channel has been initialised by a pair source and destination pointers for the data transfer, its service is said to be *enabled*. The two pointers are either fixed or one of them can be incremented after a data transfer. The latter option is convenient when feeding some peripheral, e.g. the CAPCOM6 unit, with data from a table. The table size has an upper limit of either 254 bytes or 127 words.

When the PEC has transfered the last element in a table, it activates a normal ISR and becomes *disabled*. The PEC will not respond to any interrupt requests until it has been re-initialised, i.e. it has been given a new pair of source and destination pointers. The activated ISR may perform the re-initialisation of the PEC.

10.4.1.6 Execution Times

In this section, architectural influences of the C164 microcontroller on the execution times are described.

Running at a 20MHz frequency the C164 microcontroller performs ten million machine cycles per second [Infineon, 1999, ch. 2]. In other words, the C164 performs a machine cycle for every 100ns. Most machine instructions are performed with one machine cycle. This gives up to ten million instructions per second (MIPS) for a C164 running at 20MHz.

It is possible for the C164 microcontroller to have both internal and external memory. In practice, internal memory has faster access times than external memory. In this case study no distinction is made between internal and external memory. Thus, internal memory speed is assumed. Furthermore, the C164 has no cache memory to make influence on the execution times.

The C164 has a four-stage instruction pipeline. The stages are described below:

Fetch An instruction is fetched from internal or external memory.

Decode The fetched instruction is decoded and required operands are fetched.

Execute The decoded operation is performed on the before fetched operands.

Write back The result is written back to the specified memory location.

Branch instructions require only one additional machine cycle when the branch is taken. Furthermore, most branches in loops require no additional machine cycles at all. Procedure calls require one additional machine cycle. The pipeline is primarily held by wait states for external memory accesses.

Table 10.1 shows execution times for a partial survey of machine instructions. The execution times are found by investigating [Infineon, 1999].

Operation	Execution time [ns]
Majority of operations	100
Multiplication of two words	500
Division of a double-word with a word	1 000
One PEC byte/word transfer	100
Initialisation of one PEC-channel	100
Starting/stopping the ADC	100
NPCS access to read/write two words	300
NPCS access to read/write sixteen words	2 100

Table 10.1: *Execution times for a 20MHz C164 microcontroller. Multiplication- and division operations may be delayed if an interrupt request occurs during the operation.*

Table 10.2 shows the execution times for the computations performed in the case study on the C164 microcontroller. These execution times are well qualified estimates by Critical ApS.

Computation	Execution time [μs]
One run of control algorithm, incl. filtering of inputs	5 000
Generation of three PWM patterns	40

Table 10.2: *Execution times for the computations performed by the control system.*

Table 10.3 on the next page shows other time factors which have influence on the scheduling analysis of the motor control system. The times for interrupt latencies are found in [Infineon, 1999] and the time for an ADC reading is from Critical ApS.

10.4.2 The Nucleus Plus Real-Time Operating System

The characteristics of the NP operating system, necessary for understanding the mapping of the logical design into the physical design, are summarised in this section.

Legend	Time [ns]
Typical interrupt latency	300
Maximum interrupt latency	500
Reading of an ADC-channel	100 000

Table 10.3: *Other time factors with influence on the motor control system.*

10.4.2.1 Low-Level Interrupt Service Routines

A low-level interrupt service routine (LISR) is started in a response to an interrupt request and is not scheduled by the operating system.

In NP a LISR has very limited access to the services of the operating system. The only possibility for communicating with NP tasks is using the NPCS protocol or to activate a HISR. It is not possible to release a semaphore from within a LISR. This must be done indirectly by activating a HISR, which then releases the semaphore.

10.4.2.2 High-Level Interrupt Service Routines

A HISR is activated by a LISR and is scheduled by the operating system.

Many of the NP system services are available for a HISR. A HISR must not be blocked by any system service. Thus, system services that are subject to blocking shall be called in a non-blocking way, e.g. if a HISR tries to obtain a semaphore, it must not be suspended if the semaphore cannot be obtained at the time.

10.4.2.3 Tasks

A NP task is scheduled by the operating system and have all system services at its disposal. A task in NP has always lower priority than a HISR.

10.4.2.4 Execution Times

This section presents a table of execution times for a selected set of NP system services. The execution times are found on basis of a performance test program, supplied from Accelerated Technology Incorporated (ATI). Furthermore, administrative overhead for LISRs, HISRs and tasks are quantitative estimates. The estimation is based on the authors experiences with NP.

10.5 Logical Design

This section documents the logical design of the motor control system, i.e. it presents all the design decisions that has been made independently of the execution environment. We shall not consider the initialisation phase of the system, i.e. we only consider a system which has successfully passed its initialisation phase.

Operation	WCET [μs]
Obtain a semaphore	55.1
Release a semaphore	55.1
Disable interrupts (NPCS)	2.6
Enable interrupts (NPCS)	2.6
Go to sleep	55.8
LISR administration†	5.0
Task and HISR administration†	50.0

Table 10.4: *Worst case execution times for selected NP system services and administrative operating system overhead. Execution times marked with a dagger are quantitative estimates.*

In the following section the partitioning of the motor control system is documented. In section 10.5.2 the system interface is identified. In section 10.5.3 the system is structured by arranging the events constituting the system interface event sequences. Finally, in section 10.5.4 the event sequences are transformed into abstract programs, and the resulting logical design is summarised by a structure diagram in subsection 10.5.4.4.

10.5.1 System Partition

In this section the motor control system is partitioned into subsystems. Being a traditional control system, as discussed in section 10.3, it is reasonable to partition the motor control system into the following three subsystems:

- Sensor
- Control
- Actuation

10.5.2 Interface Definition

The motor control system communicates with the environment through the peripherals: ADC, CAPCOM6, and RTC. By identifying the events between the control system and its environment the interface is documented.

Events are divided into two kinds: input and output. To indicate the type of information carried by an event, every event is associated with a type. In table 10.5 on the facing page the identified input events are presented. Table 10.6 on the next page presents the identified output events.

10.5.3 System Structuring

The events identified in section 10.5.2 are now arranged into *event sequences* to characterise the temporal behaviour of the system. Events that are temporally de-

Subsystem	Event name	Data type	Legend
Sensor	InvTemperature	number	Read inverter temperature
	InvVoltage	number	Read inverter voltage
	InvCurrent	number	Read inverter current
	RefVoltage	number	Read reference voltage
	Timer_1	unit	Periodic event
	Timer_2	unit	Periodic event
	Timer_3	unit	Periodic event
	Timer_4	unit	Periodic event
Control	Timer_5	unit	Periodic event
Actuation	CapComReady	unit	Periodic event; CAPCOM6 is ready for new PWM patterns

Table 10.5: *External input events for the motor control system.*

Subsystem	Event name	Data type	Legend
Actuation	FeedCapCom	number-triple	Feed the CAPCOM6 unit

Table 10.6: *External output events for the motor control system.*

pendent are sequenced together. In table 10.7 the result of this ordering is presented. The table reveals six event sequences of the motor control system.

Subsystem	Name	Event sequence
Sensor	DriverIT	(Timer_1; InvTemperature)*
	DriverIV	(Timer_2; InvVoltage)*
	DriverIC	(Timer_3; InvCurrent)*
	DriverRV	(Timer_4; RefVoltage)*
Control	Control	(Timer_5)*
Actuation	DriverCAPCOM6	(CapComReady; FeedCapCom)*

Table 10.7: *Event sequences containing only external events.*

The event sequences document the design decisions taken at this early stage of the design process. The behaviour of the subsystems of the motor control system is controlled by the synchronisation events identified in section 10.5.2. Thus, structuring the synchronisation events into event sequences restricts the possible behaviour of the motor control system.

The event sequences *DriverIT*, *DriverIV*, *DriverIC*, and *DriverRV* structures the data acquisition activities of the motor control system. The data acquisition is structured into four individual event sequences. The rationale for four individual sequences is that sufficient information for ordering the events into a smaller number of event sequences is not available. In each event sequence a periodic event is followed by an input event. Thus, the event sequences indicate that the individual data acquisition activities are performed periodically.

The *Control* event sequence documents the periodic execution of the control algorithm of the control system. The event sequence is simply the repeated sequence, which consists of a single periodic event. The *Control* event sequence is not associated with any data acquisition or actuation events.

The event sequence *DriverCAPCOM6* structures the actuation behaviour of the control system. The actuation is the periodic feeding of PWM patterns to the CAPCOM6 unit. Hence, the event sequence is the repeated sequence, which consists of the periodic event indicating the CAPCOM6 unit is ready for new PWM patterns followed by the event outputting the patterns to the CAPCOM6 unit.

10.5.4 Abstract Programs

The main purpose of this design step is to analyse the motor control system, currently structured by the event sequences. The product of the analysis shall be an identification of:

- The computations performed by the system.
- The data flow in the system.
- The data stores necessary to support the data flow. Data stores are either local to a process or shared between processes.

The findings of the analysis are documented by:

- A table presenting the identified computations.
- The event sequences augmented with the identified computations.
- A number of *abstract programs*.

In the following sections the analysis is performed, and its findings are documented.

10.5.4.1 Computations

Section 10.3 gave an overview of the computations performed in the motor control system. The computations are summarised by the three items below.

- Filtering of all four types of analog input data: the speed reference input, and the inverter temperature, voltage, and current.
- The control computation yields a voltage amplitude and frequency.
- The voltage amplitude and frequency are turned into compare values necessary for the generation of PWM signals in the actuation subsystem.

Subsystem	Computation	Legend
Control	FilterDataIT	Process raw inverter temp. data
	FilterDataIV	Process raw inverter voltage data
	FilterDataIC	Process raw inverter current data
	FilterDataRV	Process raw reference voltage data
	CalculateUF	Calculate new values of voltage amplitude and frequency
Actuation	CalculatePWM	Calculate the PWM patterns necessary for the generation of the PWM signals

Table 10.8: *Computations performed in the motor control system.*

In table 10.8 the computations are grouped according to the subsystems in which they are performed.

To further structure the behaviour of the system the previously formed event sequences are augmented by the identified computations. The resulting event sequences are shown in table 10.9.

It can be seen from the event sequence *Control* that the analog input values are filtered before they are used in the computation of the voltage amplitude and frequency.

The computation of PWM patterns used for the generation of the PWM signal have been placed in *DriverCAPCOM6*. This design decision separates the generation of PWM signals from the control computation of the motor control system.

Event Sequence	Event sequences with computations
DriverIT	(T_1; InvTemperature)*
DriverIV	(T_2; InvVoltage)*
DriverIC	(T_3; InvCurrent)*
DriverRV	(T_4; RefVoltage)*
Control	(T_5; (FilterRawDataIT ‖ FilterRawDataIV ‖ FilterRawDataIC ‖ FilterRawDataRV); CalculateUF)*
DriverCAPCOM6	(CalculatePWM; CapComReady; FeedCapCom)*

Table 10.9: *Event sequences augmented by the identified computations.*

10.5.4.2 Abstract Program Skeletons

For the purpose of constructing abstract programs, the event sequences are transformed into *abstract program skeletons*. As the syntax of a CSP-like abstract program resembles that of an event sequence, the translation into an abstract program skeleton is done by a number of simple transformations [Løvengreen, 1997].

The abstract program skeletons corresponding to the event sequences containing only input events are given below.

DriverIT $\stackrel{\text{def}}{=}$ **do** Timer_1?; InvTemperature? **od**
DriverIV $\stackrel{\text{def}}{=}$ **do** Timer_2?; InvVoltage? **od**
DriverIC $\stackrel{\text{def}}{=}$ **do** Timer_3?; InvCurrent? **od**
DriverRV $\stackrel{\text{def}}{=}$ **do** Timer_4?; RefVoltage? **od**

The transformation of the event sequence for the control computation results in the following program skeleton.

BasicMotorControl $\stackrel{\text{def}}{=}$
do
Timer_5?;
(FilterDataIT || FilterDataIV || FilterDataIC || FilterDataRV);
CalculateUF
od

Finally, the event sequence for the actuation is transformed into the skeleton below.

DriverCAPCOM6 $\stackrel{\text{def}}{=}$
do
CalculatePWM;
CapComReady?;
FeedCapCom!
od

10.5.4.3 Data Flow

Having transformed the event sequences into abstract program skeletons, we now look for data sources for the output data.

In general, the data needed by an output action in some process may have three sources [Løvengreen, 1997]:

- The data can be found as a function f of a recently received data value stored in a local variable x. In this case, the output takes the form $O!f(x)$.
- The data can be computed from data received previously by the process itself. In this case, a process-wide variable is introduced, updated in connection with the relevant events, and used in the output action.
- The data can be computed using data produced by other processes. In this case, a *state monitor* to hold relevant information produced by one or more other processes, is introduced. A state monitor must provide means for reading and writing such that data integrity is ensured. Thus, the processes given by abstract program skeletons shall not share variables.

To introduce the state monitors into the motor control system, we analyse the system in order to identify the data flow. Starting at an output event, we identify the data flow necessary for computing the associated output value. This is done by tracing the data flow backwards through the system until reaching the input events, that provide the input data for all the computations in the system.

The Voltage Amplitude and Frequency Monitor

In the motor control system data must flow to the CAPCOM6 unit. Thus, output data must be delivered to *FeedCapCom*. The output data is the PWM patterns needed by the CAPCOM6 unit to produce the PWM signal driving the inverter, and hence the motor. The calculation of the PWM patterns is computed by *CalculatePWM*. There are no input events in *DriverCAPCOM6*, thus the data necessary for computing the PWM patterns are not accessible to the process.

The PWM patterns are computed from the voltage amplitude and frequency data produced by *BasicMotorControl*. To provide *DriverCAPCOM6* with the data, a state monitor *MonitorUf* is introduced. The name 'Uf' is derived from the voltage amplitude (U) and frequency (f).

MonitorUf This monitor holds the voltage amplitude and frequency. The monitor provides the following operations:

putUf store voltage amplitude and frequency in the monitor.

getUf retrieve voltage amplitude and frequency from the monitor.

In an abstract program a monitor operation is implemented by a channel with an identical name. Thus, the invocation of a monitor operation is modelled by a communication on the particular channel. To ensure the channels have unique names, the monitor operations shall be prefixed with the monitor's name using a dotted notation, e.g. *MonitorUf.putUf*.

The Analog Input Monitors

The process *BasicMotorControl*, which is executing the system's control computation, has no external input events. Thus, the data necessary for computing the voltage amplitude and frequency data is inaccessible to the process.

The processes *DriverIT*, *DriverIV*, *DriverIC*, and *DriverRV* receive the necessary external input events from the ADC.

A state monitor is introduced for each driver process: *MonitorIT*, *MonitorIV*, *MonitorIC*, and *MonitorRV*. Each monitor has the same interface: an input operation and an output operation, hence we will only describe *MonitorIT*. The capacity of the monitor has been left underspecified in the logical design.

MonitorIT This monitor holds the raw inverter temperature data sampled by the ADC. The monitor has the following operations:

putValue inserts a value into the monitor.

getData obtains the data inserted into the monitor.

10.5.4.4 Structure Diagram

Figure 10.6 on the next page presents a structure diagram for the logical design of the motor control system. The structure diagram presents the internal structure of the

design: the processes, and the state monitors synchronising processes and supporting the data flow through the system. Thus, the structure diagram summarises the relationships between the components in the logical design.

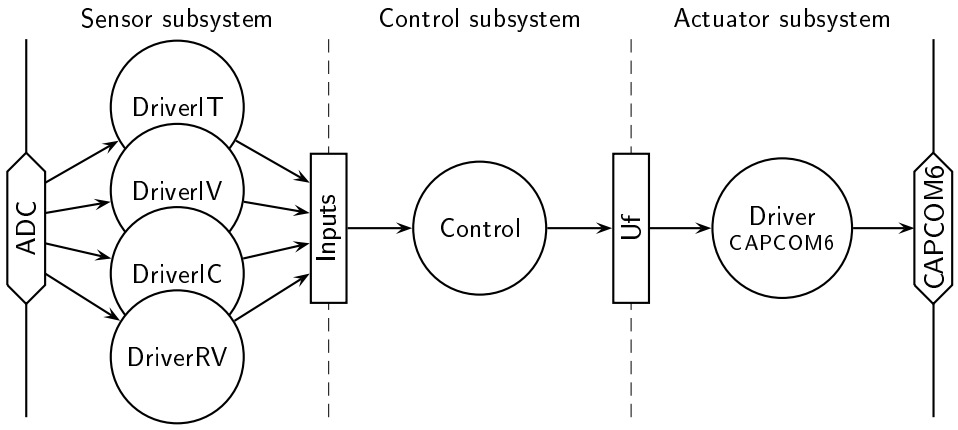

Figure 10.6: *Structure diagram for the motor control system.*

10.5.4.5 Abstract Programs

The findings of the data flow analysis is documented in a number of *abstract programs*. The programs are constructed from the abstract program skeletons defined in section 10.5.4.2.

A local variable has been identified in each of the four ADC driver processes, which sample the analog input data. The four processes in the sensor subsystem are analogous. Thus, we only show the abstract program for the *DriverIT* process.

```
DriverIT =def
    var it : number;
    do
      Timer1 ?;
      InvTemperature?it;
      MonitorIT.putValue!it
    od
```

The four monitors storing the analog input data in the sensor subsystem are analogous. Thus, we only show the abstract program for the *MonitorIT* process.

```
MonitorIT =def
    var it : number;
    var itl : number−list;
    do
      putValue?it → itl := it :: itl
      []
      getData!itl → itl := [ ]
    od
```

The abstract program for the control process is shown below. The functions, *FilterData*, *CalculatePWM*, and *CalculateUf*, are left underspecified.

```
Control ≝
    var it , iv , ic , rv : number−list;
    var fit , fiv , fic , frv : number;
    var Uf : number × number;
    do
      Timer5 ? ;
      (
        MonitorIT.getData ? it; fit := FilterDataIT(it)   ||
        MonitorIV.getData ? iv; fiv := FilterDataIV(iv)   ||
        MonitorIC.getData ? ic; fic := FilterDataIC(ic)   ||
        MonitorRV.getData ? rv; frv := FilterDataRV(rv)
      );
      Uf := CalculateUf(fit , fiv , fic , frv );
      MonitorUf.putUf ! Uf
    od
```

The voltage amplitude and frequency monitor:

```
MonitorUf ≝
    var Uf : number × number;
    do
      putUf ? Uf → skip [] getUf ! Uf → skip
    od
```

The actuator process:

```
DriverCAPCOM6 ≝
    var Uf : number × number;
    var PWM : number × number × number;
    do
      MonitorUf.getUf ? Uf;
      PWM := CalculatePWM(Uf);
      CapComReady ? ;
      FeedCapCom ! PWM
    od
```

10.6 Physical Design for Sensor Subsystem

The sensor subsystem must control the ADC and pass inputs to the control subsystem. An implementation of the sensor subsystem should:

- Every time the control subsystem is about to run, provide new inputs for this.
- Produce a minimum overhead.

In this section a modified logical design for the sensor subsystem is presented. The rationale for the modified logical design is that it is better suited for the mapping onto the execution environment. Given the modified logical design, a general mapping onto the execution environment is discussed. Finally, two specific implementation alternatives for the physical design are suggested.

10.6.1 Modified Logical Design

In the logical design presented in section 10.5, it is assumed that individual ADC channels can be read in parallel. However, the execution environment does not facilitate this. The ADC supports up to eight channels, but it is only possible to read a value on one channel at a time, cf. section 10.4.1.1. This dependency among the input events in the actual execution environment was not discovered in the development of the initial logical design.

The logical design is modified by arranging the input events of the four event sequences *DriverIT*, *DriverIV*, *DriverIC*, and *DriverRV* into a single event sequence *DriverADC*. From the event sequence a program skeleton and an abstract program are developed. Thus, the four input drivers of the logical design are sequentially composed into a single driver *DriverADC*. The merging of the four drivers into one driver eliminates the four input events, $Timer_1$, $Timer_2$, $Timer_3$, and $Timer_4$, but introduces another input event $Timer_6$ which starts all the readings. *DriverADC* must perform a total of sixteen readings, four readings of each input, cf. section 10.3. When all values are read, which is expressed by the internal choice in the innermost loop, the sixteen values are transfered to the monitor.

Given a single driver process, it is no longer necessary, that the interface between the sensor and the control subsystem consists of four monitors. The four state monitors are combined into one, *MonitorInput*, which defines a new interface between the sensor and the control subsystem.

The abstract programs for the new process *DriverADC* and the new state monitor *MonitorInput* are given below:

```
DriverADC def=
    var it , iv , ic , rv : number;
    var values : number−list;
    do
      Timer6 ? ;
      values := [ ];

      do
        InvTemperature? it;
        InvVoltage? iv;
        InvCurrent? ic;
        RefVoltage? rv;
        values := it :: iv :: ic :: rv :: values;
       ( skip [] exit)
      od

      MonitorInput.putValues! (values)
    od


MonitorInput def=
    var values : number−list;
    do
      putValues? values → skip [] getData! values → skip
    od
```

10.6.2 Physical Design

The ADC shall sequentially read four channels. To filter the input values as discussed in section 10.5.4.1, each channel are read four times before there are enough readings to be passed to the control subsystem.

In this case study, a reading from an ADC channel is estimated to be accomplished in 100μs. Thus, it takes 1.6ms to perform sixteen readings if the ADC is operated in *auto scan continuous mode*, cf. section 10.4.1.1. In this mode the ADC repeatedly reads a set of channels until it becomes disabled. When a value becomes ready the ADC unit generates an interrupt.

An implementation of the subsystem must setup the ADC to start the readings. Furthermore, when the ADC has produced a new reading, the subsystem must transfer this from the ADC register to another memory location before yet another reading is produced. Using the PEC to transfer the readings is the solution with least overhead. When sixteen readings are completed the ADC must be stopped.

It is important that the readings are temporally ordered when they are passed on to the control subsystem. The control subsystem must know in which order the readings came in order to filter them correctly. One way to do this is to synchronise the two subsystems properly.

10.6.3 First Implementation Alternative of Sensor Subsystem

When developing the physical architecture, new internal events are introduced in order to model the operation of the execution environment. The internal events model the additional synchronisation found in the physical architecture. Table 10.10 summarises the necessary internal events.

Subsystem	Event name	Legend
Sensor	StartADC	Start the ADC in auto scan continuous mode
	StopADC	Stop the ADC unit
	ValueReady	A new value is ready in the ADC register
	InitADCPEC	Initialise a PEC channel to service the ADC interrupt
	StartADCPECLISR	When the PEC becomes idle the *ADCPECLISR* is activated
	putValues	Insert the input values into *MonitorInput*
	getData	Read the input data from *MonitorInput*

Table 10.10: *Internal subsystem events modelling internal synchronisation and behaviour of execution environment.*

Considering the original input events for the sensor subsystem in table 10.5 on page 83 and the new internal events in table 10.10, three event sequences were formed: *DriverADC*, *ADCPEC*, and *ADCPECLISR*. Then, program skeletons were constructed, and abstract programs derived.

A periodic NP task shall implement *DriverADC*. The task starts the ADC in auto

scan continuous mode. In addition, the task initialises a PEC channel, implementing the process *ADCPEC*, to transfer sixteen readings to a local memory location.

```
DriverADC =def
  do
    Timer6;
    InitADCPEC![ ];
    StartADC!
  od

ADCPEC =def
  var value : number;
  var values : number−list;
  do
    InitADCPEC?values;
    do
      ValueReady?value;
      values := value :: values;
    ( skip [] exit)
    od
    StartADCPECLISR!values
  od
```

When *ADCPEC* has transferred all the readings, which is expressed by the internal choice in the innermost loop, the PEC service is disabled and a LISR, implementing *ADCPECLISR*, is started, cf. section 10.4.1.5. The *ADCPECLISR* stops the ADC and then transfers all the readings from the local memory location to *MonitorInput*, which is the critical section shared between the sensor- and control subsystem. The critical section is protected by the NPCS.

```
ADCPECLISR =def
  var values : number−list;
  do
    StartADCPECLISR?values;
    MonitorInput.putValues!values;
    StopADC!
  od
```

10.6.3.1 Constructing a Situation Table

As explained in section 2.8 a situation table may be used for the representing a real-time design. In this section we illustrate how the periods, execution times, and relative deadlines presented in the situation tables of every subsystem are found.

We will only give a rigorous explanation of the construction of a situation table for the first implementation alternative of sensor subsystem. For the remaining implementation alternatives we only indicate, unless special problems occurred, how the execution time estimates and deadlines were found when forming the situation table.

Periods

In section 10.2 it was required that the readings of the inputs should be repeated with a period of ten milliseconds. From this requirement we can derive that the

sensor subsystem has a period of ten milliseconds.

Execution Times

Based on the found and/or estimated execution times derived of the execution environment, presented in section 10.4.1.6 and section 10.4.2.4, we find execution times for the abstract programs.

When mapping the *DriverADC* abstract program onto the execution environment it is implemented as a NP task. Table 10.4 shows that a NP task is estimated to have an administrative overhead of 50μs. The task must start the ADC and release the *ADCPEC* task. In the presented execution environment this can be achieved in $(100 + 100)$ns $= 200$ns. Thus, the total execution time is $50\mu\text{s} + 200\text{ns} = 50.2\mu$s.

When mapping the *ADCPEC* abstract program onto the execution environment it is implemented as sixteen PEC operations. Table 10.1 shows that a PEC operation is performed in 100ns of which the execution time is derived.

When mapping the *ADCPECLISR* abstract program onto the execution environment it is implemented as a LISR. Table 10.4 shows that a LISR is estimated to have an administrative overhead of 5μs. Furthermore, this task is responsible of transferring the sixteen readings to the input monitor. Using the simplest and fastest implementation of NPCS this can be done in 2.1μs, cf. table 10.1. Finally, the ADC must be stopped. This operation takes 100ns. Thus, the total execution time is $(5 + 2.1 + 0.1)\mu\text{s} = 7.2\mu$s.

Deadlines

In section 10.2 it was required that all the readings should be completed within ten milliseconds after the first reading was started.

At a first glance we do not have enough information to fix a deadline for the *DriverADC* task. We can only say that it must be less than ten milliseconds. We return to this deadline when we have been through the rest of the subsystem.

Table 10.3 shows that the ADC will produce a reading every 100μs, i.e. the ADC result register must be read within 100μs before yet another reading is produced. This gives a deadline for each of the sixteen PECs of 100μs.

The LISR must stop the ADC within 100μs after the ADC has produced the sixteen readings. This must be done before yet another reading is completed. The LISR is released at the same time the last PEC service is started. This gives a deadline of the *ADCPECLISR* of 100μs.

We now return to the deadline of the *DriverADC* task. The whole operation, of one run of the sensor subsystem, must be completed within ten milliseconds. We can see that the sixteen readings is completed in $16 \cdot 100\mu\text{s} = 1.6$ms. Together with the deadline of the *ADCPECLISR* we find that once the ADC is started it takes $1.6\text{ms} + 100\mu\text{s} = 1.7$ms, before all the sixteen readings are transfered to the input monitor. Thus, in order to meet the overall deadline of the subsystem the deadline for the *DriverADC* task must be $(10 - 1.7)\text{ms} = 8.3$ms.

In table 10.11 on the following page the derived situation table is presented. The situation table captures the real-time design of the first implementation alternative

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]
ADCPEC [1–16]	PEC	10.0000	0.0001	0.1000
ADCPECLISR	LISR	10.0000	0.0072	0.1000
DriverADC	Task	10.0000	0.0502	8.3000

Table 10.11: *Situation table for sensor subsystem; first implementation alternative. The 'type' column indicates mapping onto execution environment.*

of the sensor subsystem. The table contains the initial execution times estimates for the tasks of the subsystem. In the subsequent phases of the development process the execution times are refined as the implementation of the individual tasks progresses.

For example, the execution time for the computation of three PWM patterns are estimates based on previous experience. As the implementation is carried out better estimates may be provided. When the final implementation is complete the exact execution time is measured.

Given the situation tables for all subsystems we may analyse a complete design. In section 10.9 we analyse different designs of the motor control system.

10.6.4 Second Implementation Alternative for Sensor Subsystem

The second implementation alternative differs from the first in that we take advantage of the timing behaviour of the system. The idea is illustrated by the example in section 5.5. If the sensor subsystem has the same period as the control subsystem and the two subsystems are displaced in their phases, it can be arranged that the two subsystems will not access the critical section at the same time. Thus, no protection mechanism is necessary to make sure all sixteen values are written before any of them are read.

The internal events introduced in the first implementation alternative, cf. table 10.10, are identical to the second implementation alternative, except the event *putValue* which differs from the internal events of the first implementation alternative.

As in the previous implementation a periodic NP task implements *DriverADC*, i.e. starts the ADC in auto scan continuous mode and initialises the PEC channel, which implements *ADCPEC*. Now the PEC transfer the sixteen readings directly into the critical section with no protection all. As before, when the PEC has transferred all the readings a LISR, *ADCPECLISR* is started. The LISR stops the continuous mode of the ADC.

The second implementation alternative gives rise to different abstract programs for *MonitorInput*, *ADCPEC*, and *ADCPECLISR*:

```
MonitorInput ≝
    var value : number;
    var values : number−list;
    do
      do putValues ? value → values := value :: values od
      ||
      do getData ! values → values := [ ] od
```

```
    od

ADCPEC ≝
  var value : number;
  do
    InitADCPEC?;
    do
      ValueReady? value;
      MonitorInput.putValue! value;
    ( skip [] exit)
    od
    StartADCPECLISR!
  od

ADCPECLISR ≝
  do
    StartADCPECLISR?;
    StopADC!
  od
```

10.6.4.1 Situation Table

The idea behind the second implementation alternative is to ensure mutual exclusion by assigning the tasks of the sensor subsystem a phase different from the phase of the control system, cf. section 5.5. We assign the sensor subsystem a phase of zero milliseconds and the control subsystem a phase of two milliseconds. Both subsystems share the same period of ten milliseconds.

We effectively place a precedence constraint on the control subsystem, which shall execute after the sensor subsystem. Thus, the sensor subsystem must complete within the first two milliseconds of its period, while the control subsystem is allowed to execute for the remaining eight milliseconds.

Due to the phasing the tasks of the two subsystems will never share a critical instant.

The deadline of the task *DriverADC* of the sensor subsystem must be reduced according to the phasing of the subsystem, $(2 - 1.7)\text{ms} = 0.3\text{ms}$.

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]
ADCPEC [1–16]	PEC	10.0000	0.0001	0.1000
ADCPECLISR	LISR	10.0000	0.0051	0.1000
DriverADC	Task	10.0000	0.0502	0.3000

Table 10.12: *Situation table for sensor subsystem; second implementation alternative.*

10.7 Physical Design for Control Subsystem

The control subsystem is implemented as a periodic NP task. The task shall first retrieve and filter the sixteen readings from the sensor subsystem. The task must

then compute a new pair of voltage amplitude and frequency values with the control algorithm. The two values must be available for the actuation subsystem which contains the phase generator algorithm. The two values shall be written to a critical section using a NPCS.

```
Control ≝
    var values : number−list;
    var Uf : number × number;
    do
      Timer5 ? ;
      MonitorADC.getData ? values;

      Uf  = CalculateUf(FilterData(values));

      MonitorUf.putUf ? Uf;
    od
```

10.7.1 Situation Table

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]
Control	Task	10.0000	5.0524	10.000

Table 10.13: *Situation table for control subsystem.*

The above situation table is based on the configuration where the control system is coupled with the first implementation alternative of the sensor subsystem. The execution time for the *Control* task is based on the control algorithm execution estimate time of 5ms, cf. table 10.2, the NPCS access of both monitors estimated to $(2.1 + 0.3)\mu s = 2.4\mu s$, cf. table 10.1, and finally the estimated administration task overhead of $50\mu s$, cf. table 10.4. The total estimated execution time is found to be $5ms + (2.4 + 50)\mu s = 5.0524ms$ as stated in the situation table.

If the control subsystem is coupled with the second implementation alternative the deadline of the *Control* task must be reduced to eight milliseconds and the phase shall be two milliseconds, as discussed in section 10.6.4.1.

10.8 Physical Design for Actuation Subsystem

We now turn our attention to the actuation subsystem. We shall consider two different implementation alternatives, exploring the effects of using different features of the execution environment in the development of the physical architecture.

10.8.1 First Implementation Alternative for Actuation Subsystem

The first implementation alternative for the actuation subsystem does not imply any modifications to the logical architecture. Hence, a PWM pattern for each phase

is calculated every time the CAPCOM6 unit is fed. The structure diagram in figure 10.6 is still valid for the first implementation alternative of the actuation alternative.

The interface between the control and the actuation subsystem is *MonitorUf*, which provides the operations *putUf* and *getUf* for storing and obtaining a voltage amplitude and frequency pair. Mutual exclusive access to the monitor's data is achieved by NPCS. The abstract program for the monitor is identical to the one given in the logical design.

As explained in section 10.4.1.3 the calculation of the PWM patterns used for the generation of PWM signals with the CAPCOM6 unit can be done in response to an interrupt. This approach is taken in the first implementation alternative for the actuation subsystem.

Thus, *DriverCAPCOM6* is implemented by a LISR. The use of NPCS in *MonitorUf* ensures mutual exclusion between *BasicMotorControl* and the LISR. When the CAPCOM6 unit generates an interrupt, the LISR obtains the current voltage amplitude and frequency values from the monitor, computes the PWM patterns for the three phases, and writes the patterns to the shadow latches of the CAPCOM6 unit.

10.8.1.1 Situation Table

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]
DriverCAPCOM6	LISR	0.1000	0.0456	0.1000

Table 10.14: *Situation table for actuation subsystem; first implementation alternative.*

The period is derived from the required switching frequency, cf. section 10.2. Jitter is not allowed on the output operation. Using the shadow latches this jitter requirement is meet, cf. section 10.4.1.3. After the PWM patterns are transfered from the shadow latches to the CAPCOM6 unit new patterns must be supplied within 100 microseconds.

The estimated execution time for the *DriverCAPCOM6* task is based on the estimated execution time of 40μs for generating the PWM pattern triple, cf. table 10.2, the estimated LISR administration overhead of 5μs, cf. table 10.4, the NPCS access of the *MonitorUf* of 300ns, and finally the execution time required to write the PWM patterns into to the registers of the CAPCOM6 unit of 300ns. The total estimated execution time is found to be $(40 + 5)\mu\text{s} + 2 \cdot 300\text{ns} = 45.6\mu\text{s}$ as stated in the situation table.

10.8.2 Second Implementation Alternative for Actuation Subsystem

The second implementation alternative introduces a modified logical design. The modified logical design is presented here, as it is an intermediate step in the process of developing the physical design. The rationale for the modified logical design is

that it is better suited for deriving a physical design that utilises the PEC of the C164CI microcontroller; a special hardware feature of the execution environment.

10.8.2.1 Modified Logical Design

The calculation of PWM patterns is separated from the task of driving the hardware device. Thus, *DriverCAPCOM6* shall no longer be concerned with the calculation of PWM patterns for the CAPCOM6 unit. The decomposition indicates that too many events have been ordered into one event sequence in the initial logical design, in order to support all possible physical designs.

In this alternative *DriverCAPCOM6* shall only be concerned with moving the computed PWM patterns into the CAPCOM6 unit generating the PWM signals. A new process *PhaseGenerator* shall compute the PWM patterns.

The logical architecture is modified by rearranging the events of the event sequence *DriverCAPCOM6* into two new sequences. Then, these are translated into abstract program skeletons, and the data flow of the new architecture is analysed. The result is two abstract programs *DriverCAPCOM6* and *PhaseGenerator* and an additional state monitor *MonitorPWM*.

From the voltage amplitude and frequency data stored in *MonitorUf*, *PhaseGenerator* calculates the PWM patterns use in the PWM signal generation.

```
PhaseGenerator ≝
    var Uf : number × number;
    var PWM : number × number × number;
    do
      MonitorUf.getUf ? Uf;
      PWM := CalculatePWM(Uf);
      MonitorPWM.putPWM ! PWM
    od
```

In the state monitor *MonitorPWM* the PWM patterns are stored in a queue. The internal choice in the abstract program for *MonitorPWM* models the necessary blocking enforced by the monitor operations. *putPWM* shall block if the queue is full, and *getPWM* shall block if the queue is empty. Hence, in the modified logical design the capacity of the queue is left underspecified. The abstract program for the monitor is given below.

```
MonitorPWM ≝
    var PWM : (number × number × number);
    var FIFO : (number × number × number)−list;
    do
      putPWM ? PWM → FIFO := FIFO @ [PWM]
      []
      getPWM ! hd(FIFO) → FIFO := tl(FIFO)
    od
```

The process *DriverCAPCOM6* feeds the PWM patterns stored in the *MonitorPWM* to the CAPCOM6 unit.

```
DriverCAPCOM6 ≝
    var PWM : number × number × number;
```

```
do
  MonitorPWM.getPWM ? PWM;
  CapComReady ? ;
  FeedCapCom ! PWM
od
```

10.8.2.2 Physical Design

In the development of the physical architecture the modified logical architecture is modified even further. These modifications as well as the main points in the development of the physical architecture are summarised below.

- New internal events modelling the operation of the execution environment are introduced.
- Mutual exclusive access to the data stored in *MonitorUf* is enforced by NPCS.
- The monitor *MonitorPWM* is implemented as a double buffer data structure. The monitor shall synchronise the execution of the processes accessing the buffers, and in particular ensure mutual exclusive access to each buffer.
- *DriverCAPCOM6* is split into three processes, when mapping the logical architecture onto the execution environment.
  - A process *DriverCAPCOM6*. The process moves PWM patterns from a buffer into the CAPCOM6 unit. The process is implemented by three PEC channels, one for each phase of the inverter.
  - A process *PECLISR* is implemented by a LISR. The process is started when *DriverCAPCOM6* moves the last three PWM patterns in a buffer into the CAPCOM6 unit, thus when the PEC service is disabled
  - A process *PECHISR* implemented by a HISR. The process is started by *PECLISR* and is responsible for:
    - Obtaining a new buffer of PWM patterns from *MonitorPWM* to be processed by *DriverCAPCOM6*.
    - The re-initialisation of the PEC service, hence activating the PEC service.
    - Returning the buffer processed by *DriverCAPCOM6* to *MonitorPWM*.
- The process *PhaseGenerator* is implemented as a NP task. The process obtains an empty buffer from *MonitorPWM* and fills it with PWM patterns.

The internal events introduced to model the additional synchronisation found in the physical architecture are listed in table 10.15 on the next page. Due to the complexity of the second implementation alternative of the actuation subsystem we show a structure diagram in figure 10.7 on the following page.

Data Flow

Though the identification of state monitors is a result of the data flow analysis and is performed late in the design process, the presentation of the physical design begins with a discussion of the identified state monitors.

Subsystem	Event name	Legend
Actuation	InitPEC	Initialise the PEC
	StartPECLISR	When the PEC becomes idle the *PECLISR* is activated
	StartPECHISR	The *PECLISR* activates the *PECHISR*
	getEmptyBuffer	Partially implements logical event putPWM
	putEmptyBuffer	Partially implements logical event getPWM
	getFullBuffer	Partially implements logical event getPWM
	putFullBuffer	Partially implements logical event putPWM

Table 10.15: *Internal events for the actuation subsystem; second implementation alternative.*

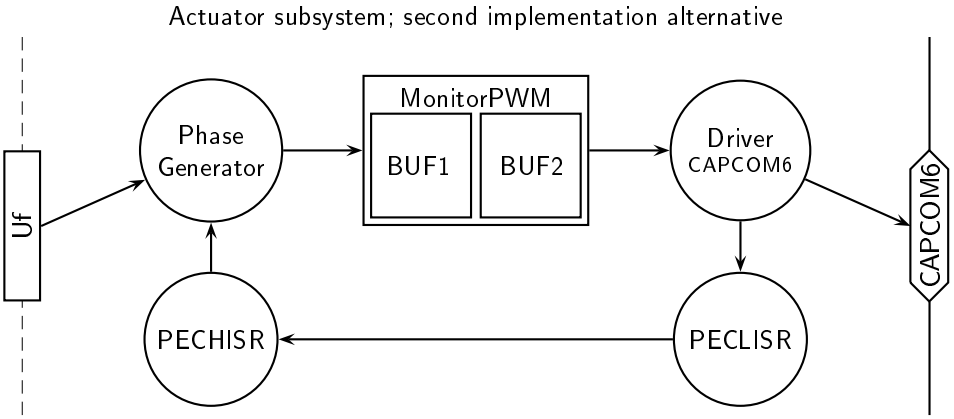

Figure 10.7: *Structure diagram for actuation subsystem; second implementation alternative.*

A new state monitor, *MonitorPWM*, shall hold the PWM patterns computed by *PhaseGenerator* until they are fed to the CAPCOM6 unit by *DriverCAPCOM6*. The implementation of the monitor uses a concrete data structure consisting of two equally sized buffers. A buffer element is a triple of PWM patterns, a pattern for each of the three phases driving the motor.

The *PhaseGenerator* and the *DriverCAPCOM6* has a simple producer/consumer relationship. *PhaseGenerator* produces one buffer of PWM patterns, while *DriverCAPCOM6* consumes the contents of the other, i.e. the PWM patterns of the buffer is fed into the CAPCOM6 unit.

The following terminology is used in the discussion of the monitor's operations: a buffer is said to be full if none of the contained PWM patterns have been fed to the CAPCOM6 unit. A buffer is said to be empty, if all the PWM patterns of a buffer have been fed to the CAPCOM6 unit. The monitor provides four operations:

getFullBuffer	allows *DriverCAPCOM6* to obtain a full buffer of PWM patterns. The operation blocks if no full buffer is available.
putFullBuffer	stores a full buffer of new PWM patterns computed by *PhaseGenerator* in the monitor. The operation blocks if the monitor already holds a non-empty buffer.
getEmptyBuffer	allows *PhaseGenerator* to obtain an empty buffer. The operation blocks if no empty buffer is available.
putEmptyBuffer	an empty buffer is returned to the monitor by *DriverCAPCOM6*. The operation will block if the monitor already holds an empty buffer.

An abstract program for *MonitorPWM* is given below. The abstract program specifies that, initially, a buffer must be filled before a full buffer can be obtained. Afterwards, one buffer can be filled while the other buffer is being emptied.

```
MonitorPWM =def
    var FullBuffer : ( number × number × number)−list;
    var EmptyBuffer : (number × number × number)−list;
    do
       getEmptyBuffer ! EmptyBuffer;

       do putFullBuffer ? FullBuffer ; getFullBuffer ? FullBuffer od
       ||
       do putEmptyBuffer ? EmptyBuffer ; getEmptyBuffer ? EmptyBuffer od
    od
```

Abstract Programs

For the remaining abstract programs of the subsystem, event sequences were formed, considering the new internal events in table 10.15 and the events in the original event sequence for *DriverCAPCOM6*, see table 10.9 on page 85. Then, program skeletons were constructed, and abstract programs derived.

As already discussed *PhaseGenerator* acquires an empty buffer. After the buffer has been filled with PWM patterns, it is stored in *MonitorPWM*.

```
PhaseGenerator def=
    var Uf : number × number;
    var Buffer : (number × number × number)−list;
    do
      MonitorPWM.getEmptyBuffer ? Buffer;
      MonitorUf.getUf ? Uf;
      Buffer := CalculatePWM(Uf);
      MonitorPWM.putFullBuffer ! Buffer
    od
```

The process *DriverCAPCOM6* of the modified logical design is replaced by the three processes: *DriverCAPCOM6*, *PECLISR*, and *PECHISR* in the physical design. How the processes map onto the execution environment is explained below.

The process *DriverCAPCOM6* of the physical design is mapped onto three PEC channels, cf. section 10.4.1.5. Given a buffer of PWM patterns the process feeds patterns to the CAPCOM6 unit. While the buffer of PWM patterns is not empty, each PEC channel moves a single PWM pattern into the shadow latches of the CAPCOM6 unit in response to the interrupt *CapComReady* generated by the CAPCOM6 unit, cf. section 10.4.1.3.

When feeding the last pattern in the buffer to the CAPCOM6 unit, detected by the internal choice of the innermost loop, the process exits the innermost loop and starts the process *PECLISR*. The PEC service is disabled until it is reinitialised, i.e. given a new buffer.

```
DriverCAPCOM6 def=
    var Buffer : (number × number × number)−list;
    do
      InitPEC ? Buffer

      do
        CapComReady ? ;
        FeedCapCom ! hd(Buffer);
        Buffer := tl(Buffer);
       ( skip  ∏  exit)
      od

      StartPECLISR ! Buffer;
    od
```

In the Nucleus Plus operating system a LISR responds to an interrupt request. Since a LISR can only perform very simple system calls, cf. section 10.4.2.1, a HISR must be started with the purpose of switching buffers and reinitialising the PECs.

```
PECLISR def=
    var EmptyBuffer : (number × number × number)−list;
    do
      StartPECLISR ? EmptyBuffer;
      StartPECHISR ! EmptyBuffer
    od
```

When *PECHISR* is started by *PECLISR* it returns the empty buffer to *MonitorPWM* and requests a full one. The PECs are then reinitialised with the full buffer, thus reactivating the PEC service.

```
PECHISR ≝
    var EmptyBuffer : (number × number × number)−list;
    var FullBuffer : ( number × number × number)−list;
    do
      StartPECHISR ? EmptyBuffer;
      MonitorPWM.getFullBuffer ? FullBuffer;
      InitPEC ! FullBuffer;
      MonitorPWM.putEmptyBuffer ! EmptyBuffer
    od
```

PWM patterns shall be continuously fed to the CAPCOM6 unit at the rate of the switching frequency. Thus, it is a real-time requirement that *PhaseGenerator* has already delivered a new buffer of PWM patterns when *DriverCAPCOM6* requests a new buffer of PWM patterns. With a switching frequency of 10kHz, cf. section 10.2, a buffer must hold 100 triples of PWM patterns.

10.8.2.3 Situation Table

The real-time design of the second implementation alternative for the actuation subsystem is summarised in situation table 10.16.

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]
PhaseGenerator	Task	10.0000	4.1054	9.9000
DriverCAPCOM6	PEC	0.1000	0.0003	0.1000
PECLISR	LISR	10.0000	0.0050	0.1000
PECHISR	HISR	10.0000	0.1057	0.1000

Table 10.16: *The situation table for actuation subsystem; second implementation alternative. The estimated execution time for* PECHISR *exceeds its deadline.*

Execution times

For each process in the physical design we will briefly introduce the estimated execution times, based on the estimates in the tables 10.1, 10.2, and 10.4.

The administrative overhead of releasing a NP task is 50μs. The current values for the voltage amplitude and frequency must be retrieved, at the cost of 0.3μs. The *PhaseGenerator* must produce a buffer of PWM pattern triples. The execution time for the calculation of one PWM pattern triple is estimated to be 40μs according to table 10.2. Thus, a buffer 100 triples can be calculated in 4 000μs. When the buffer has been filled it is returned to the *MonitorPWM* causing the *PhaseGenerator* to suspend on a semaphore at the cost of 55.1μs. Thus, the total execution time estimate for the *PhaseGenerator* is 4,105.4μs.

DriverCAPCOM6 is implemented by three PEC channels servicing the *CapComReady* interrupt. Thus, the estimated execution time is simply three times the execution time for a single PEC, i.e. 0.3μs by table 10.1.

The execution time estimate for the *PECLISR* is given by the estimated administrative overhead introduced by a LISR. It is the responsibility of the *PECLISR* to

activate the *PECHISR*. However, the cost in execution time for activating a HISR is included in the administrative overhead estimate for the *PECHISR* and will be paid by *PECHISR*. Thus, by table 10.4 the estimated execution time for *PECLISR* is 5μs.

For the *PECHISR* the administrative overhead is estimated to be 50μs. In table 10.4 the execution time for the operation of releasing the semaphore blocking *PhaseGenerator* is estimated to 55.1μs. The PWM pattern buffer switch can be performed by only six instructions having an execution time of 0.6μs. Thus the total estimate for the *PECHISR* is 105.7μs.

Deadlines

With a switching frequency of 10kHz *DriverCAPCOM6* must have a deadline of 100μs in order to service the CAPCOM6 unit. When all PWM patterns of the buffer has been fed to the CAPCOM6 unit a new buffer must be acquired and the three PECs must be reinitialised within the same deadline of 100μs.

To summarise the current design, the operation of feeding the last triple of PWM patterns to the CAPCOM6 unit generates the *CapComReady* interrupt, which activates the *PECLISR*. The *PECLISR* activates the *PECHISR*. It is the job of the *PECHISR* to switch PWM pattern buffers, reinitialise the three PECs, and release the *PhaseGenerator*.

Thus, from the generation of the interrupt the *PECLISR* and *PECHISR* must complete within 100μs. However, the estimated execution time for the *PECHISR* is 105.7μs, hence the deadline of 100μs can never be met.

10.8.2.4 Revised Physical Design

The physical design for the second implementation alternative of the actuation subsystem must be revised before it can meet its deadlines. The re-initialisation of the three PECs implementing *DriverCAPCOM6* can meet its deadline of 100μs if it is placed in in the *PECLISR* rather than the *PECHISR*.

Thus, the *PECLISR* shall obtain a new buffer from *MonitorPWM*, reinitialise the three PECs, and the finally activate the *PECHISR*.

```
PECLISR ≝
    var EmptyBuffer : (number × number × number)−list;
    var FullBuffer : ( number × number × number)−list;
    do
      StartPECLISR ? EmptyBuffer;
      MonitorPWM.getFullBuffer ? FullBuffer;
      InitPEC ! FullBuffer;
      StartPECHISR ! EmptyBuffer
    od
```

When *PECHISR* is started by *PECLISR* it simply returns the empty buffer to *MonitorPWM*. This will release the *PhaseGenerator*, which in turn fills the empty buffer.

```
PECHISR ≝
    var EmptyBuffer : (number × number × number)−list;
```

```
do
  StartPECHISR ? EmptyBuffer;
  MonitorPWM.putEmptyBuffer ! EmptyBuffer
od
```

10.8.2.5 Situation Table

The real-time design for the revised physical design of the second implementation alternative of the actuation subsystem is summarised in situation table 10.17.

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]
DriverCAPCOM6	PEC	0.1000	0.0001	0.1000
PECLISR	LISR	10.0000	0.0056	0.1000
PECHISR	HISR	10.0000	0.1051	1.0000
PhaseGenerator	Task	10.0000	4.1051	9.0000

Table 10.17: *Situation table for actuation subsystem; second implementation alternative.*

Given the revised design, the execution time estimate for the *PECLISR* is increased to 56μs and the execution time estimate of the *PECHISR* is reduced to 105.1μs.

In the revised physical design the *PECLISR* obtains a new buffer of PWM patterns and reinitialises the three PECs, hence its deadline must be 100μs. The deadline of 100μs is no longer required for the *PECHISR*, thus we relax the deadline to 1 000μs. This requires the *PhaseGenerator* to complete before a deadline of 9 000μs.

10.9 Schedulability Analysis

The initial iteration of the logical and physical design phases of the development process is now complete. In the logical design phase we decomposed the system into three subsystems by identifying two general interfaces between the subsystems. The logical design phase was completed by the definition of abstract programs for the three subsystems. In the physical design phase, the abstract programs were mapped onto the execution environment. Having successfully defined general subsystem interfaces in the logical design phase, it was possible to construct two different physical design alternatives for the sensor and actuation subsystems. For each implementation alternative its real-time design was summarised in a situation table.

In this section we will investigate different implementations of the motor control system by combining different subsystem implementation alternatives. We investigate two implementations based on the combination of the first implementation alternative for the sensor subsystem, the control subsystem implementation, and the two implementation alternatives for the actuation subsystem.

At this early time in the development process we will investigate the following aspects of the two motor control system implementations:

- test the feasibility

- calculate the total utilisation U
- estimate the spare capacity with respect to increasing the switching frequency while maintaining the current system responsiveness

Finally, for the motor control system implementation based on the first implementation alternative for the actuation subsystem, we will predict the reduction in system responsiveness necessary to obtain a switching frequency of 12.5kHz.

10.9.1 Tool Support

We will use a tool, when performing the feasibility tests in the following sections. The tool implements the feasibility test described in section 5.6, hence it supports arbitrary deadlines and blocking, but it does not support arbitrary phasing. The tool 'Response Time' by Stéphane Decleire is found on a CD-ROM bundled with [Briand and Roy, 1999]. The tool only supports the feasibility test, hence it does not provide support for priority assignment or blocking time computation.

Unfortunately, it has not been possible to obtain a tool, which implements Audsley's priority assignment algorithm and feasibility test, which supports deadlines less than or equal to the period and arbitrary phasing. Given such a tool a necessary and sufficient feasibility test would have been available for the motor control system.

10.9.1.1 Implication of Tool Limitations

The limitations of the tool imply that precedence constraints are ignored. Thus, we only consider the worst-case situation of a critical instant.

For example, for the sensor subsystem we will ignore the different phases of the sixteen $ADCPEC$ tasks. Instead, we will investigate the worst-case situation of a critical instant, though it will never occur.

The rationale for this approach is discussion found in section 4.2.3. If for every task T_i, in a task set $\mathbf{T}$, a job in T_i released at a critical instant meets its deadline, then every job in the task set $\mathbf{T}$ will meets its deadline.

Thus, for the set of tasks in an implementation of the motor control system, the feasibility test is not sufficient and necessary, but only sufficient. Hence, we cannot conclude that the motor control system is infeasible when the feasibility test fails, as the worst-case situation may never occur.

The tool is the reason why we do not consider the second implementation alternative for the sensor subsystem. Let us for the moment ignore blocking. Then, when phasing is ignored the second alternative simply reduces to the first alternative, as all periods, execution times and relative deadlines are identical. Let us again consider blocking, and consider it as execution time added to the start of each job in a task. Now, the second alternative places a lower demand on the processor as blocking is only found in the first implementation alternative. In the second implementation alternative for the sensor subsystem the mutual exclusive access was implemented by different phases, cf. section 10.6.4. Thus, if a motor control system using the first

implementation alternative is feasible, then a motor control system using the second alternative is feasible too.

10.9.2 Calculating Blocking

In this section we explain how the blocking for the motor control system implementations considered in the following was calculated. In all implementation the *Control* task is assigned the lowest priority.

In the interfaces between the three subsystems we find two monitors, *MonitorADC* and *MonitorUf*, both protected by the simplest NPCS implementation to ensure mutual exclusive access.

The *MonitorADC* contains sixteen words. The operation of reading or writing all sixteen words in a mutual exclusive manner will be carried out in 2.1μs, cf. table 10.1. The *MonitorUf* contains two words. The operation of reading or writing both words in a mutual exclusive manner will be carried out in 0.3μs.

Both monitors are utilised by the *Control* task but *not* at the same time. Thus, the longest continuous resource allocation is made by accessing the *MonitorADC*. Because *Control* is the task with lowest priority, *all* other tasks can be blocked for the duration of 2.1μs.

10.9.3 First Implementation of the Motor Control System

In the first implementation of the motor control system the first implementation alternative of the sensor subsystem, the control subsystem, and the first implementation alternative of the actuation subsystem are combined.

10.9.3.1 Feasibility Test

The three situation tables, 10.11, 10.13, and 10.14, of the three subsystems are combined into one, which is used as input to the feasibility test.

Priorities are assigned using the deadline monotonic approach, as deadlines are less than or equal to periods. The priorities are listed in the 'Priority' column of situation table 10.18 on the next page. As blocking is present, the deadline monotonic priority assignment is no longer optimal. The blocking times for the individual tasks are listed in the 'Blocking' column of the situation table.

The feasibility test indicates that the motor control system is feasible. The worst-case response times of the individual tasks released at a critical instant are listed in the 'Completion time' column of table 10.18.

The total utilisation of the first implementation of the motor control system is calculated to be $U = 96.7\%$.

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]	Priority	Blocking [μs]	Completion time [μs]
ADCPEC [1]	PEC	10.000	0.0001	0.100	1	2.1	2.2
ADCPEC [2]	PEC	10.000	0.0001	0.100	2	2.1	2.3
ADCPEC [3]	PEC	10.000	0.0001	0.100	3	2.1	2.4
ADCPEC [4]	PEC	10.000	0.0001	0.100	4	2.1	2.5
ADCPEC [5]	PEC	10.000	0.0001	0.100	5	2.1	2.6
ADCPEC [6]	PEC	10.000	0.0001	0.100	6	2.1	2.7
ADCPEC [7]	PEC	10.000	0.0001	0.100	7	2.1	2.8
ADCPEC [8]	PEC	10.000	0.0001	0.100	8	2.1	2.9
ADCPEC [9]	PEC	10.000	0.0001	0.100	9	2.1	3.0
ADCPEC [10]	PEC	10.000	0.0001	0.100	10	2.1	3.1
ADCPEC [11]	PEC	10.000	0.0001	0.100	11	2.1	3.2
ADCPEC [12]	PEC	10.000	0.0001	0.100	12	2.1	3.3
ADCPEC [13]	PEC	10.000	0.0001	0.100	13	2.1	3.4
ADCPEC [14]	PEC	10.000	0.0001	0.100	14	2.1	3.5
ADCPEC [15]	PEC	10.000	0.0001	0.100	15	2.1	3.6
ADCPEC [16]	PEC	10.000	0.0001	0.100	16	2.1	3.7
ADCPECLISR	LISR	10.000	0.0072	0.100	17	2.1	10.9
DriverCAPCOM6	LISR	0.100	0.0456	0.100	18	2.1	56.5
DriverADC	Task	10.000	0.0502	8.300	19	2.1	152.3
Control	Task	10.000	5.0524	10.000	20	0.0	9 397.8

Table 10.18: *Situation table for the motor control system; first implementation alternative. The 'Priority' column indicates the priorities assigned using the deadline monotonic approach. The blocking time times for the individual tasks are listed in the 'Blocking' column. The total utilisation is* $U = 96.7\%$.

10.9.3.2 Estimating Spare Capacity

In this section we will estimate the spare capacity of the first motor control system implementation.

By *spare capacity* we mean the amount of time that can be used for increasing the switching frequency, while maintaining the responsiveness of the motor control system. The *responsiveness* is defined as the period of the *Control* task.

It is important to understand that the amount of spare capacity is not computed by subtracting total utilisation from 100% [Klein et al., 1993]. Instead, we shall find the extra amount of time within one period of the *Control* task, which can be used for increasing the switching frequency, without any task in the motor control system misses its deadline.

To find the maximum achievable switching frequency, we reduce the period of *DriverCAPCOM6*, i.e. increasing the switching frequency, until a deadline is missed in the motor control system.

The maximum achievable switching frequency was found to be 10.7kHz, which equals a period of 93.4μs for *DriverCAPCOM6*. The total utilisation for the motor control system was calculated to be $U = 99.9\%$ for the maximum switching frequency. *Control* is the first task in which a job released at a critical instant will miss its deadline.

Comparing the total utilisations of the first implementation of the motor control system for the two switching frequencies, we see that the spare utilisation is $0.999 - 0.967 = 0.032$. Hence, the spare capacity of the first implementation of the motor control system is $10\text{ms} \cdot 0.032 = 0.32\text{ms}$.

10.9.4 Second Implementation of the Motor Control System

In the second implementation of the motor control system the first implementation alternative of the sensor subsystem, the control subsystem, and the second implementation alternative of the actuation subsystem are combined.

10.9.4.1 Feasibility Test

The three situation tables, 10.11, 10.13, and 10.17, for the three subsystems are combined into situation table 10.19. Again, priorities are assigned according to the deadline monotonic approach.

The feasibility test indicates that the second implementation of the motor control system is also feasible. The worst-case response times of the individual tasks released at a critical instant is listed in the 'Completion time' column of table 10.19.

Let us return to the problem discussed in section 10.8.2.3, where the estimated execution time for *DriverCAPCOM6* exceeding its deadline. If we had not discovered the problem when constructing situation table 10.16, the problem would have been identified by the feasibility test. The revised design solved the problem. By inspecting the worst-case completion times in situation table 10.19, we see that *PECLISR*

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]	Priority	Blocking [μs]	Completion time [μs]
ADCPEC [1]	PEC	10.000	0.0001	0.100	1	2.1	2.2
ADCPEC [2]	PEC	10.000	0.0001	0.100	2	2.1	2.3
ADCPEC [3]	PEC	10.000	0.0001	0.100	3	2.1	2.4
ADCPEC [4]	PEC	10.000	0.0001	0.100	4	2.1	2.5
ADCPEC [5]	PEC	10.000	0.0001	0.100	5	2.1	2.6
ADCPEC [6]	PEC	10.000	0.0001	0.100	6	2.1	2.7
ADCPEC [7]	PEC	10.000	0.0001	0.100	7	2.1	2.8
ADCPEC [8]	PEC	10.000	0.0001	0.100	8	2.1	2.9
ADCPEC [9]	PEC	10.000	0.0001	0.100	9	2.1	3.0
ADCPEC [10]	PEC	10.000	0.0001	0.100	10	2.1	3.1
ADCPEC [11]	PEC	10.000	0.0001	0.100	11	2.1	3.2
ADCPEC [12]	PEC	10.000	0.0001	0.100	12	2.1	3.3
ADCPEC [13]	PEC	10.000	0.0001	0.100	13	2.1	3.4
ADCPEC [14]	PEC	10.000	0.0001	0.100	14	2.1	3.5
ADCPEC [15]	PEC	10.000	0.0001	0.100	15	2.1	3.6
ADCPEC [16]	PEC	10.000	0.0001	0.100	16	2.1	3.7
DriverCAPCOM6	PEC	0.100	0.0003	0.100	19	2.1	4.0
ADCPECLISR	LISR	10.000	0.0072	0.100	20	2.1	11.2
PECLISR	LISR	10.000	0.0056	0.100	21	2.1	16.8
PECHISR	HISR	10.000	0.1051	1.000	22	2.1	122.2
DriverADC	Task	10.000	0.0502	8.300	23	2.1	172.4
PhaseGenerator	Task	10.000	4.1051	9.000	24	2.1	4 289.8
Control	Task	10.000	5.0524	10.000	25	0.0	9 355.4

Table 10.19: *Situation table for the motor control system; second implementation alternative. The 'Priority' column indicates the priorities assigned using the deadline monotonic approach. The blocking time times for the individual tasks are listed in the 'Blocking' column. The total utilisation is* $U = 93.6\%$.

and *PECHISR* both complete before their respective deadlines.

The total utilisation of the second implementation of the motor control system is $U = 93.6\%$.

Assuming the estimate of the administrative overhead of a LISR is correct, the lower total utilisation is due to the use of specialised features of the execution environment, i.e. the PEC unit of the C164CI microcontroller.

10.9.4.2 Estimating Spare Capacity

In this section we will estimate the spare capacity of the second motor control system implementation. We refer to section 10.9.3.2 for a definition of spare capacity and a discussion of how to calculate it.

To increase the switching frequency of the second implementation, we must modify the tasks *PhaseGenerator* and *DriverCAPCOM6* of the second implementation alternative for the actuation subsystem.

We briefly summarise the relevant parts of the design. The task *PhaseGenerator* fills a buffer with PWM pattern triples. The PWM pattern triples in the buffer are fed to the CAPCOM6 unit by *DriverCAPCOM6* for each period of the *Control* task.

Hence, to increase the switching frequency the *PhaseGenerator* must put more PWM pattern triples into the buffer, and the period of *DriverCAPCOM6* must be reduced to feed the buffer contents to the CAPCOM6 unit, all within the period of the *Control* task. The execution time estimate of *PhaseGenerator* must be revised for every change in buffer size.

The maximum achievable switching frequency was found to be 11.5kHz. To support this switching frequency the buffer must hold 115 PWM pattern triples. The increased buffer size yields an increase in the execution time estimate for *PhaseGenerator* from 4,105.1μs to 4,705.1μs. The required period of *DriverCAPCOM6* is 85.9μs.

The resulting total utilisation for the motor control system was calculated to be $U = 99.6\%$ for the maximum switching frequency. *Control* is the first task in which a job released at a critical instant will miss its deadline.

Comparing the total utilisations of the second implementation of the motor control system for the two switching frequencies, we see that the spare utilisation is $0.996 - 0.936 = 0.060$. Hence, the spare capacity of the first implementation of the motor control system is $10\text{ms} \cdot 0.060 = 0.60\text{ms}$.

10.9.5 Increasing Switching Frequency by Reducing Responsiveness

In the sections 10.9.3.2 and 10.9.4.2 we estimated the spare capacity of the two implementations of the motor control system. The spare capacity corresponded to an increase in the switching frequency from 10kHz to 10.7kHz and 11.5kHz for for the first and second implementation, respectively.

In this section we predict the cost in reduced responsiveness that must be paid for a 25% increase of the switching frequency, that is a switching frequency of 12.5kHz. We will only consider the first implementation of the motor control system, i.e. the combination of the first implementation alternative of the sensor subsystem, the control subsystem, and the first implementation alternative of the actuation subsystem.

To reduce the responsiveness of the motor control system, we increase the period of all tasks in the sensor and control subsystems. All tasks in these two subsystems share the same period. The switching frequency can then be increased by decreasing the period of the task *DriverCAPCOM6* in the actuation subsystem. *DriverCAP-COM6* is implemented by a LISR.

Based on experiments we found that by increasing the period of the control and sensor subsystem by 19.5% the system was able to meet its deadlines with a switching frequency of 12.5kHz. A situation table for the resulting motor control system is shown in table 10.20 on the facing page. The resulting total utilisation is $U = 99.8\%$.

10.9.6 Summary

This section concludes the first schedulability analysis performed in the design phase of the development process. We have performed a rather extensive schedulability analysis of the initial physical design, and our findings are summarised below:

- The first implementation of the motor control system was feasible with a total utilisation $U = 96.7\%$.
- The second implementation of the motor control system was also feasible with a lower total utilisation $U = 93.6\%$.
- The spare capacity of the first implementation of the motor control system was 0.32ms, which could be used for increasing the switching frequency from the required 10kHz to 10.7kHz, with a resulting total utilisation $U = 99.9\%$.
- The spare capacity of the second implementation of the motor control system was 0.60ms, which could be used for increasing the switching frequency from the required 10kHz to 11.5kHz, with a resulting total utilisation $U = 99.6\%$.
- Finally, we have estimated the necessary reduction in the responsiveness of the first implementation of the motor control system in order to increase the switching frequency by 25%. By increasing the period of the sensor and control subsystems by 19.5% the switching frequency of 12.5kHz was feasible.

Based on the findings of the schedulability analysis we conclude that both designs are feasibible and can achieve high total utilisations. When using the spare capacity to increase the switching frequency of the two implementations, the performance of the second implementation is better than the performance of the first implementation. However, the better performance comes at the price of a more complex physical design using specialised features of the execution environment.

The findings of the schedulability analysis may be used for deciding how to proceed in the design process. We have three main options:

Task name	Type	Period [ms]	Exec. [ms]	Deadline [ms]	Priority	Blocking [μs]	Completion time [μs]
ADCPEC [1]	PEC	11.950	0.0001	0.100	1	2.1	2.2
ADCPEC [2]	PEC	11.950	0.0001	0.100	2	2.1	2.3
ADCPEC [3]	PEC	11.950	0.0001	0.100	3	2.1	2.4
ADCPEC [4]	PEC	11.950	0.0001	0.100	4	2.1	2.5
ADCPEC [5]	PEC	11.950	0.0001	0.100	5	2.1	2.6
ADCPEC [6]	PEC	11.950	0.0001	0.100	6	2.1	2.7
ADCPEC [7]	PEC	11.950	0.0001	0.100	7	2.1	2.8
ADCPEC [8]	PEC	11.950	0.0001	0.100	8	2.1	2.9
ADCPEC [9]	PEC	11.950	0.0001	0.100	9	2.1	3.0
ADCPEC [10]	PEC	11.950	0.0001	0.100	10	2.1	3.1
ADCPEC [11]	PEC	11.950	0.0001	0.100	11	2.1	3.2
ADCPEC [12]	PEC	11.950	0.0001	0.100	12	2.1	3.3
ADCPEC [13]	PEC	11.950	0.0001	0.100	13	2.1	3.4
ADCPEC [14]	PEC	11.950	0.0001	0.100	14	2.1	3.5
ADCPEC [15]	PEC	11.950	0.0001	0.100	15	2.1	3.6
ADCPEC [16]	PEC	11.950	0.0001	0.100	16	2.1	3.7
ADCPECLISR	LISR	11.950	0.0072	0.100	17	2.1	10.9
DriverCAPCOM6	LISR	0.800	0.0456	0.800	18	2.1	56.5
DriverADC	Task	11.950	0.0502	10.250	19	2.1	152.3
Control	Task	11.950	5.0524	11.950	20	0.0	11 905.8

Table 10.20: *Situation table for the motor control system; first implementation alternative with increased switching frequency and reduced responsiveness. Priorities are assigned using the deadline monotonic approach. The total utilisation is* $U = 99.8\%$.

- If a simple design is desired for simplifying the implementation, or if a future increase in the switching frequency of 7% is considered sufficient, or if a reduction of the system's responsiveness is an option, we should proceed with the first implementation of the motor control system.
- If the property of a future increase in the switching frequency of 15% is desired, a reduction of the systems responsiveness is not an option, and we are confident that we can implement the more complex physical design, we should proceed with the second implementation of the motor control system.
- If the requirements of a switching frequency of 10kHz and a responsiveness of 10ms are final, the microcontroller of the execution environment may be overdimensioned. If the motor control system is to be produced in high numbers it may be profitable to consider a less powerful microcontroller and repeat the design if necessary.

It should be emphasised, that this has been the first schedulability analysis of the design process. The analysis was based on initial execution time estimates. Naturally, the analysis depends on correct estimates. When physical design for the motor control system has been selected and the implementation is started, more precise estimates may become available.

Hence, during the following phases of the development process, we should continue to analyse the schedulability of the design, while the individual subsystems are at various degrees of completion. During the following phases the execution time estimates will converge towards the actual execution times.

If the sufficient feasibility test of the schedulability analysis indicates that the system is infeasible, we must revert to the design process and improve the design. Before doing so, we should apply a necessary and sufficient feasibility test, to ensure the design is indeed infeasible.

11 Conclusion

After the motivation of hard real-time systems we introduced an informal model for hard real-time systems. The model formed a basis for investigating important components of such systems, e.g. schedules, priority-driven schedulers, and the problem of validating that a schedule is indeed feasible for a simplified periodic task model without shared resources. Having investigated the problem of uncontrolled blocking associated with resource contention we introduced the concept of synchronisation protocols, which allows the introduction of shared resources into the periodic task model. Based on the theory of the periodic task model, we defined a computational model, for which an implementation in a real-time operating system was proposed. Finally, a development process for hard real-time systems was proposed, and the design phases of the process were applied to a case study.

11.1 Results

The Reference Model

An informal model of real-time systems was constructed. The model presented definitions and designations, which allowed us to describe and discuss aspects of hard real-time systems.

Hard Real-Time Scheduling

We presented the taxonomy of hard real-time scheduling. Rather than exploring the static scheduling paradigm, we investigated the more flexible priority-driven scheduling paradigm. We discussed the optimality of priority-driven schedulers. The dynamic-priority earliest deadline first scheduling algorithm was found to be optimal for the most general sets of tasks.

We chose to focus on the fixed-priority schedulers as they are widely supported by current real-time operating systems. A fixed-priority scheduler may be divided into two components:

- The priority assignment algorithm
- The dynamic scheduling decision

Three optimal priority assignment algorithms have been discussed. The rate monotonic algorithm required deadlines to equal periods in order to be optimal. The deadline monotonic algorithm allowed deadlines less than or equal to periods, hence it provides means for controlling jitter, while remaining optimal. Finally, Audsley's algorithm allowed the task of a task set to have arbitrary phases and deadlines less than or equal to periods and remain optimal.

Feasibility Tests

For hard real-time systems it must be possible to validate that a schedule is indeed feasible, i.e. all jobs in a task set will meet their deadline. We presented different techniques for performing a feasibility test. Most necessary and sufficient feasibility tests require tool support for realistic systems.

Synchronisation Protocols

We have discussed the problems introduced by blocking, and we have addressed methods to control blocking, i.e. the highest locker synchronisation protocol. Thus, it was possible to extend the simplified periodic task model to include shared resources. Additionally, the blocking factor was introduced in the feasibility tests.

Implementation of a Computational Model

Based on a subset of the presented theory a computational model for hard real-time designs was defined. The computational model supports designs of hard real-time systems, which are analysable using the feasibility tests.

Based on an investigation of the functionality and concepts of a commercially available real-time operating system, we implemented the defined computational model upon the operating system. The overhead of the implemented synchronisation protocols were measured, compared, and discussed.

The Hard Real-Time Development Process

We proposed a development process for hard real-time systems. We focussed especially on the activity of architectural design, which was divided into two phases: the logical and physical design phase. The objective of the design process is to trace a feasible design from requirements to deployment. The logical design addresses the requirements that does not relate to time. The requirements that relate to time are addressed by the physical design phase. In the initial iteration of the physical design execution time estimates are associated with the individual processes of the design. During the development process the execution time estimates are revised. A feasible design is traced by repeatedly applying a feasibility test.

The Case Study

The design phases of the proposed development process was tested on a case study, concerning the construction of a motor control system. The first iterations of the logical and physical design phases were completed for the case study.

The logical design phase successfully divided the motor control system into three subsystems. The physical design phase resulted in several design proposals for each subsystem. By introducing timing constraints in the physical design phase we could immediately detect problems in the design of one of the subsystems. The design was modified in order to resolve the problem.

The different design proposals of subsystems could successfully be combined resulting in a complete design of a motor control system.

Two different designs of the motor control system were composed from the design alternatives for each subsystem. Both designs were found to be feasible. We inves-

tigated the spare capacity of the two designs. For one of the designs the necessary reduction in responsiveness was estimated for a 25% increase in the switching frequency of the motor control system.

11.2 Evaluation

Scheduling theory for hard real-time systems has matured during the early 1990s. Thus, a large number of results are now available. During the project we have studied the fundamentals of the scheduling theory for hard real-time systems, and we have discussed relevant parts in this thesis.

The Hard Real-Time Development Process

We have tested the proposed development process in a larger case study. We completed the first iterations of the logical and physical design phases.

The idea of separating the architectural design phase into a logical and physical design phase seems to work well. Addressing the real-time requirements in a separate design phase forces the designer to focus on the real-time constraints during the initial design of the system.

Applying the schedulability analysis repeatedly during the development process allows the designer to trace a feasible design through the development process. Additionally, the schedulability analysis may provide guidance through the development process and provide the designer with a tool for making design decisions.

The Computational Model

Based on the presented scheduling theory we formed a computational model, which we have successfully implemented onto a commercial real-time operating system. The computational model could be extended. We discuss two possible extensions below:

- The computational model did not allow arbitrary deadlines though we mentioned a technique which made it possible to validate systems with arbitrary deadlines. In our situation we had a throughput requirement hence deadlines was less then or equal to periods.

  If the computational model was extended to allow arbitrary deadlines the implementation of the computational model had to be reconsidered. We have identified two approaches to implement a computational model that allows arbitrary deadlines:

  - A new task of same priority is dynamically created each time the periodic event occurs. This approach involves dynamic task creation and task deletion and is expected to produce a large overhead.
  - The periodic task has a counter of outstanding jobs. Each time the periodic event occurs the counter is incremented. As long as there is outstanding jobs the task continues to execute.

- In the reference model we identified aperiodic events, but we did not present any results for this. Aperiodic events was not allowed in the computational model. Thus, in order facilitate validation and analysis of systems with aperiodic events these must be cast into the periodic task model.

  Far from all systems have a truly periodic behaviour. Thus, the computational model should be extended to deal with aperiodic events. There are several possibilities when dealing with aperiodic activities:

  Periodic polling Aperiodic activity generated by peripherals must be transformed into periodic polling if possible. The hardware engineer must have this in mind when constructing interfaces to peripherals. Periodic polling introduces more event latency and more overhead but nevertheless periodic overhead is analysable.

  Sporadic servers Aperiodic activity can be normalised into a periodic activity using *sporadic servers*. A sporadic server is a periodic task with budget of execution time which can be used to aperiodic activity. For more information about sporadic servers see [Sprunt, 1990], [Liu, 2000] and [Briand and Roy, 1999].

  Ignore them If a system is overwhelmed by handling aperiodic activities it may result in missed deadlines. The system should be designed in a way such that only the newest is needed and therefore treated. The old ones should be discarded.

Hard Real-Time Operating Systems

We have examined a commercially available real-time operating system. It is our opinion that the particular operating system is too big and complex for hard real-time systems.

In our work with the scheduling theory for hard real-time systems we have identified a small set of required functionality that a simple kernel should provide:

- A fixed-priority preemptive scheduler.
- Support for periodic tasks, i.e. absolute timers.
- Provide fast task priority changes, i.e. to be used in the implementation of the highest locker protocol.
- To make it possible for an interrupt service routine to signal on a semaphore, i.e. to release a task.
- Support for sporadic servers.

In general a hard real-time operating system should provide support for common real-time abstractions. Thus, the operating system should not introduce superfluous concepts.

Final Remarks

We have accomplished the objectives of the thesis. By preparing this thesis we have obtained fundamental understanding of the problems involved in the development

of hard real-time systems. In particular, we feel confident in the approach to the development and documentation of hard real-time systems suggested in this thesis.

A Highest Locker Protocol

This appendix features listings of the source code presented in chapter 8.2.2.

A.1 Simple Implementation

Listing A.1: *Interface to the simple highest locker protocol.*

```
/*
 * file: simple_highest_locker.h
 * desc: interface to the simple highest locker protocol
 * copyright 2001, Thomas Hedemand Nielsen & Jens Christian Schwarzer
 */

typedef struct critical_region_control_block {
  OPTION priority;    /* must be initialised to the ceiling priority */
  OPTION saved_priority;
} CRCB;

VOID enter_crit(CRCB *);

VOID leave_crit(CRCB *);
```

Listing A.2: *Implementation of the simple highest locker protocol.*

```
/*
 * file: simple_highest_locker.c
 * desc: implementation of the simple highest locker protocol
 * copyright 2001, Thomas Hedemand Nielsen & Jens Christian Schwarzer
 */

#include "nucleus.h"
#include "simple_highest_locker.h"

VOID enter_crit(CRCB *region) {
  region->saved_priority =
    NU_Change_Priority(NU_Current_Task_Pointer(),
                       region->priority);
}

VOID leave_crit(CRCB *region) {
  NU_Change_Priority(NU_Current_Task_Pointer(),
                     region->saved_priority);
}
```

A.2 General Implementation

Listing A.3: *Interface to the general highest locker protocol.*

```
1  /*
2   * file: general_highest_locker.h
3   * desc: interface to the general highest locker protocol
4   * copyright 2001, Thomas Hedemand Nielsen & Jens Christian Schwarzer
5   */
6
7  typedef struct critical_region_control_block CRCB;
8
9  typedef struct critical_region_control_block {
10   OPTION priority;    /* must be initialised to the ceiling priority */
11   CRCB *next, *prev;
12 };
13
14 typedef struct thread_control_block_extension {
15   OPTION normal_priority;   /* must be initialised with the priority */
16   OPTION current_priority;  /* must be initialised with the priority */
17   CRCB *region_list;        /* must be initialised with 0 */
18 } TCBE;
19
20 VOID enter_crit(CRCB *, TCBE *);
21
22 VOID leave_crit(CRCB *, TCBE *);
```

Listing A.4: *Implementation of the general highest locker protocol.*

```
1  /*
2   * file: general_highest_locker.c
3   * desc: implementation of the general highest locker protocol
4   * copyright 2001, Thomas Hedemand Nielsen & Jens Christian Schwarzer
5   */
6
7  #include "nucleus.h"
8  #include "general_highest_locker.h"
9
10 VOID insert(CRCB *list, CRCB *node) {
11   if (0 == list)
12     list = node->next = node->prev = node;
13   else {
14     node->next = list;
15     node->prev = list->prev;
16     list->prev->next = node;
17     list->prev = node;
18   }
19 }
20
21 VOID remove(CRCB *list, CRCB *node) {
22   if (node->next == node)
23     list = 0;
24   else {
25     node->prev->next = node->next;
26     node->next->prev = node->prev;
27     if (node->next == node->prev)
28       list = node->next;
29   }
30 }
31
32 OPTION find_greatest_priority(CRCB *list, OPTION max) {
```

```
33    CRCB *node;
34    if (0 != list) {
35      node = list;
36      do {
37        if (node->priority > max)
38          max = node->priority;
39        node = node->next;
40      } while (node != list);
41    }
42    return max;
43  }
44
45  VOID enter_crit(CRCB *region, TCBE *thread) {
46    if (region->priority > thread->current_priority) {
47      NU_Change_Priority(NU_Current_Task_Pointer(),
48                         region->priority);
49      thread->current_priority = region->priority;
50    }
51    insert(thread->region_list, region);
52  }
53
54  VOID leave_crit(CRCB *region, TCBE *thread) {
55    remove(thread->region_list, region);
56    thread->current_priority =
57      find_greatest_priority(thread->region_list,
58                             thread->normal_priority);
59    NU_Change_Priority(NU_Current_Task_Pointer(),
60                       thread->current_priority);
61  }
```

A.3 Overhead Comparison

Listing A.5: *Application for highest locker protocol overhead measurement.*

```
/*
 * file: speed_test_demo.c
 * desc: overhead measurement of the highest locker protocols
 * copyright 2001, Thomas Hedemand Nielsen & Jens Christian Schwarzer */

/* use the simple HL implementation */
/*#define SHL */

/* use the general HL implementation */
/*#define GHL*/

/* use the built-in NP semaphore */
/* none of the two above should be defined */

#include "nucleus.h"

#if defined(SHL)
#include "../simple_highest_locker/simple_highest_locker.h"
#elif defined(GHL)
#include "../general_highest_locker/general_highest_locker.h"
#endif

NU_TASK Task;

VOID task(UNSIGNED argc, VOID *argv);

NU_MEMORY_POOL System_Memory;

#if defined(SHL) || defined(GHL)
CRCB Crit;
#else
NU_SEMAPHORE Semaphore;
#endif

#ifndef GHL
typedef UNSIGNED TCBE;
#endif

TCBE tcbe;

VOID lock(TCBE *tcbe) {
#if defined(SHL)
  enter_crit(&Crit);
#elif defined(GHL)
  enter_crit_g(&Crit, tcbe);
#else
  NU_Obtain_Semaphore(&Semaphore, NU_SUSPEND);
#endif
}

VOID unlock(TCBE *tcbe) {
#if defined(SHL)
  leave_crit(&Crit);
#elif defined(GHL)
  leave_crit_g(&Crit, tcbe);
#else
  NU_Release_Semaphore(&Semaphore);
```

```
58 #endif
59 }
60
61 VOID Application_Initialize(VOID *first_available_memory) {
62
63   VOID *pointer;
64
65   NU_Create_Memory_Pool(&System_Memory, "SYSMEM",
66                         first_available_memory, 20000, 50, NU_FIFO);
67
68   NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
69
70   NU_Create_Task(&Task, "Task", task, 0, NU_NULL, pointer, 1000, 5, 0,
71                  NU_PREEMPT, NU_START);
72
73 #if defined(SHL) || defined(GHL)
74   Crit.priority = 4;
75 #else
76   NU_Create_Semaphore(&Semaphore, "SEMAPHORE", 1, NU_PRIORITY);
77 #endif
78
79 #ifdef GHL
80   tcbe.normal_priority = tcbe.current_priority = 5;
81   tcbe.region_list = 0;
82 #endif
83 }
84
85 VOID task(UNSIGNED argc, VOID *argv) {
86
87   lock(&tcbe);
88
89   unlock(&tcbe);
90
91 }
```

B Execution Times of Nucleus Plus System Services on an Infineon C164

This appendix shows the results of a test application constructed to measure execution times of a selected set of Nucleus Plus system services.

The test application consists of two tasks which have a controlled behaviour due to their set up. The measurement was carried out using the CrossView Pro debugger as in section 8.2.3. The test application can be found in listing B.1. Twenty-one measurements are performed by the application and the result can be found in table B.1.

Test number	Legend	Context switch	Machine cycles
1	Task go to sleep	yes	558
2	Task relinquish itself	yes	324
3	Suspend a task	no	306
4	Resume a task	no	287
5	Suspend a task	yes	466
6	Resume a task	yes	478
7	Immediate release semaphore, no task waiting at all	no	170
8	Immediate obtain semaphore	no	185
9	Obtain semaphore, blocked by task of lower priority	yes	551
10	Release semaphore, no task of higher priority waiting	no	339
11	Release semaphore, task of higher priority waiting	yes	551
12	Task priority change	yes	527
13	Task priority change	no	525
14	Retrieve the current task pointer	no	32
15	Disabling of task preemption	no	170
16	Enabling of task preemption, no task of higher priority waiting	no	179
17	Enabling of task preemption, task of higher priority waiting	yes	320
18	Protect kernel data structure	no	67
19	Unprotect kernel data structure	no	61
20	Disabling of all interrupts using OS system call	no	26
21	Enabling of all interrupts using OS system call	no	26

Table B.1: *Execution times of a selected set of Nucleus Plus system services on a C164 microcontroller.*

Listing B.1: *Application for Nucleus Plus system services execution times.*

```
/*
 * file: nucleus_wcet_c164ci.c
 * desc: program to measure the execution times for system services of
 *       the Nucleus Plus operating system on a Infineon C164 target.
 * copyright 2001, Thomas Hedemand Nielsen & Jens Christian Schwarzer
 */

#include "nucleus.h"

/* Define Application data structures. */
NU_TASK Control;
NU_TASK Sub_Task;
NU_SEMAPHORE Semaphore;
NU_MEMORY_POOL System_Memory;

/* Data structure used in test 13 and 14. */
NU_PROTECT test_protect;

/* Define prototypes for function references. */
VOID Control_Entry(UNSIGNED, VOID *);
VOID Sub_Task_Entry(UNSIGNED, VOID *);

/* Define the Application_Initialise routine that determines the
   initial Nucleus PLUS application environment. */

VOID Application_Initialize(VOID *first_available_memory) {

  VOID *pointer;

  /* Create a system memory pool that will be used to allocate task
     stacks, queue areas, etc. */
  NU_Create_Memory_Pool(&System_Memory, "SYSMEM",
                        first_available_memory, 8192, 50, NU_FIFO);

  /* Create Control task. */
  NU_Allocate_Memory(&System_Memory, &pointer, 2000, NU_NO_SUSPEND);
  NU_Create_Task(&Control, "Control", Control_Entry, 0, NU_NULL,
                 pointer, 2000, 5, 0, NU_PREEMPT, NU_START);

  /* Create Sub Control task. */
  NU_Allocate_Memory(&System_Memory, &pointer, 2000, NU_NO_SUSPEND);
  NU_Create_Task(&Sub_Task, "Sub Task", Sub_Task_Entry, 0, NU_NULL,
                 pointer, 2000, 6, 0, NU_PREEMPT, NU_START);
}

VOID Control_Entry(UNSIGNED argc, VOID *argv) {

  /* -1A- Calculate time for a sleep with context switch. */
  NU_Sleep(1);

  /* -2B- */

  /* -3A- Calculate time for suspending a task */
  NU_Suspend_Task(&Sub_Task);
  /* -3B- */

  /* -4A- Calculate time for resuming a task with no context switch */
  NU_Resume_Task(&Sub_Task);
  /* -4B- */

```

```
61    NU_Suspend_Task(&Sub_Task);
62
63    NU_Change_Priority(&Sub_Task, 4);
64
65    /* -5A- Calculate time for resuming a task with context switch */
66    NU_Resume_Task(&Sub_Task);
67
68    /* -6B- */
69
70    NU_Change_Priority(&Sub_Task, 5);
71
72    NU_Resume_Task(&Sub_Task);
73
74    NU_Create_Semaphore(&Semaphore, "Semaphore", 0, NU_FIFO);
75
76    /* -7A- Calculate the time for immediate release of semaphore. */
77    NU_Release_Semaphore(&Semaphore);
78    /* -7B- */
79
80    /* -8A- Calculate the time for immediate obtain semaphore. */
81    NU_Obtain_Semaphore(&Semaphore, NU_NO_SUSPEND);
82    /* -8B- */
83
84    /* -9A- Calculate the time to suspend trying to obtain the
85       semaphore. */
86    NU_Obtain_Semaphore(&Semaphore, NU_SUSPEND);
87
88    /* Control comes back here after semaphore is released by sub
89       task. */
90    NU_Obtain_Semaphore(&Semaphore, NU_SUSPEND);
91
92    /* -11B- */
93
94    /* -12A- Calculate the time for priority change involving a context
95       switch. */
96    NU_Change_Priority(&Control, 6);
97
98    /* -14A- Calculate the time to retrieve the current task pointer. */
99    NU_Current_Task_Pointer();
100   /* -14B- */
101
102   /* -15A- Calculate the time for disabling the task scheduler. */
103   NU_Change_Preemption(NU_NO_PREEMPT);
104   /* -15B- */
105
106   /* -16A- Calculate the time for enabling the task scheduler. */
107   NU_Change_Preemption(NU_PREEMPT);
108   /* -16B- */
109
110   NU_Change_Preemption(NU_NO_PREEMPT);
111
112   NU_Change_Priority(&Sub_Task, 4);
113
114   /* -17A- Calculate the time for enab. the sched. w/context switch. */
115   NU_Change_Preemption(NU_PREEMPT);
116
117   /* Set up a protect data structure. */
118   test_protect.words[0] = 0;
119   test_protect.words[1] = 0;
120
121   /* -18A- Calculate the time for protecting a data structure from
```

```
122       simultaneous access. */
123     NU_Protect(&test_protect);
124     /* -18B- */
125
126     /* -19A- Calculate the time for unprotecting a data structure from
127        simultaneous access.*/
128     NU_Unprotect();
129     /* -19B- */
130
131     /* -20A- Calculate the time for disabling all interrupts using a OS
132        system call. */
133     NU_Control_Interrupts(NU_DISABLE_INTERRUPTS);
134     /* -20B- */
135
136     /* -21A- Calculate the time for enabling all interrupts using a OS
137        system call.*/
138     NU_Control_Interrupts(NU_ENABLE_INTERRUPTS);
139     /* -21B- */
140   }
141
142   VOID Sub_Task_Entry(UNSIGNED argc, VOID *argv) {
143
144     UNSIGNED clock_value;
145
146     /* -1B- */
147
148     /* Both tasks must now have same priority. */
149     NU_Change_Priority(&Sub_Task, 5);
150
151     /* Wait for clock tick to make sure control task is not sleeping. */
152     clock_value = NU_Retrieve_Clock();
153     while (NU_Retrieve_Clock() == clock_value) { }
154
155     /* -2A- Calculate Relinquish time with context switch. */
156     NU_Relinquish();
157
158     /* -5B- */
159
160     /* -6A- Calculate time for suspending a task with context switch. */
161     NU_Suspend_Task(&Sub_Task);
162
163     /* -9B- */
164
165     /* -10A- Release the semaphore to resume the waiting control task. */
166     NU_Release_Semaphore(&Semaphore);
167     /* -10B- */
168
169     /* Control task is now ready.  Relinquish to let control task to
170        run.  Processing returns here when another obtain is performed by
171        the control task. */
172     NU_Relinquish();
173
174     /* Raise the priority level of control task. */
175     NU_Change_Priority(&Control, 4);
176
177     /* -11A- Calculate semaphore release with immediate context switch. */
178     NU_Release_Semaphore(&Semaphore);
179
180     /* -12B- */
181
182     /* -13A- Calculate the time for priority change involving no context
```

```
183         switch. */
184     NU_Change_Priority(&Control, 5);
185     /* -13B- */
186
187     NU_Relinquish();
188
189     /* -17B- */
190
191     NU_Change_Priority(&Sub_Task, 5);
192
193     NU_Relinquish();
194
195 }
```

C Acronyms

AC	alternating current
ADC	analog digital converter
ANSI	American National Standards Institute
ATI	Accelerated Technology Incorporated
BCET	best-case execution time
CAPCOM6	capture/compare unit 6
CPU	central processing unit
DAC	digital analog converter
DC	direct current
DM	deadline monotonic
DMA	direct memory access
EDF	earliest deadline first
FAQ	frequently asked questions
FIFO	first-in, first-out
HISR	high-level interrupt service routine
HL	highest locker protocol
HPF	highest priority first
ISR	interrupt service routine
LISR	low-level interrupt service routine
MIPS	million instructions per second
NP	Nucleus Plus, by ATI, is a real-time operating systems for embedded applications
NPCS	non-preemptive critical section protocol
PCP	priority ceiling protocol
PEC	peripheral event controller
PIP	priority inheritance protocol

PWM	pulse-width modulation
RISC	reduced instruction set computing
RM	rate monotonic
RMA	rate monotonic analysis
RTC	real time clock
SVM	space vector modulation
SWPWM	sinusoidally weighted pulse-width modulation
TCB	thread control block
WCET	worst-case execution time
WRR	weighted round-robin

Bibliography

[ATI, 2000a] ATI (2000). *Nucleus PLUS Internals Manual.* Accelerated Technology Incorporated.

[ATI, 2000b] ATI (2000). *Nucleus PLUS Reference Manual.* Accelerated Technology Incorporated.

[Audsley et al., 1993] Audsley, A. N., Burns, A., Richardson, M., and Tindell, K. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292.

[Audsley, 1991] Audsley, N. C. (1991). Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS164, Dept. Computer Science, University of York.

[Audsley et al., 1991] Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. (1991). Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software.*

[Briand and Roy, 1999] Briand, L. P. and Roy, D. M. (1999). *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach.* IEEE Computer Society.

[Burns and Wellings, 1995] Burns, A. and Wellings, A. (1995). *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems.* Real-Time Safety Critical Systems. Elsevier.

[Copeland, 1999] Copeland, M. (1999). Space vector modulation and overmodulation with an 8-bit microcontroller. Microcontroller Applications AP0836, Infineon Technologies AG, St. Martin Straße 53, D-81541 München.

[Danfoss, 1998] Danfoss (1998). *Værd at vide om frekvensomformere.* Danfoss A/S, 1st edition. In Danish.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice-Hall.

[Infineon, 1999] Infineon (1999). *Microcontrollers, C166 Family, 16-Bit Single-Chip Microcontroller, C164 User's Manual.* Infineon Technologies AG, St. Martin Straße 53, D-81541 München, 2nd edition.

[Infineon, 2001] Infineon (2001). *Instruction Set Manual for the C166 Family of Infineon 16-Bit Single-Chip Microcontrollers.* Infineon Technologies AG, St. Martin Straße 53, D-81541 München, 3rd edition.

[Jackson, 1995] Jackson, M. (1995). *Software Requirements and Specifications.* Addison-Wesley.

[Klein et al., 1993] Klein, M. H., Ralya, T., Pollak, B., Obenza, R., and Harbour, M. G. (1993). *A Practitioner's Handbook for Real-Time Systems: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers.

[Labrosse, 1998] Labrosse, J. J. (1998). *MicroC/OS-II, The Real-Time Kernel.* R&D Books.

[Leung and Whitehead, 1982] Leung, J. Y. T. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation North Holland*, 2:237–250.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20:46–61.

[Liu, 2000] Liu, J. W. S. (2000). *Real-Time Systems.* Prentice Hall.

[Løvengreen, 1997] Løvengreen, H. H. (1997). Design of reactive programs. Technical report, Department of Information Technology, Technical University of Denmark.

[Rischel et al., 1987] Rischel, H., Mortensen, B. G., and Ravn, A. P. (1987). *Konstruktion af Formålsbundne Systemer.* Teknisk Forlag A/S. In Danish.

[Sprunt, 1990] Sprunt, B. (1990). *Aperiodic Task Scheduling for Real-Time Systems.* PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University.

[Stankovic and Ramamritham, 1988] Stankovic, J. A. and Ramamritham, K. (1988). *Tutorial — Hard Real-Time Systems.* IEEE Computer Society Press.

[Tasking, 2000a] Tasking (2000). *C Cross-Compiler: User's Guide C166/ST10 v7.0.* Tasking Incorporated, 5.10 edition.

[Tasking, 2000b] Tasking (2000). *Cross-Assembler, Linker/Locator, Utilities: User's Guide C166/ST10 v7.0.* Tasking Incorporated, 5.10 edition.

[Tasking, 2000c] Tasking (2000). *CrossView Pro Debugger: User's Guide C166/-ST10 v7.0.* Tasking Incorporated, 10.99 edition.