Paradigms and tools for developing dependable realtime software

Author(s):
Keller, Daniel E.

Publication Date:
2013

Permanent Link:

https://doi.org/10.3929/ethz-a-009930598 →

Diss. ETH No. 20818

Paradigms and tools for developing dependable realtime software

A dissertation submitted to the
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
DANIEL KELLER
Dipl. Informatik-Ing. ETH
born April 12, 1972
citizen of Konolfingen (BE)

accepted on the recommendation of
Prof. Dr. Jürg Gutknecht, examiner
Prof. Dr. Peter Müller, co-examiner
Prof. Dr. med. Patrick Hunziker, co-examiner

2013

Acknowledgments

A big “Thank you” to Prof. Jürg Gutknecht for his continuous guidance, support, help and never ending patience in the years past. Thank you to Prof. Peter Müller who agreed to be co-examiner. A joint effort with him on the priority inheritance protocol implementation proved very productive. I am indebted to Prof. Patrick Hunziker with whom I have implemented several challenging medical IT projects and who has biased my career towards the med-tech industry.

I would like to mention a few colleagues to whom I owe insights, inspiration and knowledge. Thomas Frey taught me all about hands on System Construction. Luc Bläser’s clear conceptual understanding has broadened my mind significantly. Felix Friedrich gave me a hand with the Oberon Compiler issues I faced. Alexey Morozov was in charge of implementing the signal processing algorithms of the presented use cases. And last, but not least, my office mate Svend Knudsen. Svend is a walking encyclopaedia of computer history. Thanks to all of them.

Numerous students have contributed to the presented implementation.

Abstract

This thesis presents research into application-aware operating systems (OS) for safety-critical applications. It has been motivated by the necessity to develop of a “next generation” wearable medical monitoring device. Commercially available devices are often based on 16 bit microcontrollers with limited possibilities in many respects. A paradigm shift towards fully fledged 32 bit technology is overdue. It allows a new generation of *wearable devices* following the “multiple parameter, multiple purposes” paradigm to be developed. Such devices offer great flexibility regarding their field of application, and they will finally replace simpler single purpose devices and even some of the stationary monitoring systems.

A de facto prerequisite to the design and build of complex embedded systems is the availability of an OS that offers a sufficiently abstract application programming interface (API) and programming model. In the case of safety-critical applications, like the one envisioned here, the OS must in addition be highly *dependable*. This thesis presents an approach towards the goal of a fully dependable OS based on a natively implemented runtime layer with some provable properties. To accompany this, an evolved high-level programming language will be introduced to support the development of dependable application software.

The scientific contribution of this work largely lies in the symbiotic relationship between programming language and OS. In particular, how to take advantage of the programming language in order to exploit and build on particular static properties of the runtime system and increase its runtime predictability will be explored.

As proof of concept, a wearable device based on a 32 bit ARM processor technology and operated by the entirely developed OS was built and field-tested in the context of a medical application with the goal of reliably monitoring

heart patients and detecting abnormalities.

Zusammenfassung

Diese Arbeit präsentiert ein applikationsspezifisches Betriebssystem für sicherheitskritische Anwendungen. Ausgangspunkt war die Entwicklung eines neuartigen, tragbaren medizinischen Überwachungsgerätes. Kommerziell erhältliche Geräte basieren häufig auf 16 bit Mikrokontrollern und sind in mehrfacher Hinsicht limitiert. Ein Paradigmenwechsel hin zu voll ausgebildeten 32 bit Prozessoren ist überfällig. Dieser Paradigmenwechsel erlaubt den Bau einer neuen Generation von Geräten, die verschiedenartigste Sensoren für diverse Zwecke nutzbar machen. Solche Geräte sind flexibel einsetzbar und können einfachere oder sogar stationäre Überwachungssysteme substituieren.

Eine Voraussetzung um komplexe, eingebettete Systeme zu bauen, ist die Verfügbarkeit eines Betriebssystems, das ein genügend abstraktes Programmiermodel und eine genügend abstrakte Programmierschnittstelle bietet. Bei den angepeilten sicherheitskritischen Anwendungen wird zusätzlich ein hohes Mass an Zuverlässigkeit gefordert. Diese Arbeit präsentiert Schritte hin zu einem vollständig zuverlässigen Betriebssystem basierend auf einem nativen System mit beweisbaren Eigenschaften. Um die Entwicklung zuverlässiger Applikationssoftware zu ermöglichen, wird die Evolution einer Programmier-Hochsprache wird eingeführt, .

Der wissenschaftliche Beitrag dieser Arbeit liegt in der Symbiose zwischen Programmiersprache und Betriebssystem. Im Speziellen wird untersucht, wie die Programmiersprache genutzt werden kann, um statische Eigenschaften des Betriebssystems zu stärken und das Laufzeitverhalten deterministischer zu machen.

Als Machbarkeitstudie wurde ein tragbares Gerät basierend auf einem 32 Bit ARM Prozessor und dem vollständig entwickelten Betriebssystem vorgestellt. Die medizinische Applikation um Herz Patienten zu überwachen, wurde im Feld getestet.

Contents

Chapter 1

Introduction

The topic and essence of this thesis is about the synthetic power of programming language and operating system (OS) co-design to build dependable systems. The term “dependable” stands for systems that produce correct results, are stable over time and tolerant with respect to unpredictable faults. The major question to be answered here is: If the chance for a green-field design is given, how should an OS and programming language be co-designed so as to be accessible to state of the art formal verification methods and also informal verification strategies.

However, introducing or improving a general formalism for verification purposes was not the goal of this thesis. Our approach is more modest, but no less effective: Efforts on different levels have been made to facilitate reasoning about the correctness of particular system properties. These efforts include:

- A programming language enhanced by suitable and sufficiently abstract primitives.
- Design patterns as generic solutions to commonly occurring problems in dependable system software design.
- An implementation that allows complexities to be easily managed.

In summary, a generic software framework for implementing dependable data driven embedded systems will be presented.

1.1 Motivation

This thesis is basically a by-product of an extensive e-health related research project about advanced health state monitoring. Since Medical Device engineering is an emerging and challenging discipline, it is an interesting showcase for a custom runtime system like the one presented in this thesis.

At the time of this thesis, wearable diagnostic devices are not very popular in medical daily routine. The limitations of such devices are at least twofold: On the one hand, the available sensor technology is not fool-proof. Depending on the application domain, experts are required to apply the sensors in order to get accurate results. And on the other hand, current mobile devices are still unable to apply sophisticated algorithms to realtime data; they usually only carry out data recording and / or visualization. Data processing is most often done offline by experts or by semi-automated expert systems. In an ideal world, patients could handle the devices and sensors themselves and experts would only have to be involved in the case of relevant medical incidents.

However, wearable medical monitoring devices have gained more interest recently due to three major trends: 1.) wide-ranging RF communication infrastructures, 2.) power-efficient processors and 3.) software tools to build dependable systems. But it is not at all a priori evident that autonomous wearable medical monitoring devices are conceptually promising in the long term. The ratio of acquired data versus wireless communication capacities will largely determine where and how realtime data will be processed. We assume that the amount of data acquired by upcoming devices based on nano sensors for instance, will grow faster than the capacities to transmit data over the air. Hence processing data locally to where it has been acquired will gain importance in the future.

With the available power aware and highly integrated multi-core 32 bit processors, the way wearable signal processing devices will be built is going to change radically. Off the shelf processors with on chip memory, signal processors, operation amplifiers and all kinds of communication controllers will allow substituting analog building blocks with software routines. In particular, analog circuits for filtering signals will become obsolete. Fewer Integrated Circuit components will reduce the overall production costs significantly.
This paradigm shift towards fully digital information processing, justifies investigations into a software framework adapted to smart wearable signal process-

ing devices.

1.2 State of the Art

By far most of the failures in dependable systems result from memory violations, leaks in the use of resources like memory for instance, deadlocks, infinite loops and also, in the context of real-time systems, scheduling anomalies. *Language-based systems* are state of the art that mitigate memory leaks and memory violations. “A language-based system is a type of operating system that uses language features to provide security, instead of, or in addition to, hardware mechanisms. In such systems, code referred to as the trusted base is responsible for approving programs for execution, assuring they cannot perform operations detrimental to the system’s stability without first being detected and dealt with” [47].
Thus, there is a need to improve the *temporal predictability* of dependable systems.

The benchmark for the presented generic software framework for implementing dependable data driven embedded systems are other *language-based systems* such as Singularity [17] by Microsoft Research or the OVM project [3] by Boeing corp. and Purdue University. Since Singularity was not primarily targeted to realtime applications, the OVM project based on the Realtime Specification for Java [5] will be used as a reference throughout this thesis. Most comparisons will not refer to OVM specific aspects, but to concepts introduced by the Realtime Specification for Java and thus relevant for the whole class of systems implementing this standard.
Another class of systems that are referenced in this thesis are the linux based realtime operating systems, LynxOS and QNX. These systems are not language-based but still offer interesting conceptual insights. Also of conceptual interest is the Microsoft Task Parallel Library (TPL) for .NET [37].

Temporal predictability is probably the most important property a realtime system is supposed to have. According to the Realtime Specification for Java [5], it involves resolving the following OS challenges:

- Scheduling
- Memory Management

- Synchronization
- Resource Sharing
- Asynchronous Event Handling
- Asynchronous Transfer of Control

State of the art operating systems all face these challenges in one way or another. Crucial to the application programmer in the context of real-time is the ease of achieving temporal predictability. The end to end quality of an application certainly benefits from easy to predict programming primitives. The famous Mars Rover case [21] has exemplified the potential risk of runtime systems that expose unnecessary complexity to the application programmer.

There are basically two different control abstractions for synchronizing threads. The most common one is lock based and the other is software transaction memory (STM) based. OVM proposes STM based *preemptable atomic regions* [32] for instance.
Lock-based synchronization control suffers from programmability, scalability, and composability challenges [22], whereas STM based synchronization control promises to alleviate these difficulties. Additionally, significant advantages include the absence of deadlocks and the fact that no other priority inversion avoidance technique is needed [32]. Unfortunately, there are also major limitations of STM based synchronization control: Low level I/O operations are a challenge because these cannot be undone in general. Hence a system cannot be built on STM based synchronization control only. A hybrid solution with lock based synchronization control is inevitable.

Since unification of low level hardware programming and application programming was the driving idea behind the presented OS, we have decided to choose the lock based synchronization control approach. To achieve temporal predictability with a completely uniform programming model we have only introduced locked based synchronization control. A uniform programming model is considered as more valuable than the mentioned advantages of STM based synchronization control. Especially since this thesis also presents improvements regarding the drawbacks of locked based synchronization control, namely the required priority inheritance protocol and deadlock avoidance.

Regarding temporal predictability, the mentioned state of the art systems include one or more suboptimal elements listed in the following enumeration:

- a first class interrupt handling system that works according to its own scheduling rules.
- a first class garbage collector scheduled according to its own rules.
- different types of concurrent activities.

This thesis shows how better predictability is achieved with a single precise and concise set of scheduling rules, a single heap space and a single type of activity.

1.3 Research Context

Research in Dependable Systems has an especially long tradition at the ETHZ. Different approaches have been taken such as strongly typed programming languages like Pascal, Modula and Oberon, and systems entirely written in such high level languages. Such systems benefit from strong type-safety in two respects: Multiple address spaces for shielding different processes against each other are not required, thus simplifying the operation of a system considerably. A second benefit is the use of type information for managing heap data structures. Garbage collection relies heavily on type meta information. Such systems are also called *fully managed*.

The outcome of Project Oberon [14] has been an incubator for several OS projects, including the runtime system and programming language presented in this thesis. It has been derived from the *Active Object System* [38] and is targeted to data driven real-time applications. An earlier project by Roberto Brega [6] has already shown the potential of a fully managed Oberon based system for real-time industrial controls. In hindsight, using an Oberon derivative for building dependable system technology was a very reasonable choice.

1.4 Contributions

This thesis' higher order contribution is based on a programming language and runtime system co-design. It takes advantage of a highly integrated runtime system and programming language environment that was designed by

exploring the full degrees of freedom that co-designing a system and a language offers. The presented co-design is somehow a counterpoint to the general trend of compiling different programming languages on the same runtime system such as, for instance, the Microsoft runtime does.

The first contribution aims at fostering predictability:

- The *unified activity model* helps the programmer to better understand the overall temporal behavior of the system. The system does not host any first class citizens with particular privileges, all (concurrent) activities are treated uniformly. Hardware level programming is done according to exactly the same principles as application level programming. Interrupt handling, finalization and garbage collection obey exactly the same rules as ordinary application activities do. Applying the priority inheritance protocol uniformly to all concurrent activities enables application programmers to better anticipate the behavior of the underlying system.

- The runtime system and programming language environment supports *data driven scheduling* rather than time-driven scheduling. Non-effective context switches triggered by traditional timeslicing are conceptually avoided. Arriving data-packets are used as a natural “pace-maker” for scheduling the basic activities, which saves battery power overall.

- A proposal on how to employ *priority inheritance as a thread orchestrating tool* rather than just as a means to overcome a well known scheduling anomaly. Priority inheritance enables conceptually sound scheduling of system services like garbage collection and interrupt handling.

Deadlocks, infinite loops and scheduling anomalies have been mentioned in 1.2 as a major threat for dependable systems. The following contributions help to disarm those:

- A set of hands on programming patterns to build systems *without deadlocks by mutual exclusion or cyclic waiting.* Our approach does not rely on type systems or annotations, but it guides the way applications should be structured.

- It has been verified by formal methods that a *scheduling anomaly due to priority inversion* is not going to happen. This is a step ahead compared

to conformance testing proposed by [53]. This contribution is based on joint work with [45].

- An essential aspect of real-time programming, namely *time-wise constrained synchronization* among cooperating threads has been reviewed. Time-wise constrained synchronization is useful to detect infinite loops for instance.

Finally, a data driven *sample application* that takes advantage of a dependable application aware OS. This medical IT application has been fully implemented and field tested.

1.5 Overview

This text mainly discusses the addressed class of problems, the way the problems are meant to be solved, the programming language and runtime support needed to formulate an elegant solution to the addressed problems and last, but not least, one concrete solution to a problem instance. Chapters 2 to 6 will refer to the contributions listed in 1.4 where appropriate.

Chapter 2 presents the envisaged class of problems to be computed, the problems' characteristics and a suitable computing model to solve these problems.

Chapter 3 introduces some enhancements to the programming language Oberon needed to implement the proposed computing model in chapter 5. These enhancements comprise a unified event handling and temporal constraints on blocking primitives. Instructive code samples will show how to use these newly introduced programming primitives.

Chapter 4 explains the scheduling support provided by the runtime system needed to implement the proposed computing model of chapter 5.

Chapter 5 lists relevant programming patterns recommended to achieve the results announced in 1.4. It will be demonstrated how to fit hardware programming into the proposed computing model. An example about employing priority inheritance as a tool for structuring a set of threads is given by listing 8.2. Another topic is how to achieve deadlock avoidance.

Chapter 6 offers a look behind the scenes by revealing selected implementation details of the runtime system. In particular, it will focus on a provable priority inversion implementation.

Chapter 7 presents three practical use cases. They are meant to demonstrate the benefits of the presented runtime system.

Chapter 8 evaluates the presented generic software framework for implementing dependable data driven embedded systems with other state of the art systems mentioned in chapter 1.2.

Chapter 9 concludes this thesis and suggests directions for further research.

Chapter 2

Realtime Processing Challenges

This chapter outlines the computational challenges to be addressed and how these are approached. It introduces the *active objects computing model* that Realtime Oberon has been designed to support and explains how the model fits the envisaged application domain.
The application domain is the class of computing systems targeted to monitor 24/7 objects of any kind. One has to take into account the intrinsic complexity of the observed object that determines the computational complexity of the algorithm used. Thus, the computing model needs to take into account the expected workload in order to not oversimplify the problem. This chapter will explain how the presented model matches the expected workload in the case of observing complex physiologic systems, like humans.

Structuring a real-time system requires knowledge about all (concurrent) activities in the system. Activities requesting hard deadlines have to be identified. The better the identified real-time activities are isolated from non-real-time activities, the easier they are to deal with. A fine grained decomposition of all activities is key to finding a feasible schedule that satisfies realtime constraints.

A general approach to decompose a system's intrinsic activities into a set of manageable threads is to reason about task and pipeline parallelism. Task parallelism is about decomposing a single task into multiple subtasks computed in parallel, pipeline parallelism is about decomposing a single subtask into multiple subsequent steps. With pipeline parallelism in mind we will structure the task of signal processing in the next section. The goal is again to isolate the steps that require hard real-time guarantees in order to be able to define a fine granular and feasible schedule.

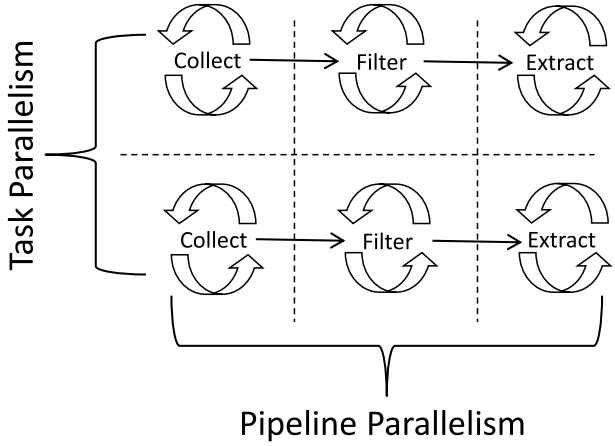

Figure 2.1: Task and Pipeline Parallelism

2.1 Structuring the Task of Realtime Signal Processing

Monitoring complex real world objects primarily involves digital signal processing. One possible approach to model digital signal processing is to decompose the overall task into five subsequent subtasks:

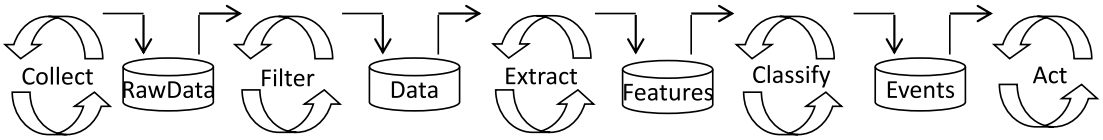

Figure 2.2: Generic Data Processing Pipeline

1. The first task collects raw data and assembles it to a data buffer. This task is usually triggered by hardware interrupts that are signaling the availability of new chunks of raw data. Polling would be an alternative option.

2. The second task pre-processes the raw data. For instance, it could down sample the raw data if the data acquisition rate is higher than appropriate.

3. The third task extracts features from the filtered data. For example, it detects peaks in the signal.

4. The classifier task processes the previously detected features. The classifier decides whether a detected feature has any relevance in the current

context or not. For instance, if the previously detected peak's magnitude reaches a particular threshold, then an event is raised and forwarded to the rightmost actor task.

5. Actor tasks take actions according to the received events.

The modular decomposition of the signal processing task into these five subtasks has the following advantages:

- The model enables fine grained scheduling. Especially when computing resources are scarce.
- The model helps to avoid redundant computing where different tasks rely on the same intermediate results.

The next chapters outline the computational complexity to be expected when monitoring physiologic systems.

2.2 Computational Challenge

"Physiologic systems in health and disease display an extraordinary range of temporal behaviors and structural patterns that defy understanding based on linear constructs, reductionist strategies, and classical homeostasis" [13]. It is well known that physiologic systems have non-linear and self-similar properties. Both properties will be explained in a qualitative way, each with a well known phenomenon at hand.

2.2.1 Non-Linearity Challenge

A linear system's output is proportional to its input. And if there is more than one input, then the output is equal to the sum of the responses that would have been caused by each input individually. However, linearity may not be assumed when analyzing a physiologic system. Figure 2.3 shows an instance of a non-linear relation.

One would naively assume a linear relation between the pulmonary function and the heart function. If the respiration is in a steady state, one would

assume that the heart activity is also in a steady state. However, “visual inspection of the time series in 2.3 reveals dramatic differences in the temporal structure. The time series from a healthy person reveals a complex pattern of nonstationary fluctuations. In contrast, the heart rate diagram from people with sleep apnea shows a much more predictable pattern [...]. Both the complex behavior in the healthy case and the sustained oscillations in the pathologic case suggest the presence of nonlinear mechanisms.” [13]

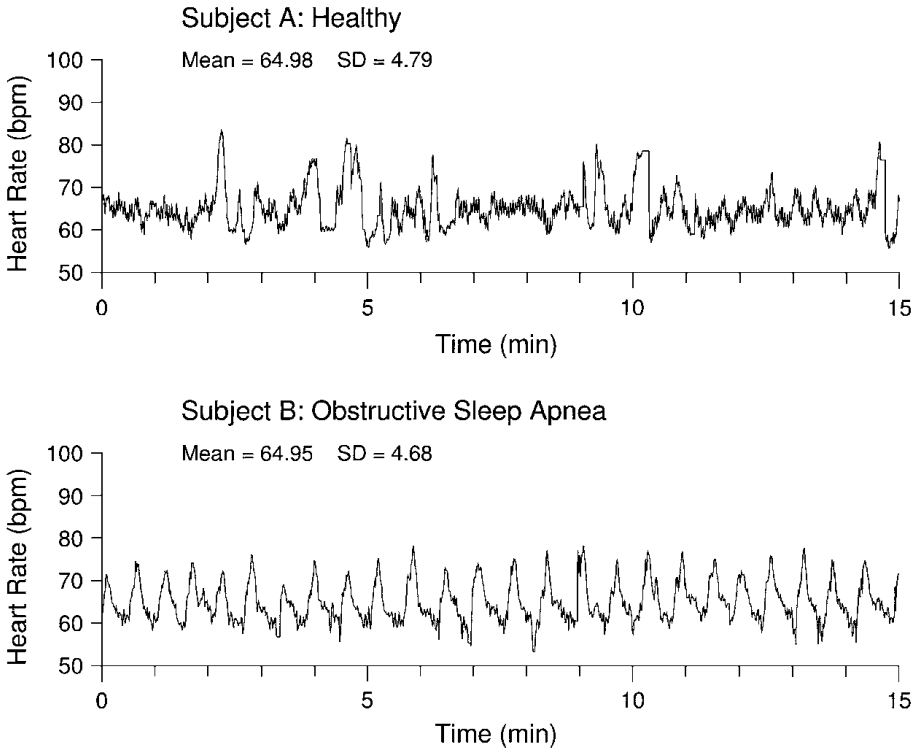

Figure 2.3: Cardiopulmonary nonstationary fluctuaions [13]

2.2.2 Multiscale Challenge

As mentioned, there is strong evidence that the expected signal has *self-similar* properties. A self-similar structure looks similar on different spatial or temporal scales. The tracheobronchial tree depicted left in figure 2.4 is a fractal structure that reproduces itself at different spatial scales. A temporal example of fractal behavior is Internet traffic. The data volume over time is self-similar, its standard deviation does not converge toward zero when increasing the considered time interval. It shows growing peaks on a daily, weekly and even on

a yearly basis [42].

The heart rate regulation process is another interesting case: A healthy regulation generates fluctuations on different time scales that are statistically self-similar according to the graph on the right in figure 2.4. There are three time slices of three different orders of magnitude, which look quite similar from a morphological point of view. “Findings from life-threatening conditions, such as chronic heart failure, cause the breakdown of fractal correlations. Abrupt transitions to strongly periodic dynamics are observed in many other pathologies, including at high altitudes and with obstructive sleep apnea, sudden cardiac death, epilepsy, and fetal distress syndromes, to name but a few.” [13] Obviously information is encoded on different time scales in the signal. A holistic analysis requires computations on all timescales in parallel in order to detect a potential breakdown of fractal correlations indicating pathologies.

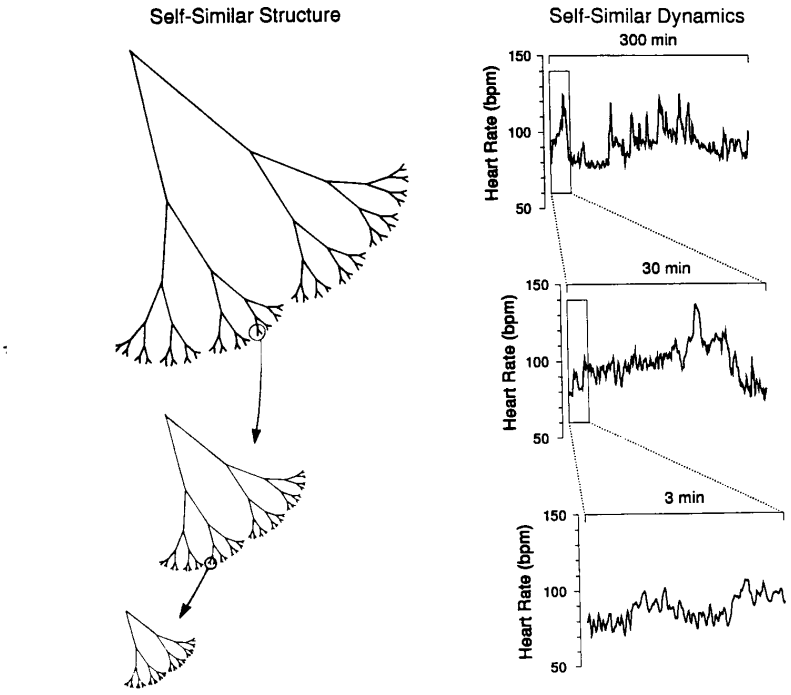

Figure 2.4: Heart Rate Self-Similarity [13]

2.3 Active Objects Computing Model

As a consequence of the specific characteristics mentioned in the previous section, a computing model is needed for our class of applications that is able to respond to a multitude of events arriving at unpredictable times and in an unpredictable order. [38] was a promising basis to start with. Its model allows event processing based on active objects. Autonomous active objects engage other autonomous active objects asynchronously in activities via events or messages. Active objects are connected via thread safe buffers used for exchanging data.

Briefly said, an active object;

- runs to completion in the sense that it releases the processor only while waiting for the next event to be processed,
- encapsulates the state of event processing, and
- engages other autonomous active objects asynchronously.

A more formal description of the computing model is: A set of active objects are connected through threadsafe buffers, which are used for synchronization purposes. The following rules apply on the active objects / buffer graph:

- Each active object plays the role of a producer and / or consumer. A producer fills the buffer, a consumer empties it.
- Each active object manages one or more buffers. Each buffer is either operated as a producer or as a consumer, but never both.
- circular producer - consumer - producer chains are forbidden - a cycle free topology thereby avoids deadlocks by design.

The next section will demonstrate how this computing model can cope with the expected workload caused by the likely computational complexity described above.

2.4 Scalability through Modular Composition

This section explains how to compose cascaded and / or parallel modular processing chains based on the five step model introduced in 2.1. Similar to the previously mentioned sleep apnoea example, we can demonstrate how our computing model scales.

As mentioned, there are two related indicators for detecting sleep apnoea reliably: frequent respiration stalls accompanied by the breakdown of fractal correlations of the heart rate. Figure 2.5 sketches a fine grained pipeline that models the corresponding computation. Note that the buffers connecting the individual threads have been omitted for the sake of simplicity.

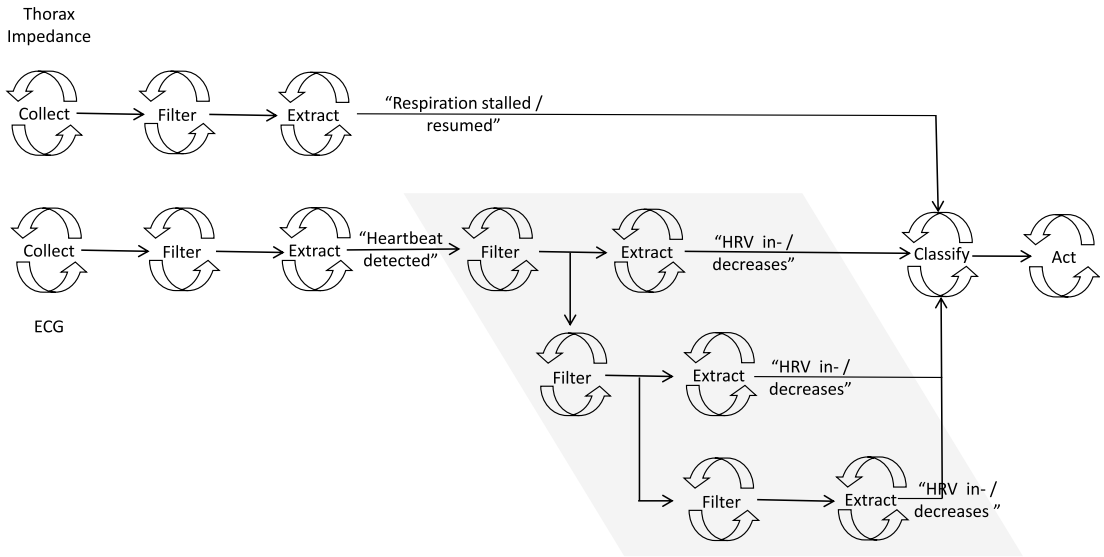

Figure 2.5: Parallel Pipelines

The two input signals are: the thorax impedance used for detecting respiration stalls and the ECG to calculate the heart rate. The extraction process in the thorax impedance pipeline extracts "respiration stalled" and "respiration resumed" features, which are classified by the subsequent classifier thread. The first extraction thread in the ECG pipeline detects heart beats.
Then the heart rate's self-similar property enters the scene: The *Heart Rate Variability (HRV)* expresses the amount of fluctuation in a heart rate signal like the ones shown in figure 2.3. According to figure 2.4, the heart rate variability has to be computed on several (in this example three) time scales in parallel, in order to not miss the point in time when the fractal correlations of the heart rate

start to break down. This is done by three additional filter / extract pipelines that work in parallel. Changes in the heart rate variability on any time scale are reported to the classification thread together with corresponding features.
The non-linear property of the computation pops up at the classification thread on the right in figure 2.5. In general, the reported events are not linearly separable, but there are different algorithmic approaches that are suited to deal with nonlinear classification.

2.5 Conclusions

In summary we can state the following pros of our modeling approach:

- The proposed *active objects computing model* is sufficiently abstract to be useful for implementing complex signal processing systems.
- A high computational dynamic is likely when monitoring complex systems. The fine grained tasks allow fine grained realtime scheduling especially in cases of a computational overload.

The remaining challenges are:

- To provide a runtime system that supports smooth mapping of the proposed active objects computing model. All activities should be treated in a uniform way in order to guarantee predictability.
- Efficient scheduling of a large number of fine grained tasks.

The next two chapters introduce the programming language and runtime scheduling infrastructure necessary to guarantee efficient mapping of the proposed active objects computing model.

Chapter 3

Programming Language Enhancements

This chapter presents Realtime Oberon as an evolution of the original Oberon programming language.

Realtime Oberon uses a concurrency model based on *active objects*. Active objects are objects with an intrinsic thread of control. A short and informal introduction into the active object concept will be given below, a more detailed presentation has been done by Pieter Muller ([38] chapter 2) and Patrick Reali ([44] chapter 4).

The main reason for enhancing the existing language is Oberon's lack of expressiveness to cope with temporal constraints. A second incentive was to strengthen the static properties of the execution path along the module dependency graph. The unrestricted use of function pointers, delegates and virtual calls severely compromises static assertions on the execution order of code.

3.1 Concurrency and Synchronization in Programming Languages

Concurrent programming brings up three issues: parallelism, mutual exclusion, and synchronization. There are different approaches to what extent a programming language supports these three issues. Java and C# do only

address mutual exclusion for instance. Parallelism and synchronization are implemented with API support. C++ is completely concurrency agnostic, it does not provide support for any of the three concerns. Whereas Realtime Oberon supports mutual exclusion, synchronization and parallelism with only four basic constructs. At the end of this section, the motivation for including the statements in the programming language will be explained.

To develop an idea of how these four basic constructs are used, listing 3.1 shows an illustrative example. It shows a producer composed of a bounded buffer and an intrinsic activity. All “active objects” postulated in chapter 2.3 will be implemented similarly to this example.

Listing 3.1: Producer

```
Producer = OBJECT
 VAR buffer : POINTER TO ARRAY OF Item;
     notFull, notEmpty : BOOLEAN;

 PROCEDURE & Init();
 BEGIN
    NEW(buffer, initialSize);
 END Init;

 PROCEDURE Append(x:Item);
 BEGIN {EXCLUSIVE}
    AWAITCONDITION(notFull);
    (*insert item into buffer, set condition notEmpty*)
 END Append;

 PROCEDURE Remove():Item;
 BEGIN {EXCLUSIVE}
  AWAITCONDITION(notEmpty);
  (*remove item from buffer, set conditon notFull, return*)
 END Remove;

BEGIN{ACTIVE}
    WHILE (alive) DO
        AWAITEVENT SYSTEM.INTERRUPT, anInterrupt;
        (*Read item from hardware register, Append(item)*)
    END;
END Producer;
```

An active object such as the producer in 3.1 contains four parts: a number of variables, a constructor, a number of methods, plus a so called “body”. The body is enclosed in a BEGIN .. END block annotated with a modifier {ACTIVE } appended to BEGIN . This code block is executed by a dedicated intrinsic thread. Additionally, the initial priority may be specified in brackets as an add on to {ACTIVE}. This is how parallelism is supported by Realtime Oberon.
Another block modifier is {EXCLUSIVE}. It assures mutual exclusion, the variable buffer is therefore protected from concurrent accesses by appending and / or removing threads.
AWAITCONDITON stalls an invoking thread (at least) until the requesting boolean condition evaluates to true. It must be embedded in a BEGIN{EXCLUSIVE} .. END block.
And finally, an AWAITEVENT statement in the body synchronizes the intrinsic thread of the object with an event.

As already outlined, parallelism, mutual exclusion, and synchronization are supported differently by other programming languages like Java or C++. All introduced language constructs could be substituted by ordinary API calls. There are three reasons why {EXCLUSIVE}, AWAITEVENT, AWAITCONDITION and ACTIVE should be included into the language:

- An important design goal was to keep the OS Application Programming Interface free of *side effects* on other than the caller’s schedule. In contrast, the four statements *do* have side effects on other than the caller’s schedule. Very obvious ones in the case of ACTIVE when a new thread is created, hidden ones in the case of {EXCLUSIVE}, AWAITEVENT, and AWAITCONDITION mainly due to the priority inheritance protocol.

- Another reason are *static guarantees*. Critical sections tagged with { EXCLUSIVE} are always freed correctly, it is not left to the programmer. The compiler is also able to check that {AWAITCONDITION} is always enclosed by an {EXCLUSIVE} section.

- A third reason is *symmetry*. AWAITEVENT is a natural, symmetric complement to AWAITCONDITION. It would be inconsistent only to include one statement into the language rather than both.

Let us now take a closer look at the mutual exclusion construct.

3.2 Time Constrained Mutual Exclusion

In order to avoid data races, critical sections are protected by EXCLUSIVE blocks. "The EXCLUSIVE modifier on a statement block defines the block as a critical section of the immediately enclosing object instance or module." [38]

By specifying an optional upper limit in milliseconds, the timespan a thread is allowed to wait before entering the monitor is limited. If a thread was not granted access to the monitor within the specified timespan, the entire BEGIN{

Listing 3.2: Bounded Buffer

```
BoundedBuffer = OBJECT

 VAR head,num:LONGINT; buffer:POINTER TO ARRAY OF Item;

 PROCEDURE Append(x:Item; max:LONGINT);
 BEGIN {EXCLUSIVE(max)}
  IF(num # LEN(buffer)) THEN
   buffer[(head+num) MOD LEN(buffer)]:=x;
   INC(num);
  END;
 END Append;

 PROCEDURE Remove():Item;
 VAR x:Item;
 BEGIN {EXCLUSIVE}
  AWAITCONDITION(num # 0);
  x:=buffer[head];
  head:=(head+1) MOD LEN(buffer);
  DEC(num);
  RETURN x
 END Remove;

 PROCEDURE &Init(n:LONGINT);
 BEGIN
  head:=0;num:=0;NEW(buffer,n)
 END Init;

END BoundedBuffer;
```

`EXCLUSIVE(max)} (*statements*)END` block will simply be skipped.

There are three scenarios where time-limited monitor locks can beneficially be used:

- Listing 3.2 shows a bounded buffer in a producer / consumer scenario for UDP handling. Let us assume a low prioritized consumer removing UDP packets and a high prioritized producer appending UDP packets to the buffer. If the buffer is full `(num = LEN(buf))`, the UDP packets are simply dropped, which is perfectly consistent with the general understanding of UDP packet transfers. But what if the low prioritized consumer has been preempted for more than `max` milliseconds within the `Remove` method and the producer tries to append a new packet at the same time? The producer drops the packet implicitly by skipping the `BEGIN{EXCLUSIVE( max)} (*statements*)END`. That is again in agreement with the general understanding of UDP data transfer.

- Another application is coping with starvation. If a heavily used resource causes starvation, a thread that does not get access to the resource within a reasonable time frame may skip the questionable monitor lock and launch countermeasures.

- The third application is connected with runtime predictability: Handling the worst case execution time becomes much easier by setting time limits for entering a monitor.

- Finally, timely restricted monitor guards can be used for energy aware systems. An energy aware system optimizes its power consumption by dynamically adjusting the CPU clock rate. In order to do that, the system can exploit the information about timing constraints to dynamically adjust the clock rate.

3.3 Time Constrained Conditional Waiting

`AWAITCONDITION` is used for conditional synchronization. It blocks the invoking thread until a generic boolean condition becomes true. Time constraints can

also be applied. In addition to a generic boolean condition, an optional upper limit in milliseconds may be appended as an argument.

The formal idea is the following: If an invariant I of the associated exclusive region is valid before `AWAITCONDITION`, then the invariant I is still valid after `AWAITCONDITION` and additionally either the condition C or the timeout T, or both.

$$\{I\}, \text{AWAITCONDITION}(C \vee T), \{I \wedge (C \vee T)\} \tag{3.1}$$

Since condition C must be valid after `AWAITCONDITION`, condition C must be protected by the same monitor as `AWAITCONDITION` is. In other words, test and set operations on conditions must run mutually exclusive. Applied to listing 3.3: `state:=TimerAwake;` runs mutually exclusive with `AWAITCONDITION state # TimerSleeping, timeout;`. The second implication is that invariant I may not be invalidated by `AWAITCONDITION`. Thus the evaluation of condition C must not have any side effects. This restricts the semantics of the generic boolean expression C.

In fact, there are no compelling reasons for a time constrained `AWAITCONDITION` statement. The same semantics could be implemented otherwise, for example by timer call backs invalidating the questionable boolean condition. However, the “inverse programming” style induced by callbacks disturbs execution path predictability as mentioned earlier. The Timer Object example in listing 3.3 shows how time-constrained `AWAITCONDITION` leads to straight forward and compact code.

Listing 3.3: Timer Object

```
Timer = OBJECT

 VAR state:SHORTINT;

 PROCEDURE Sleep(timeout:LONGINT);
 BEGIN {EXCLUSIVE}
  state:=TimerSleeping;
  AWAITCONDITION state # TimerSleeping, timeout;
  state:=TimerFree
 END Sleep;

 PROCEDURE Wakeup;
 BEGIN {EXCLUSIVE}
  IF state = TimerSleeping THEN state:=TimerAwake;END
 END Wakeup;
```

```
  PROCEDURE &Init;
  BEGIN
   state:=TimerFree;
  END Init;

END Timer;
```

3.4 Time Constrained Event Based Synchronization

The AWAITEVENT statement was introduced to handle different kinds of events in a uniform way. The event types are:

- I/O hardware interrupts.
- timer events, usually also triggered by hardware interrupt requests.
- signals triggered by application threads .
- garbage events triggered by the garbage collector when an object has become unreachable and is about to be collected by the garbage collector. Garbage events are used for predictable finalization of objects.

In contrast to the AWAITCONDITION statement, AWAITEVENT does not have to occur in an EXCLUSIVE block. If it nevertheless occurs in an EXCLUSIVE block, the monitor lock is not implicitly freed when the invoking thread is passivated on AWAITEVENT. This is unlike the AWAITCONDITION boolean_condition case, where releasing the monitor lock is necessary to allow other threads to establish the awaited boolean condition.
AWAITEVENT statements are embedded in an EXCLUSIVE section in the following cases:

- Events are to be handled sequentially. The EXCLUSIVE section serializes threads waiting on the event. When an event has been raised and the handling thread leaves the EXCLUSIVE section, another thread can enter the section and wait for the next event.

- The waiting thread is supposed to inherit the priority of other threads trying to enter the monitor. An example will be shown later in the chapter on hardware programming.

AWAITEVENT statements are not embedded in an EXCLUSIVE section in the following case:

- Events may be handled interleaved. When a pool of threads is in charge of the same event type, an event handling thread may be launched before the previous event handler has finished.

The main reason for introducing a special event handling statement was rigorously eliminating all event handler registration services in the interest of execution path predictability.

3.4.1 Interrupts as Events

The benefits of a time constrained AWAITEVENT statement are exemplified by listing 3.4. It shows the body of a thread operating an external I/O board [49]. An external microcontroller regularly signals the availability of data by triggering an interrupt each time a chunk of data is ready for processing. The AWAITEVENT statement blocks the invoking thread until the general purpose I/O interrupt request zero (GPIO0IRQ) is raised for at most `timeout` [milliseconds]. If the interrupt is not raised within `timeout` [milliseconds], the external I/O board is stopped and restarted.

Multiple threads are allowed to wait for the same interrupt when interrupts are shared.

Listing 3.4: Interrupt Thread

```
Handler = OBJECT
VAR
    run : BOOLEAN;
    t : AosKernel.MiliTimer;
    maxTimeSpan : LONGINT;

  PROCEDURE RestartInputDevice();
  BEGIN
   (*restart it*)
  END RestartInputDevice();
```

```
    PROCEDURE ProcessInputData();
    BEGIN
     (*process it*)
    END ProcessInputData();

BEGIN{ACTIVE, PRIORITY(AosRuntime.Realtime)}
 WHILE(run) DO
  AosKernel.SetTimer(t, timeout);
  AWAITEVENT SYSTEM.INTERRUPT, PXA26x.GPIO0IRQ, timeout;
  IF AosKernel.Expired(t) THEN
   RestartInputDevice();
  ELSE
   ProcessInputData();
  END;
 END;
END Handler;
```

3.4.2 Timers as Events

Timer implementations are straight forward with the newly introduced `AWAITEVENT` statement. Its parameters are the event type, the interrupt involved and the maximum amount of time in [milliseconds] the invoking thread is meant to be passivated.

Listing 3.5: Timer Thread

```
WHILE (run) DO
 AWAITEVENT SYSTEM.INTERRUPT, PXA26x.TIMERIRQ, timeout;
 UpdateView()
END;
```

Listing 3.5 shows a thread releasing the CPU for at most `timeout` [milliseconds] before executing some periodic action. `AWAITEVENT SYSTEM.INTERRUPT , PXA26x.TIMERIRQ, timeout` is semantically equivalent to `AWAITCONDITION FALSE, timeout` but must not be embedded in an `EXCLUSIVE` block.

3.4.3 Self-defined Signals as Events

A new `SIGNAL` base type has been introduced. Signals are caught by `AWAITEVENT SYSTEM.SIGNAL, signal, timeout` statements. The first and second parameter specify the event type and the signal instance to be caught. The third optional parameter is again a time-out in [milliseconds]. If the timeout parameter is omitted, the invoking thread will wait infinitely long.

Listing 3.6: Signal

```
PROCEDURE Remove(): Value;
    VAR result: Value;
 BEGIN
  result := NIL;
  AWAITEVENT SYSTEM.SIGNAL, signal;
  result := divider.next.val;
  divider := divider.next;
  RETURN result
 END Remove;
```

Signals are triggered by invoking the `AosKernel.Signal(VAR signal: SIGNAL, counter: LONGINT)` Procedure. Listing 3.6 shows the Remove method of a lock free buffer. The invoking thread is blocked until the buffer contains at least one element. The corresponding `AosKernel.Signal` call is located in the buffer's Add method.

3.4.4 Finalizers as Events

One limitation of popular managed runtime systems like Java or .NET is deterministic finalization. According to [43] and [36] both runtime systems lack determinism in finalizing objects because there is no possibility to statically predict in what order objects are finalized. This leads to uncertainties not tolerable in dependable systems. The obvious approach to enforce predictability would be to perform finalization according to some global order based on the object graph. Unfortunately this would require expensive bookkeeping.

Our solution does not enforce a global finalization order, but it allows controlling the relative order of finalization where it is of interest. For example, in the case of file / writer composition. The writer is supposed to be finalized before the file, in order to flush buffered data while the file is still open. Listing

3.7 shows our solution based on an AWAITEVENT statement.

A form of resurrection is useful to recycle objects instead of destroying and reallocating them. An object is resurrected by reassigning it to a persistent root. Listing 3.7 sketches this for the case of an unused file. After invoking Finalize(), it is registered with a pool of reusable file objects. In contrast, the writer instance will definitely be collected.

Each logically connected object group, like the mentioned file / writer pair, is finalized by a dedicated thread. Partitioning the overall finalization task into different threads makes the system less vulnerable. If there was a single finalizer thread, a life- or deadlock in any finalizer method could block the overall finalization process indefinitely [36].

The AWAITEVENT statement again requires two mandatory parameters: the requested event type and the object instance to be monitored until it becomes garbage. An optional timeout could be appended as a third argument, however, there is no practical use for it.

Listing 3.7: Finalizer Thread

```
Finalizer= OBJECT

 VAR writer:AosFS.Writer;file:AosFS.File;

 PROCEDURE &Init(w:AosFS.Writer;f:AosFS.File);
 BEGIN
  writer:=w;file:=f;
 END Init;

BEGIN{ACTIVE}
 AWAITEVENT SYSTEM.GARBAGE,writer;
 writer.Update();
 AWAITEVENT SYSTEM.GARBAGE,file;
 file.Finalize();
 filePool.Recycle(file);
END Finalizer;
```

Chapter 4

Scheduling Approach

The scheduling concept is of importance for implementing the computing model proposed in chapter 2.3. The exact rules according to which the scheduler works determines the active object's behavior.

Runtime systems for embedded devices have to cope with periodic (on a fixed schedule), aperiodic (not on a fixed schedule but with a maximum frequency) and sporadic (frequency and timing not predictable) tasks. A scheduler's job is to dynamically find a feasible schedule without knowing the future workload in detail. In general, sporadic tasks challenge real-time schedulers most because of their unpredictable nature.

One of the very basic tradeoffs is *optimality* versus *predictability*. A scheduler is called "optimal" if it finds a feasible schedule whenever one exists. It has been proven that optimal schedulers exist, for instance for preemptable single processor systems. The drawback of optimal schedulers is their unpredictability in the case of computational overload. Optimality has not been a priority concern in the context of Realtime Oberon. Instead we present a non-optimal scheduler with predictable behavior in the case of overload. The presented scheduler fits systems that mostly have static set of threads of known dependencies, like in the use case sketched in section 2.4.

Importantly the priority inversion protocol has been smoothly integrated into the scheduling concept. In particular, the impact on the coexistence of real-time threads with non-real-time threads will be shown.
A second issue regarding the scheduling strategy is power awareness, which is an aspect of crucial importance in a runtime system used for operating battery powered devices. A system is said to be power proportional, if no power

needs to be wasted for computational overhead like time slice interrupts etc. Time slice interrupts periodically reevaluate the running process. If the reevaluation process takes no special action, then the consumed computing cycles simply have been wasted. With our approach, the power consumption is strictly proportional to the application's workload.

4.1 Preemption aware Scheduling

We present a non-optimal *fixed priority* based scheduler that remains predictable in the case of overload. It could be used to implement a *rate monotonic schedule*. With fixed priorities, it is easy to predict that overload conditions will cause the low-priority threads to miss deadlines, while the highest-priority threads will still meet their deadlines. In theory, the price to pay with non-optimal schedulers is that feasible schedules cannot be handled. However, the actual price will heavily depend on the specific use-cases.

The proposed scheduling model is straight forward. It allows easy anticipation of the scheduling behavior in different workload configurations. Each thread has an assigned *static priority* reflecting its importance compared to all other currently running threads. The more important a thread is, the higher its static priority. A thread can temporarily gain importance by inheriting another thread's higher priority (See chapter 4.2), but the static priority will always remain at its initial value assigned at creation time. Hence a thread's *current priority* is the maximum of its static priority and of all inherited priorities. At each of the eight transactions depicted in Figure 4.1, the scheduler always picks the thread of the highest current priority from the set of threads ready to run.

Figure 4.1 shows the state transition diagram. Context switches occur either

(1) by completion: The running thread releases the processor by either attempting to acquire a resource lock (1b), by waiting for a Boolean condition to be established (1c), by awaiting an event (1d) or by terminating (1a).

(2) or by preemption: Whenever a passivated thread of a higher priority than the currently running thread becomes ready. This *can* happen when the

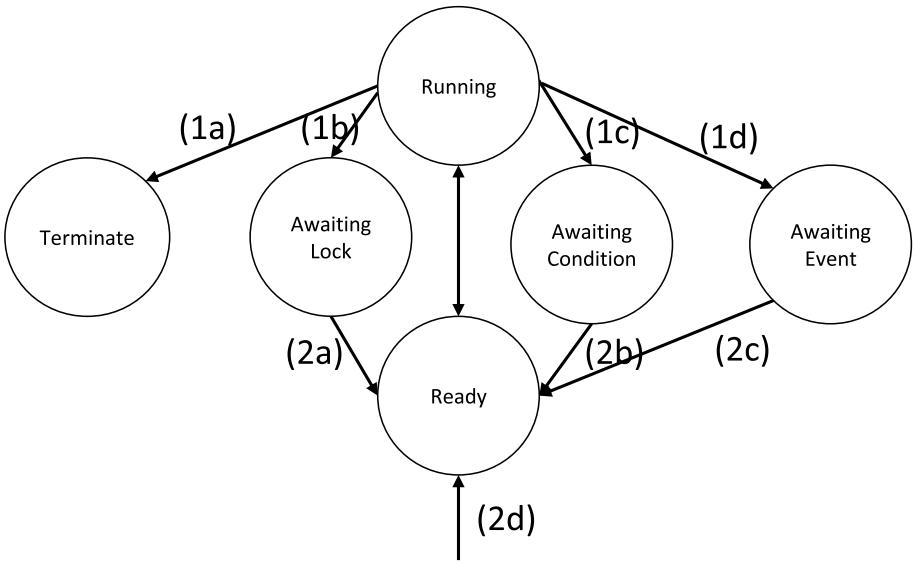

Figure 4.1: Scheduling State Machine

resource lock is transferred (2a), a boolean condition has become true (2b), an event has been raised (2c) or a newly created thread enters the stage.

As a consequence of the two rules stated above, no thread will ever be preempted by a thread of a lower or equal current priority. This must also be enforced by the interrupt handling model. An interrupt request must not interrupt the running process when there is no real impact on the schedule, namely when the corresponding event handling thread has lower priority than the currently running one. The justifications in favor of this strategy are:

1. It maintains a very simple scheduling model that allows anticipating the scheduling behavior easily;

2. It does not waste CPU time with effectless computations; and

3. It minimizes the number of context switches and thus maximizes the effectiveness of on chip cache memory.

Some remarks on time slicing: Figure 4.1 shows a transition from “Running” to “Ready” and from “Ready” to “Running” that could be triggered by time slicing. As a regularly raised time slice interrupt would substantially impede task proportional power consumption (in particular if an attempt to reallocate the

processor ends up with the same scheduled thread) our scheduling model does not support a *time slice triggered transition* from “Running” to “Ready”. Instead of being driven by the timer, the schedule will be driven by data packets arriving periodically. Data packets are delivered by a peripheral hardware controller, the DMA controller for instance. Thus scheduling decisions are implicitly triggered by incoming data packets that are supposed to be processed in time. Instrumenting the data flow as a trigger for scheduling decisions is perfectly compatible with the goal of task proportional power consumption. As long as no data is to be processed, the processor remains idle until the next data packet arrives.

The second concept normally found in popular scheduling models, but omitted in our approach is, *yielding*. From the runtime system’s point of view, yielding is a (semantically poor) strategy for processor sharing. The runtime system does not have any knowledge of when or why the yielding thread should be rescheduled. A concrete example taken from the current A2 [39] release:

Listing 4.1: Yield

```
REPEAT
 n := reader.Available();
 IF n = 0 THEN Yield();END;
UNTIL((n > 0) OR Expired(maxT));
```

From a programmer’s point of view, the idea is obviously to undergo periodical polling until there is either some data available on the stream or `maxT` time has elapsed. However, again, the scheduler has no clue when the yielding thread should be rescheduled and can only guess and try from time to time. Hence in our system the `REPEAT` loop is mapped to an `AWAITCONDITION data_received , maxT` statement that precisely indicates to the scheduler when the invoking process should be awoken. Furthermore, polling is not an option when power awareness is an issue.

4.2 Priority Inversion

The term “Priority Inversion” refers to a scheduling anomaly. A high priority thread blocked by an *unrelated* lower priority thread could be considered as the most general definition of priority inversion. An example is shown in figure

4.2. Let's assume a single processor system. A high priority thread is waiting on output produced by a related low priority thread. Suddenly an unrelated thread with medium priority preempts the low priority thread. Thus the unrelated medium priority thread blocks the high priority thread, as opposed to the general understanding of priority based scheduling. This is considered as a scheduling anomaly.

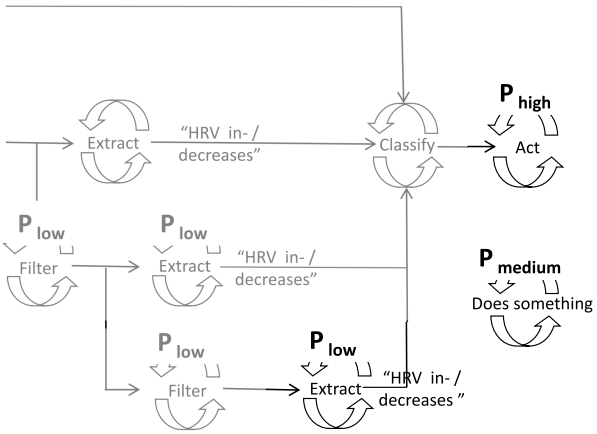

Figure 4.2: The sleep apnoea use case revisited

This thesis does not deal with this very general understanding of priority inversion. It only focuses on priority inversion induced by resource sharing. The next subsection will introduce this more specific manifestation of priority inversion.

4.2.1 Priority Inversion induced by resource sharing

Multiple threads concurrently operating on a shared set of resources may cause data races. Popular means to prevent data races are monitors or semaphores for instance. "Unfortunately, a direct application of synchronization mechanisms like semaphores or monitors can lead to *uncontrolled priority inversion caused by resource sharing*: a high priority thread being blocked by a lower priority thread for an *indefinite period of time*. Such priority inversion caused by resource sharing poses a serious problem in real-time systems by adversely

affecting both the schedulability and predictability of real-time systems." [48] According to the general understanding of mutual exclusion, a high priority thread may be blocked by a low priority thread if the latter is granted access to a critical section before the former arrives. This is a *controlled priority inversion*, since it is limited to a *finite period of time* until the low priority thread leaves the critical section. Unfortunately, two *unrelated* high and low priority threads, that do not directly share a logical or physical resource but are kept in unfortunate dependency by some middle priority thread, can also suffer from priority inversion . There is a concrete example shown in figure 4.3. This case is considered as an *uncontrolled priority inversion* since a higher priority thread is being blocked by a lower priority thread for an *indefinite period of time*.

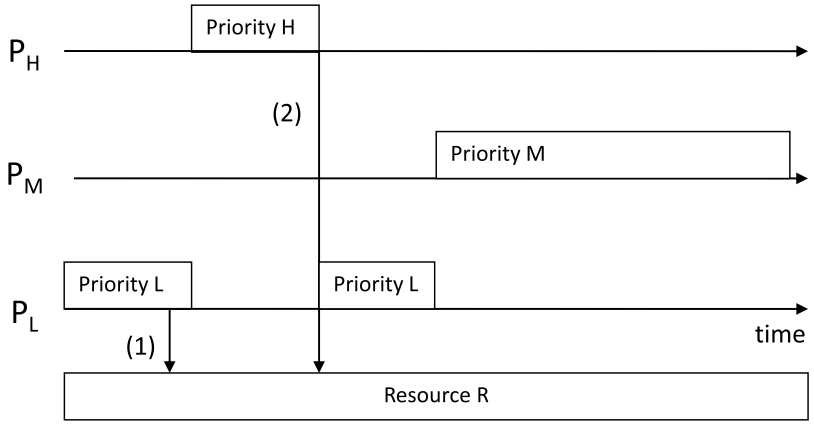

Figure 4.3: Priority Inversion Szenario

Initially, a low priority thread is running and acquires resource (R) at t = (1). Suddenly, a high priority thread gets ready and is scheduled. It also requests R at t = (2) and is therefore immediately passivated since R currently belongs to the thread with priority "low". This thread, in turn, resumes its operation until a thread with priority "medium" enters the scene and preempts the former thread. And that is exactly what should not happen: The thread with priority "high" is blocked by a lower prioritized thread although there are no common shared resources among the two threads.

One possible solution to alleviate *uncontrolled priority inversion* is *priority inheritance*. The thread with priority "low" should be raised to priority "high", in order not to be preempted by the thread with priority "medium". It should run with priority "high" until it releases R in favor of the high priority thread. The

priority inheritance protocol makes sure that a thread will be blocked, at most, as if it is executing the questionable monitor itself.

4.2.2 Priority Inheritance Protocol

Priority Inheritance is one possible method to cure the priority inversion anomaly. It is supported by most popular operating systems. Originally, priority inheritance was proposed by Lui Sha, Ragunathan Rajkumar and John P. Lehoczky [48]. We have integrated the Priority Inheritance Protocol (PIP) in our state / transition diagram introduced in the previous chapter.

The transition diagram in figure 4.4 shows the implemented PIP adaption. A solid arrow denotes a thread waiting to access the monitor. A dashed arrow denotes a thread accessing the monitor. A dotted arrow denotes a thread about to leave the monitor.

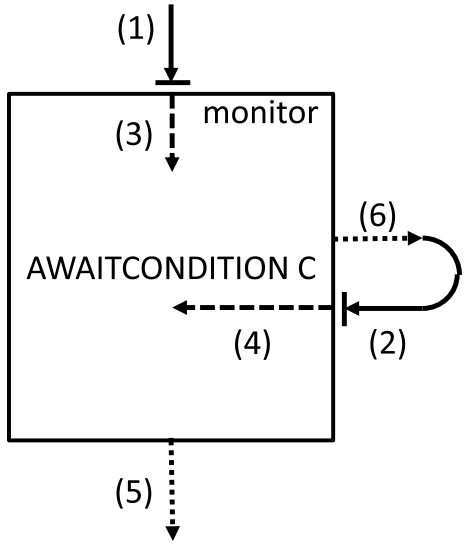

Figure 4.4: Transitions affected by the priority inheritance protocol

A thread potentially propagates its priority to other threads while it is *blocked* by the monitor (Arrow (1) or (2) in figure 4.4):

(1) A thread blocked on an `{EXCLUSIVE}` because another thread currently owns the requested monitor, will propagate its priority to the owner, if the blocked thread currently has a higher priority assigned than the monitor owning thread.

(2) A thread blocked on an `AWAITCONDITION C` statement due to the not set condition C, will propagate its priority to the monitor owning thread, if the blocked thread currently has a higher priority assigned than the monitor owning thread. The motivation for this transition is less obvious than case (1). An example to illustrate the idea: A thread T_{High} has been passivated on an unestablished condition C. In order to avoid a "low" prioritized thread T_{Low} that is just about to establish condition C within the monitor being blocked by any other thread with "medium" priority, the "low" prioritized thread T_{Low} must inherit the priority from T_{High}. One could say that, a thread awaiting a condition becoming true is continuously trying to re-enter the monitor and thus boosts the priorities of all threads entering the monitor to the priority of the awaiting thread itself.

A thread potentially inherits priorities from other threads after it has *entered* the monitor (Arrow (3) or (4)in figure 4.4):

(3) When a thread is granted access to the monitor, it inherits the maximum priority of all threads having a stake in this monitor, in particular the threads waiting for a condition being established that is associated with the monitor.

(4) Threads waiting for a boolean condition are put in the ready queue when the condition has been established. Following the so-called "egg shell" model, threads already in the monitor (passivated on an `AWAITCONDITION` statement) are preferred against threads trying to enter the monitor externally. Thus, such threads will get the maximum priority of a) all threads trying to enter in the monitor, b) all threads already in the monitor and waiting on a condition becoming established and c) its own static priority.

A thread releases its inherited priority after it has *left* the monitor (transition (5) or (6) in figure 4.4):

(5) If a thread leaves the monitor, it releases its inherited priority.

(6) If a thread is blocked on a false condition, it implicitly releases the monitor in order to allow another thread to establish the questionable condition. Thus it also releases its inherited priority.

The presented PIP adaption has been validated with two generic examples outlined in chapter 5.1.2 and chapter 6.4. Especially the generalization to condition awaiting threads has proved highly beneficial.

The correctness of the PIP is evident because each thread strictly obeys the three step protocol “blocked” - “entered” - “left”. According to figure 4.4, every execution path through the monitor repeats this sequence (n + 1) times, with 0 $<=$ n $<=$ the number of unsatisfied conditions a thread encounters within the monitor.

Two final remarks: First, note that a thread’s priority may be raised multiple times within a monitor when new threads also request the monitor.
Second, a thread is subject to priority adjustments in both directions at any point in time with no respect to its state. This must be the case because priorities are propagated transitively among threads.

4.3 Case Study

This section explains the application of our scheduling model to the computing model introduced in chapter 2.3. The sleep apnoea use case already introduced in the previous chapter will again serve as an exemplary application. Let us assume that three statically assigned priorities exist: P_{high}, P_{medium} and

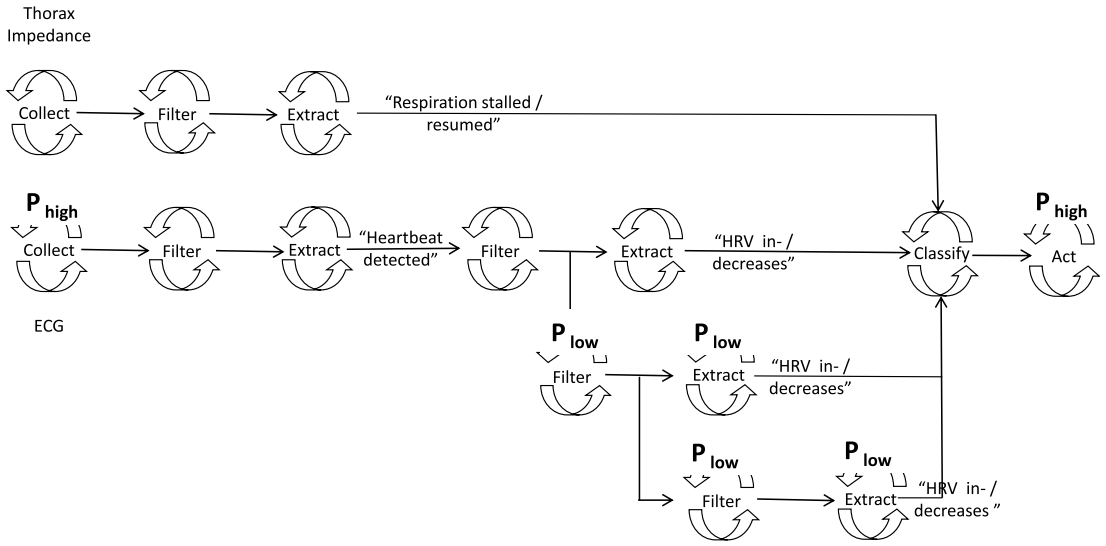

Figure 4.5: sleep apnoea use case revisited

P_{low}. A system designer has to assign a static priority to each thread according to its global importance. Intuitively, P_{high} would be assigned to the actor thread most right in figure 4.5, because taking action is inherently important after an event has been detected. The ECG Collector thread leftmost in figure 4.5 is another thread that is supposed to run with priority P_{high}, since hardware buffers not handled in time cause data losses, and the ECG analyzing algorithm is very sensitive to such losses. Therefore raw ECG data must always be buffered. All threads involved in calculating the second and third order heart rate variability fluctuation run at priority P_{low} due to some built in redundancy from the application point of view. All other threads run at P_{medium}.

In the case of a temporary overload, the system degrades in a controlled way. The system degradation process under overload is completely transparent and predictable at programming time because scheduling decisions are taken on a global priority scheme rather than on the basis of a local cost function.

This example shows how to advantageously apply the fine grained priority based scheduling model. However, the feasibility of the proposed model is subject to some constraints. It requires:

- A static set of threads and static dependencies among them.
- A manageable number of threads. The scheduling model does not scale well in terms of a large number of threads.

Chapter 5

Patterns and Paradigms for Signal Processing Applications

Event-driven programming requires a distinctly different way of thinking than conventional sequential programs. Most event-driven systems are structured according to the *inversion control* principle [46]. The control resides in some event handling machinery built deep into the system. From an application standpoint, control is inverted compared to a traditional sequential program. However, this is explicitly not the case for Realtime Oberon. In our model it is more about *distribution of control* rather than inversion of control. Control is still exercised by active objects, rather than by some underlying infrastructure tier. Inversion of control frameworks are implemented with up calls via delegates or function pointers. Extensive use of dynamically configurable up calls weakens static properties and deterministic behavior of a system. Yet static properties are of paramount importance in dependable systems.

5.1 Hardware Level Programming

Hardware is usually abstracted by an explicit hardware abstraction layer (HAL). This layer is formalized by a software interface, which is supposed to be implemented by so called “device drivers”. Drivers typically run in kernel mode and provide services for their clients, either applications or system services. While the software interface is typically generic in nature, general purpose operating systems are configured at installation time with specific device drivers; a popular deployment concept in the Personal Computer domain. All peripherals

except processors are operated by device drivers. This set up is undoubtedly one of the key concepts underlying the success story of IBM compatible personal computers. This concept has led to a wide variety of peripheral devices offered by many different manufacturers. “Plug and play” is a conceptual evolution that allows adding and removing peripheral devices on the fly.
With embedded systems, genericity does not have the same importance as with general purpose personal computers, mainly because embedded devices are usually designed as single purpose devices, and there is no compelling demand for configuring embedded systems.

There is strong evidence that writing reliable, configurable software for an environment with high diversity is very difficult in the embedded field. This is exemplified by the smart phone market, which tends towards highly standardized platforms. Microsoft for instance has so far failed to duplicate the concept of customizable operating systems to the smart phone market. Running a high diversity of devices based on the same OS kernel did not really improve productivity as expected, since all applications must be tested on all configurations to ensure the expected quality, which is a somewhat infeasible process. The latest Windows Phone 7 OS runs only on a concisely standardized runtime environment [34].

The alternative Hardware Abstraction paradigm presented in this thesis emphasizes clean integration of second level interrupt handling and application processing. Moreover, it lays the groundwork for seamless integration and interaction of application and hardware driven threads. Interrupt handling threads are abstracted as ordinary data producer and consumer threads running in user mode. The tool for implementing such threads is the `AWAITEVENT` statement introduced in chapter 3.4.1.

5.1.1 Unified Interrupt Handling

A common approach to interrupt handling is to install a dedicated interrupt handler thread for each interrupt vector. Whenever an interrupt is raised, the dedicated thread is scheduled. From the computing model perspective introduced in chapter 2.3, a one to one relationship between a logical task and an interrupt handler thread is highly desirable. Unfortunately such a scheme is not compatible with dedicated interrupt handlers.

The logical task of grabbing data from an external I/O board can easily

involve up to three threads: An external I/O board interrupt thread signaling when there is data available, a DMA interrupt thread handling the data transfer itself and a collector thread orchestrating the two interrupt handling threads. Figure 5.1 shows a UML sequence diagram illustrating how a data transfer is likely to be performed.

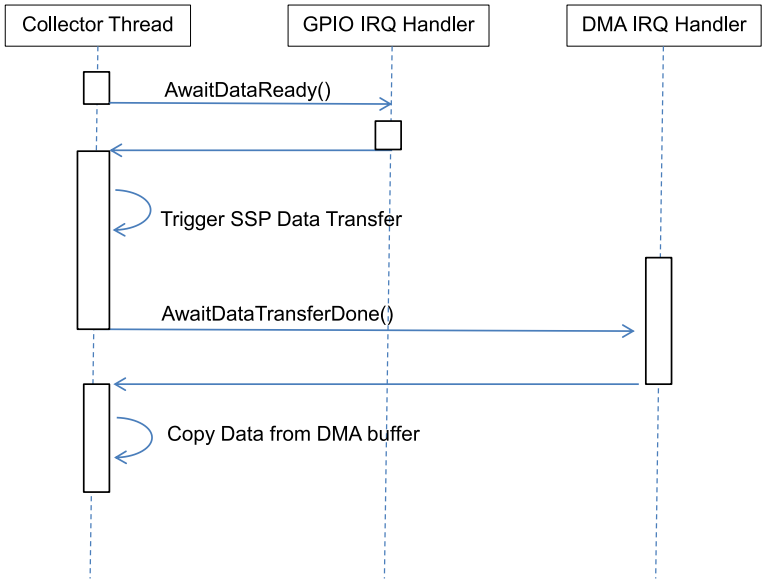

Figure 5.1: Collector Thread Interactions

A more elegant solution synthesizes the three threads into one single thread performing the data collection task. Listing 5.1 outlines an active object performing the entire control task. It pools two interrupt handlers into one single thread that first waits until the external I/O board signals that some data is available, then triggers the transfer via the Synchronous Serial Port (SSP) and finally waits until the DMA controller has completed its operation in order to copy the newly arrived data into a buffer.

Listing 5.1: Pooled Interrupt Handling Thread

```
BEGIN{ACTIVE, PRIORITY(AosRuntime.Realtime)}
 WHILE(run) DO
  (*Await Data Ready*)
  AWAITEVENT SYSTEM.INTERRUPT, PXA26x.GPIO0IRQ;
  TriggerSSPDataTransfer();
  AWAITEVENT SYSTEM.INTERRUPT, PXA26x.DMAIRQ;
  CopyDataFromDMABuffer();
 END;
```

```
END Collector;
```

Such interrupt handling threads are not system-layer specific in any respect. They are tightly coupled with application threads and run in user mode with the benefit that all rules regarding scheduling and priority inheritance introduced in the previous chapters also apply to interrupt handling threads.

5.1.2 Leveraging Priority Inheritance as a Tool

Let us revisit the producer object introduced in chapter 3.1 and let us assume the following scenario: a low priority producer putting elements in a buffer and a high priority consumer taking elements from it. If the buffer is empty, the low priority producer shall automatically inherit the consumer's high priority.
Listing 5.2 shows how this is achieved by specifically exploiting the mechanism of priority inheritance. Contrary to the producer implementation in 3.1, the `AWAITEVENT` statement and the data handling part located in the object's body have now been moved to an `EXCLUSIVE` section. Thus the producer inherits the consumer's priority while the consumer is awaiting a `newItem` in `PROCEDURE Remove`. This is an example of how priority inheritance can serve as a thread orchestrating tool. We consider this upgrade of priority inheritance from merely curing a scheduling anomaly to a thread structuring tool as an important contribution of our work as mentioned in chapter 1.4.

Listing 5.2: Producer revisited

```
Producer = OBJECT
 VAR buffer : Unbounded_Buffer; (*thread save, non blocking*)
     newItem : BOOLEAN;
     item : Item;

 PROCEDURE & Init();
 BEGIN
    NEW(buffer);
 END Init;

 PROCEDURE Remove():Item;
 VAR item : Item;
 BEGIN
  item = buffer.Get();
  IF (item = NIL) THEN
    BEGIN{EXCLUSIVE}
        newItem := FALSE;
        AWAITCONDITION(newItem);
        item =  buffer.Get();
    END;
  END;
  RETURN item;
 END Remove;

BEGIN{ACTIVE}
    WHILE (alive) DO
        BEGIN{EXCLUSIVE}
            (*runs potentially with inherited priority*)
            AWAITEVENT SYSTEM.INTERRUPT, anInterrupt;
            (*Read item from hardware register*)
            buffer.Put(item);
            newItem := TRUE;
        END;
        (*runs with static priority*)
    END;
END Producer;
```

5.2 Multicast Synchronization

The purpose of signals is synchronizing a set of threads. Monitors in combination with AWAITCONDITION provide signaling, as the producer example in

3.1 demonstrates. Producer and consumer are synchronized on condition `notEmpty`. However, there are two side effects the programmer has to be aware of:

- The producer and consumer thread propagate their priority between each other when entering into the monitor that encloses the boolean condition.
- Only one thread resumes when the condition becomes true because `AWAITCONDITION` is required to be enclosed by a monitor according to 3.3.

We have introduced a general data type `SIGNAL` in our model to offer an alternative to condition based signaling.
`SIGNAL` is an abstract datatype with two operations defined on it, one for receiving (catching) and the other one for sending (raising) signals. `AWAITEVENT SYSTEM.SIGNAL, signal` catches instance `signal`. Threads raising the instance `signal` invoke the API call `AosKernel.Signal(signal, inc)`.

In order to formalize the semantics of signals, one can view a signal as an *integer lease counter* with a blocking decrement and non-blocking increment operation defined on it. This lease counter ranges from 0 to `MAX(LONGINT)`. `AWAITEVENT SYSTEM.SIGNAL, signal` decrements the lease counter by one. The API call `AosKernel.Signal(signal, inc)` increments the assigned lease counter according to the following rule:

$$\text{lease counter}_{signal} = \begin{cases} \text{lease counter}_{signal} + \text{inc} & \text{if inc} > 0 \\ \text{lease counter}_{signal} & \text{if inc} \leq 0 \end{cases} \tag{5.1}$$

If the lease counter is equal to zero, `AWAITEVENT SYSTEM.SIGNAL, signal` blocks the invoking thread until a peer thread raises `signal`.

Signals are used like counting semaphores. A practical use case for this will be introduced in chapter 6.3: Synchronizing threads on lock free data structures.

Since `AWAITEVENT SYSTEM.SIGNAL, signal` must not be enclosed by a monitor, so that;

- signal raising and catching threads do not propagate their priority to each other per default.

- general signals based on `SIGNAL` allow multicast semantics, in contrast to condition based signals. More than one thread may resume simultaneously.

5.3 Real-Time Programming

The Real-Time Specification for Java [5] uses the following definition of real-time computing:

> The programming environment must provide abstractions necessary to allow developers to reason correctly about the temporal behavior of application logic. It is not necessarily fast, small, or exclusively for industrial control; it is all about the predictability of the execution of application logic with respect to time.

This definition exactly summarizes our own understanding of real-time computing as promoted in this thesis. At its core, real-time computing is about *predictability*, the knowledge that the system will always perform deterministically within a required time frame. It's not about minimizing response times and latencies in any respect.

Clarification on how the two terms *hard real-time* and *soft real-time* are used: A hard real-time thread is a thread that must meet all deadlines without any exception while it does not necessarily have to minimize *latency*, the time between the occurrence of an event and its response. A *soft real-time thread* on the other hand is a thread that will still work correctly if it misses deadlines occasionally. Soft real-time systems normally specify the percentage of missed deadlines acceptable.

This chapter's focus is on coexistence of threads with different temporal constraints. A complex system runs threads with hard-, soft- and no deadlines and predictability comes from easily comprehensible coexistence. This will shift our attention to heap management and to (conditional) synchronization among differently constrained threads.

5.3.1 Sources of Temporal Unpredictability

There are a number of well-known sources introducing temporal uncertainties and thus causing threads to miss their expected deadlines. The most important sources are listed below together with a brief description of our approach.

- *Scheduling.* A schedule shall be predictable and correct. Realtime Oberon comes up with a provably correct priority inheritance implementation. Combined with the uniform, and therefore easy to anticipate, scheduling model (4), predictability is facilitated.
- *Module loading.* Because loading of code modules may require reading files from a secondary storage, invoking previously unused code might cause unexpected but potentially harmful delays. The overhead is at the expense of the thread that refers first to a not yet loaded module. The solution is to statically link all modules used by a dependable system to an image and to renounce dynamic linking and loading of code.
- *Garbage collection.* The primary source of unpredictability comes from garbage collection. Applications with hard response-time requirements cannot tolerate unpredictable pauses caused by heap management interferences. Realtime Oberon comes with an interruptible trace and sweep garbage collector, which allows realtime threads to manipulate heap objects during the collection phase. Real-time threads must only operate on preallocated heap blocks.
- *The application.* This is another major source of unpredictability. Most applications consist of multiple computational activities competing for computing resources. Our solution is to divide and conquer. Tasks are statically split into a concurrent graph of subtasks (2.1) in order to find a fine grained and feasible schedule.

5.3.2 Coexistence of Hard Real-Time with Non-Hard Real-Time Threads

Coexistence of hard real-time threads with ordinary threads requires isolation on different levels. The first level is the runtime priority. Hard real-time and all the other threads are divided into two disjoint sets priority wise. The topmost

priorities are exclusively assigned to hard real-time threads. Since our approach is to use a fixed priority based scheduler, all non-hard real-time threads are preempted whenever indicated. Note that for the sake of predictability, a hard real-time thread shall never propagate its priority to a non-hard real-time thread. A non-hard real-time thread would temporarily be promoted to a hard real-time thread. That would not sustain predictability.

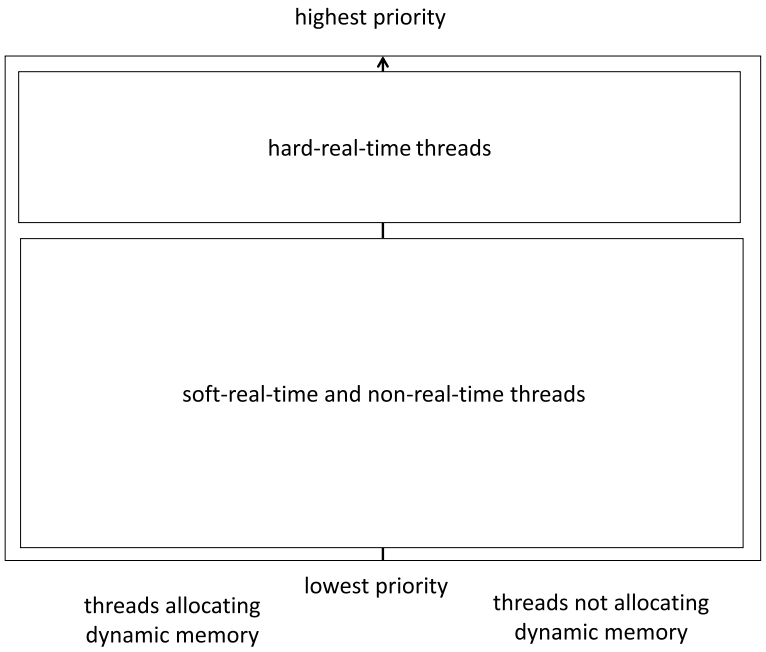

Figure 5.2: The set of threads

Unfortunately, hard real-time threads usually need to cooperate with non-hard real-time threads somehow. Let us study how hard real-time threads are connected with non-real-time threads via buffers at the familiar and generic example of the consumer and producer pattern. There are two set-ups to be considered.

hard real-time producer and non-hard real-time consumer

A common setup is a real-time thread putting data into the buffer and a non-real-time thread consuming from it. If the non-real-time thread does not get enough CPU time, the buffer runs over. There are three options to deal with such an overrun condition:

1. the buffer blocks the real-time producer,

2. the buffer is extended, or

3. the buffer discards data.

Since the hard real-time producer would lose its predictability being blocked by an unpredictable non-hard real-time consumer, the first option is unpractical. The second option is also unpractical. If the buffer extension is at the expense of the hard real-time producer, then it would interfere with the garbage collector. The next section will explain why this would seriously harm the hard real-time producer's predictability. If the buffer extension is at the expense of the non-hard real-time consumer, then the hard real-time producer would have to rely on an unpredictable thread. That is not acceptable. Hence the only appropriate strategy is to discard data.

The second issue, as previously mentioned, is priority inheritance. Locking and conditional synchronization obeys the priority inheritance protocol (4.2). Therefore the bounded buffer shown by listing 3.2 is not suited for data in exchange between a hard real-time producer and non-hard real-time consumer. An alternative is a lock-free buffer that would avoid the hard real-time thread to propagate its priority to a non-hard real-time thread such as the buffer inspired by [51] and presented in chapter 6.3.

hard real-time consumer and non-hard real-time producer

The second possible producer - consumer set up is the converse: a real-time consumer meets a non-real-time producer. If the non-real-time producer does not get enough CPU time, then the buffer runs under. A hard real-time consumer thread must have a strategy to cope with missing input. An extended implementation of the lock-free buffer shown in chapter 6.3 could limit the amount of time the hard real-time consumer waits for the signal in `PROCEDURE Remove`.

All remarks in the previous subsection about the priority inheritance issue also apply to the set-up discussed here.

5.3.3 Coexistence of Hard Real-Time Threads with Garbage Collection

Our garbage collector does not hide garbage collection overhead from hard real-time threads. Also, it has no “first-class citizen” status in the system, but obeys exactly the same rules as other threads, especially regarding the priority inheritance protocol. This has an impact on how the garbage collector and hard real-time threads coexist.

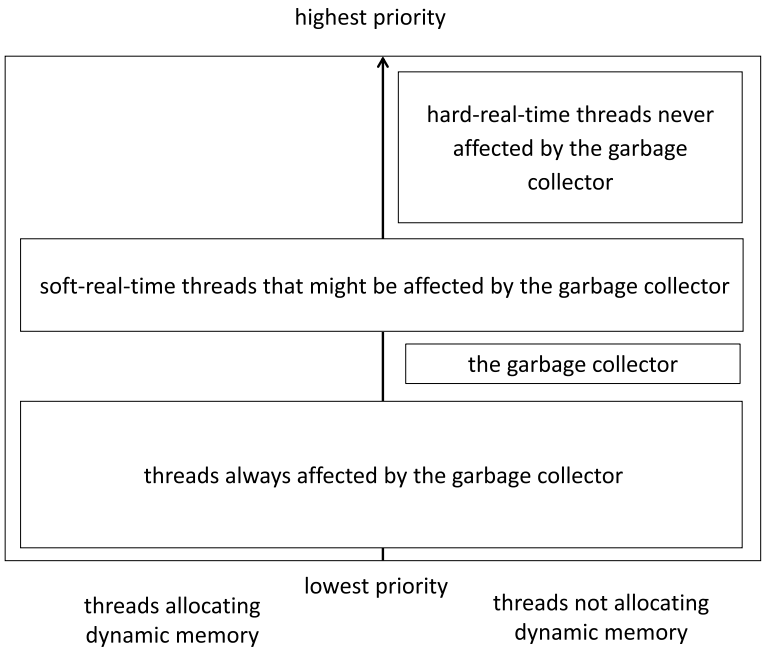

Figure 5.3: The set of threads

Since threads trying to allocate heap memory are synchronized via a monitor and a boolean condition with the garbage collector thread, the collector inherits the highest priority of all threads currently waiting for heap memory. We call this adaptive approach *Elastic Garbage Collection*, and it will be explained in chapter 6.4. As a side effect, a highly prioritized thread could harm another highly prioritized thread by pulling the garbage collector up to its own priority. This has to be taken into the account when implementing hard real-time threads.

The set of all threads can be partitioned into four disjoint subsets. Figure 5.3 shows how and refines figure 5.2. The first set contains only one element,

namely the garbage collector. The second set contains all threads that are permanently blocked by the garbage collector, the third contains threads that might be blocked and the fourth contains all threads that are never blocked. Conditions governing membership to the four disjoint subsets are:

- Threads running at lower priority than the garbage collector's static priority are preempted by the garbage collector at any point in time. These threads belong to the set of threads always blocked by the collector.
- Threads occasionally blocked by the garbage collector run at a static priority higher than the garbage collector. Each single thread of this set potentially raises the collector's priority to its own priority while it is waiting for memory due to the priority inheritance protocol. Thus the elements of this set interfere with each other via garbage collection.
- The set of threads expected to meet all deadlines without any exception must be isolated from the garbage collector because there are no useful prediction models available about the collector's runtime behavior. Threads never blocked by the garbage collector never request dynamic memory and run at a priority greater than the maximum priority of all threads that allocate dynamic memory. Hence, a member of this set will never wait for memory during an unpredictable amount of time and will never be affected by the garbage collector that is running at an equal or greater priority.

The simple (and rigid) constraint of isolating hard real-time threads from the garbage is to forbid dynamic heap memory allocation on the fly. There are three alternatives to dynamic memory:

- *Stack:* If the lifetime of a chunk of memory is well defined and local to a piece of programming code, it should be put on the stack. The stack is able to substitute the heap to a certain extent.
- *A*n Object pool is an important construction for reducing garbage collection workload. Well known examples are string pools used by compilers [44] or data buffers used for implementing a zero copy overhead network stack [27] for instance.
- *Static heap allocation:* The entire heap memory that a real-time thread is going to use during its lifetime is preallocated at the thread's creation

time, where the memory allocation code is located in the constructor's scope. The allocation overhead is then at the expense of the initializing thread rather than at the expense of the hard real-time thread itself.

5.4 Avoiding Deadlocks by Locking

There are different root causes for deadlocks. Deadlocks caused by locking and deadlocks caused by cyclic waiting. This section just focuses on deadlocks by locking, it is going to derive a set of implementation guidelines for *incrementally* building systems without deadlocks caused by locking.

The tactics for doing this takes advantage of modular system properties and applying design patterns based on the newly introduced `AWAITEVENT` statement.

In the next subsection we will demonstrate how to define a global locking order based on modular system properties and the subsequent subsection will show how the previously defined global locking order is implemented in practice.

5.4.1 Defining a global locking order

Our main goal is to impose a global locking order on all *shared* objects. A naïve approach could be to show that for every possible execution path and for every point in time all shared objects are locked according to a strict order. This might be a reasonable approach for systems with a statically fixed set of shared objects of a very limited cardinality.

A better approach is to take advantage of how modules of the system are arranged. Instead of immediately ordering each single object instance on a global scale, we use a "divides and rules" strategy: Reasoning on the global module topology and subsequently reasoning on each module's internals.

The flow of logic is as follows:

1. System modules are arranged according to a Directed Acyclic Graph. Since every Directed Acyclic Graph (DAG) is serializable, a total order on all modules exists and is easily computable.

2. There is a 1 : n relation between modules and types. No type is declared in more than one module. If types are totally ordered within each module and modules are totally ordered based on a DAG serialization, then all existing types are totally ordered on a global scale.
   Remember that each module itself represents an implicit type, which must also be considered.

3. If there is a 1 : n relation between types and object instances, then all objects are totally ordered since there is, maximum, one instance of each type involved in the locking chains.

Basically we have arrived at the desired global total ordering of all shared objects involved in our computation. In order to implement the presented type ordering approach, the implementation must address some issues:

- Whenever an additional module is implemented, an *intra module locking order on all shared object types* has to be defined and respected according to point 2 in the previous subsection. Due to the modularity of that approach, incremental systems development is not at all compromised.

- All *execution paths must strictly follow the directed acyclic graph topology* of the module tree structure. If this does not occur, then total module ordering will be compromised.

- *No locking and unlocking hysteresis* may occur along any execution path. More concretely: all locks that have been taken must be released again before the execution path leads back to the caller's scope. Locks on object instances are released implicitly when the execution path leaves an `{EXCLUSIVE}` section.
  A special remark on low level modules: Due to system bootstrapping issues, the `{EXCLUSIVE}` primitive is not available in low level modules. In this case, mutual exclusion is implemented using low level (spin) locks. To avoid a locking and unlocking hysteresis, each low level module uses exactly one low level lock instance in order to emulate the missing module monitor lock. Showing that low level locks are used properly is straight forward, as long as they are only used to emulate a module monitor. If a low level lock is always released in the same procedure that it was acquired, then its proper use can easily be verified.

- Some special care has to be exercised regarding *nested locking*. According to point 3 in the previous list, nested locking of objects of the same type must be avoided.

5.4.2 Implementing a global locking order

This subsection will show how the previously defined global locking order is implemented in practice. Since all execution paths must strictly follow the directed acyclic module graph topology, we will introduce three design patterns that help the programmer to achieve this.

As virtual calls and the like are questionable because the execution paths must strictly follow the directed acyclic graph topology of the module tree structure, alternatives are provided to either avoid or control especially virtual up-calls in the module tree.

Avoiding Inversion of Control

Traditional inversion of control based frameworks lack strong static guaranties regarding the execution path. Extensive use of dynamically configurable delegates or function pointers weakens the deterministic execution of the system. Realtime Oberon comes with a built in `AWAITEVENT` statement that was introduced to eliminate the root cause for inversion of control: up-calls like interrupt handler (listing 3.4), timer (listing 3.5) and finalizers (listing 3.7). However, the built in `AWAITEVENT` statement is necessary but not sufficient to build up-call free systems, since Realtime Oberon also enables up-calls due to polymorphism. The following paragraphs show how to employ polymorphism without introducing up-calls in the module tree hierarchy.

Up-Call Free Programming

Polymorphism is a powerful abstraction technique with a major drawback regarding our context: virtual calls. Virtual calls introduce unpredictability in the execution path, which contradicts our deadlock avoidance strategy. Let us use the “Template Method” pattern [11] to illustrate the principle of mapping virtual calls to predictable execution path codes. Figure 5.4 depicts the “Template

Method" pattern.

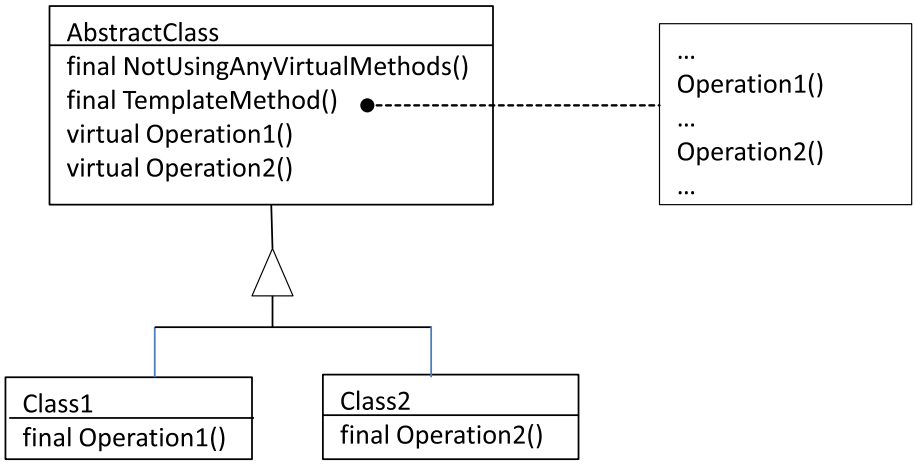

Figure 5.4: Template Pattern

The AbstractClass contains methods that *do* and *do not* invoke virtual methods. The first method does not invoke virtual methods as its identifier suggests, whereas the second method, called "TemplateMethod", invokes the virtual methods "Operation1" and "Operation2". "AbstractClass" provides a default implementation for both operations. "Operation1" and "Operation2" are overridden in Class1 and Class2. Class1 and Class2 will have to be implemented in two additional modules. The solution to avoiding up-calls in the execution path then looks like this:

- `MODULE Basics` declares `OBJECT AbstractClass` that implements `PROCEDURE NotUsingVirtualMethods` and provides a default implementation for `PROCEDURE Operation1` and `PROCEDURE Operation2`. Note that there is no `PROCEDURE TemplateMethod` declared.
- `MODULE Refine1` imports `MODULE Basics` and declares `OBJECT Class1(Basics .AbstractClass)` which overrides `PROCEDURE Operation1`.
- `MODULE Refine2` (not shown in figure 5.5) imports `Module Basics` too and declares `OBJECT Class2(Basics.AbstractClass)`, which overrides `PROCEDURE Operation2`.
- `MODULE Top` imports the three modules and implements the missing "TemplateMethod" of AbstractClass as a module procedure and not as an object method. `PROCEDURE TemplateMethod` takes an object instance as a parameter.

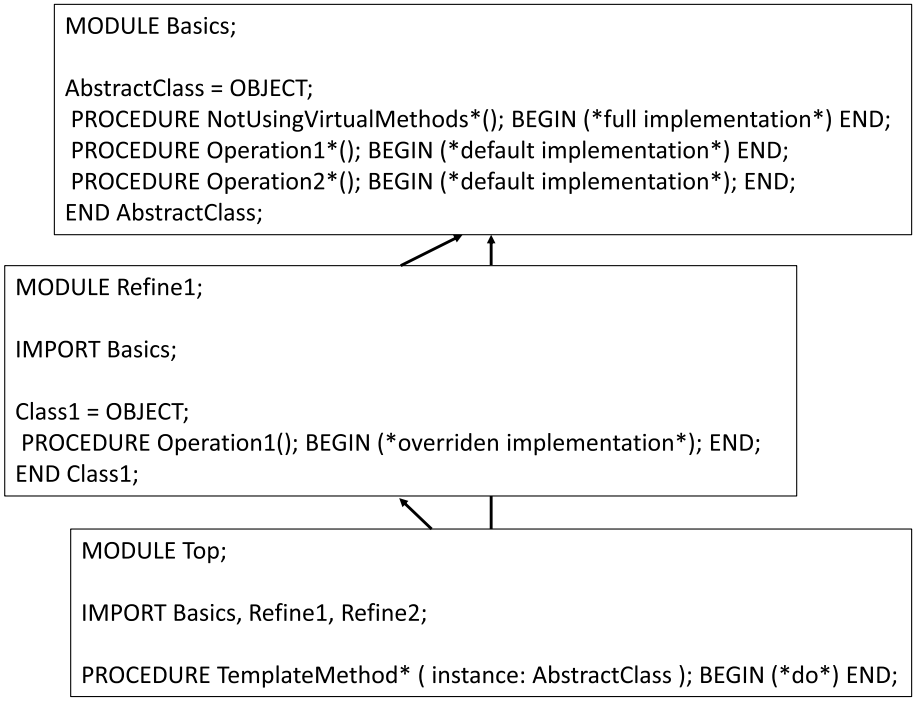

Figure 5.5: Template Pattern transposed to Oberon Modules

The method in summary: Extracting the functionality that relies on virtual operations and implementing it, not as an object method but rather as a `MODULE PROCEDURE` in a top level module. If the top level module is aware of all existing implementations of some base type that it operates on, all virtual calls made on this base type sift down along the module tree hierarchy.

Nested Locking

As mentioned in the previous subsection, shared objects of the same type are not supposed to be locked in a nested way. This means in particular that tree-like structures of the same type cannot be locked node by node while traversing. An `EXCLUSIVE` inorder traversal as indicated in 5.3 would violate the rule. It would lock `node.left` of type `Node` within an `EXCLUSIVE` section of type `Node`.

Listing 5.3: Inorder Traversal

```
Node = OBJECT
    VAR left, right : Node
```

```
    PROCEDURE inorder(node : Node)
    BEGIN (*{EXCLUSIVE}*)
        IF (node # NIL) THEN
         inorder(node.left);
         visit(node);
         inorder(node.right);
        END
    END Inorder;

END Node;
```

The principle to get around nested locking is to rethink the locking granularity. Instead of recursively locking the tree nodes, the tree as a whole is locked. The lock on a tree is implemented by conditional waiting (listing 5.4). Hence one could say, the problem of nested locking of objects of the same type is transformed into the problem of avoiding deadlocks by conditional waiting. The price for this principle is a loss of parallelism. By implementing coarser locking entities, the potential for parallel execution is decreased.

Listing 5.4: Root Lock

```
Tree = OBJECT
    VAR root : Node;
        locked : BOOLEAN;

    PROCEDURE & Init()
    BEGIN
        locked := TRUE;
    END Init;

    PROCEDURE Acquire()
    BEGIN{EXCLUSIVE}
        AWAITCONDITION ~locked;
        locked := TRUE;
    END Acquire;

    PROCEDURE Release()
    BEGIN{EXCLUSIVE}
        locked := FALSE;
    END Release;

END Tree;
```

5.4.3 Avoiding Deadlock Interference with Conditional Waiting

Enforcing a global locking order is insufficient to avoid deadlock interferences with conditional waiting. An example showing such a deadlock is outlined in listing 5.5. A `TreeAdapter` object encapsulates a `Tree` instance shown in the previous listing 5.4. If a thread invokes `Acquire()`, it will be passivated on `tree.Acquire()`. A second thread supposed to invoke `Release()` will be blocked, because the former thread has not released the monitor of the `TreeAdapter` instance since it was passivated on `tree.Acquire()`. Thus there is a deadlock.

Listing 5.5: Interference with Conditional Waiting

```
TreeAdapter = OBJECT
    VAR tree : Tree;

    PROCEDURE & Init()
    BEGIN
        NEW(tree);
    END Init;

    PROCEDURE Acquire()
    BEGIN{EXCLUSIVE}
        tree.Acquire();
    END Acquire;

    PROCEDURE Release()
    BEGIN{EXCLUSIVE}
        tree.Release();
    END Release;

END TreeAdapter;
```

To avoid deadlocks caused by interference with conditional waiting, one more restriction is required:

- Within a monitor, calls are prohibited to other monitors including an `AWAITCONDITION` statement.

5.4.4 Related Work

There are two basic strategies to cope with deadlocks: *deadlock immunity* and *deadlock avoidance*. The purpose of *Deadlock immunity* is not primarily to eliminate deadlocks but to cure their symptoms. One approach currently promoted by Candea [23] can be sketched as follows: When a deadlock has been detected (manually or by some dedicated watchdog process) the execution state of all processes involved are fingerprinted. The scheduler will subsequently prevent the system from getting into the same deadlock again by serializing the processes involved. The most important drawback of this method is that a deadlock has to be triggered once before a workaround can take effect. Another approach for deadlock immunity was presented by Vitek [33]. A side effect of substituting classical monitors [16] with preemptable atomic regions voids deadlocks caused by locking. Preemptable atomic regions are code blocks in which threads are rolled back whenever a higher prioritized thread is trying to enter. Unfortunately, preemptable atomic regions do not prevent higher order deadlocks caused by conditional synchronization.

Deadlock avoidance sounds more promising but is harder to achieve. Theorem proving and constraint solving are well established techniques to prove the absence of deadlocks. They are well suited to verify small subsystems like device drivers [35]. Proving larger systems would require considerably more computing power than is available today. Another approach is simply testing. Systematic testing is still an indicator of a system's reliability [41]. The limitation with testing is achieving acceptable coverage of all relevant cases and combinations thereof. Language based techniques rely on type systems and require either annotations [20] or impose the way programs have to be written.

5.5 Avoiding Deadlocks by Cyclic Waiting

As mentioned in the introduction of the previous chapter, deadlocks can also be caused by cyclic waiting.

Avoiding deadlocks caused by cyclic waiting is more subtle than avoiding deadlocks due to mutual exclusion. Since an `AWAITCONDITION` statement is required to be embedded in an `{EXCLUSIVE}` section, one could argue that the

posed problem is equivalent to avoiding deadlocks by mutual exclusion. Unfortunately, imposing a total locking order on shared objects, as proposed in the previous chapter, is insufficient. The reason for this is that context switches from a thread waiting for a certain condition to threads establishing the condition and back, which calls for additional precautions.

Our approach to avoid deadlocks by cyclic waiting relies on the active objects computing model presented in 2.3. Ordering all threads and shared objects in between according to a global directed acyclic graph is a sufficient (but unnecessary, as shown later) condition for avoiding deadlocks by cyclic waiting. Figure 2.5 shows an example of such a graph. The subsequent section shows how to implement the approach in practice.

5.5.1 Implementation guidelines

Iterating on the following two steps allows building systems incrementally without creating cyclic waiting conditions. Each time a newly created software module is added to an existing set of modules, one has to reiterate the two steps below:

1. First, all new object instances, which may potentially contribute to any circular dependencies, have to be identified.
   (a) Only shared (in contrast to shareable) objects have to be considered. Shared objects that never block any thread on an `AWAITCONDITION` statement do not need attention.
   (b) Object instances that block on a time-out basis (`AWAITCONDITION, condition, maxTimespan`) also need no attention. An example is an instance of the object type “Timer” shown in listing 3.3.
   (c) A blocking object instance for which a plausible explanation exists as to why it will not contribute to a deadlock need not be considered.
   (d) All other instances belong to the set of persistently blocking objects with deadlock potential and must be considered.
2. All persistently blocking object instances with deadlock potential that have been identified in step (1d) plus all threads operating on these blocking instances must be aligned according to a global directed acyclic dependency graph.

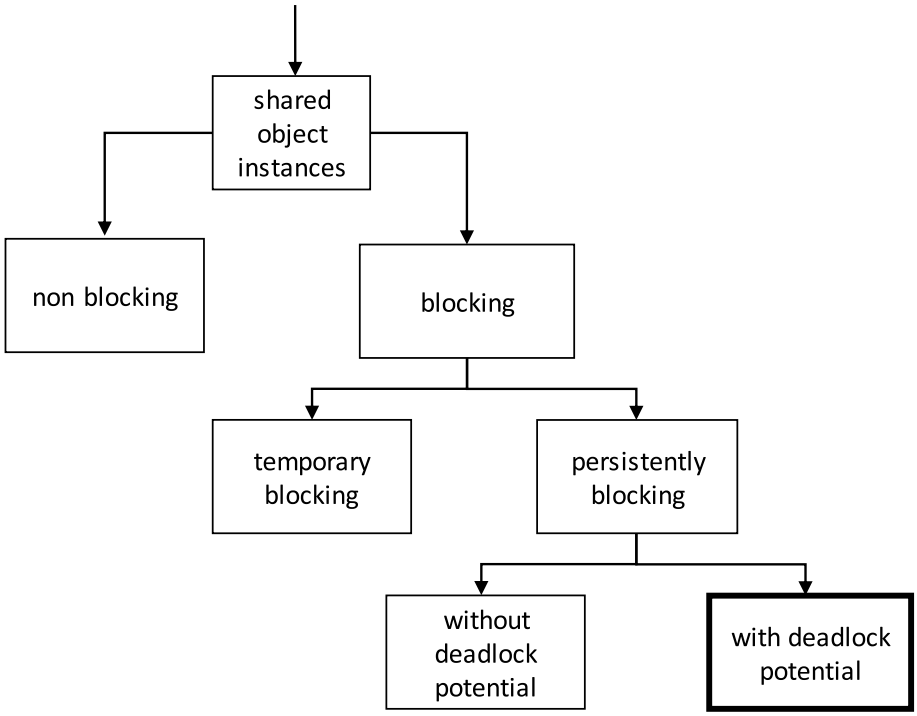

Figure 5.6: Classification with respect to deadlock potential

Building such a directed acyclic dependency graph on a global scale is obviously a reasonable approach to prove freedom of deadlock due to cyclic waiting. The active objects computing model, as proposed in chapter 2.3, fosters DAG like thread / shared object graphs like the example in figure 2.5. Arrows show how the “information” flows in the DAG. An arrow towards a thread indicates that it is waiting for a condition to become true, a pointer away from a thread indicates that it establishes some condition.

5.5.2 Short Cuts

Not all blocking objects smoothly fit into a static acyclic thread / shared object graph. A typical pattern of how threads operate blocking resources might be of a non-static nature: blocking serial connections can be sequentially used by many different threads, block devices are randomly accessed by different threads in an unordered way. In these cases, it would be very restricting from an architectural point of view to enforce a particular global order on *all* blocking resources. But the good news is: Usually there is a plausible explanation why such resources do not have a deadlock potential at all.

There are two strategies to cope with blocking objects that do not smoothly fit into a static acyclic thread / shared object graph:

- Either by identifying a thread that always breaks the deadlock chain. If such a thread exists, the blocking object instance belongs to the set of persistently blocking instances without deadlock potential according to figure 6.10. An example is the blocking “Heap” object. The garbage collector thread always unblocks memory requesting threads if there is memory available. Another example is the blocking “Serial Port” object. The interrupt handling thread always unblocks threads reading from and writing to the serial port object. Hence neither the heap nor a serial port object is going to be the root cause for deadlocks and thus are not required to be considered further. Most low-level producer / consumer compositions do not have deadlock potential because a thread always breaks the chain.

- Or by substituting a blocking instance with an unblocking instance. The blocking disk cache in the secondary storage subsystem, for example, has been replaced by an unblocking version. It was blocking in the case of an internal buffer underrun, the unblocking version refuses to cache elements when a buffer underrun occurs. The lesson learned is to use blocking objects with care.

These two strategies to cope with blocking objects render the introduced implementation guidelines in 5.5.1 sufficient but unnecessary.

Chapter 6

Selected Implementation Issues

Realtime Oberon is based on an ARM implementation of the Active Objects System [38] originally ported by Bernhard Egger [10]. Later, Egger adapted the System to Intel's XScale platform, which is currently maintained and marketed by Marvell Semiconductor, Inc.

6.1 Priority based Interrupt Handling

Interrupt handling threads, like the one sketched in listing 3.4, are from a conceptual point of view ordinary threads in our unified system and are therefore supposed to be fully compliant with the scheduling rules stated in chapter 4. The following two aspects are of special interest from an implementation point of view:

- First level interrupt handling.
- Adaption of the priority inheritance protocol.

6.1.1 First Level Interrupts

One of the very basic principles derived from the two scheduling rules in chapter 4.1 is: A lower prioritized thread never preempts a higher prioritized thread. An implementation that meets this requirement is not straight forward. Behind the scenes, there is a *first level interrupt handler* driving thread state transitions from "Awaiting Event" to "Ready" in figure 4.1. When an interrupt has

been raised, the first level interrupt handler picks the corresponding (second level) interrupt handler thread and puts it into the ready queue. The question to be answered here is: under what conditions is the first level interrupt handler allowed to preempt the currently running thread. A common approach is to let the first level interrupt handler preempt the running thread at any time. However, there are good reasons to minimize unnecessary preemption, such as keeping runtime predictability high and minimizing cache misses. Therefore, our implementation strives to minimize the number of preemptions by first-level interrupt handlers and to avoid unnecessary context switches completely.

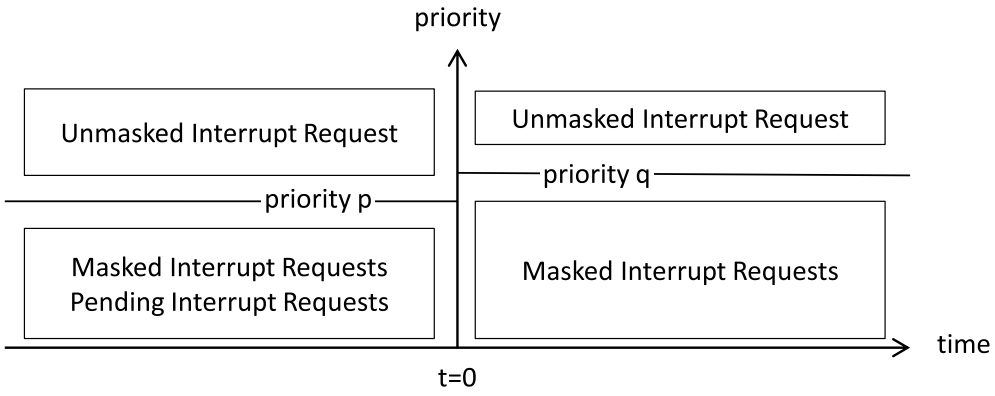

Figure 6.1: Interrupt Masking

This is achieved by making extensive use of priorities for fine granular masking and unmasking of interrupts. The first level interrupt handler preempts the running thread only if the thread in charge of handling the second level interrupt has a higher priority than the running thread.

For example, Figure 6.1 shows how masking and unmasking is done in practice. Before t = 0, a thread with priority p is running. Hence all interrupts handled by a second level interrupt handler thread with priority equal to or less than p are masked, all others are unmasked. There might also be some interrupts pending. Suddenly, an unmasked interrupt is raised at t = 0. The running thread with priority p is preempted and the handler thread with priority q is scheduled. Also at t = 0, all pending interrupt handler threads with priorities less than or equal to p are moved to the ready queue.

6.2 Provably Correct Priority Inheritance Protocol Implementation

This section will explain how the priority inheritance protocol (PIP) introduced in section 4.2 has been implemented. It will give informal proof showing how the presented implementation is correct.

The core of this proof is the class `PIPNode` shown in listing 6.4. It has been proven by Rudich et al. [45] that priority inversion is never going to occur in graphs of `PIPNode` instances. Two subclasses named `PIPThread` (6.5) and `PIPMonitor` (6.6) have been derived from `PIPNode`. Since the two derived classes do not override any code, but only add a property and a few members, priority inversion is neither going to occur in graphs of `PIPMonitor` and `PIPThread` instances.

Section 6.2.2 specifies the implemented PIP and derives the required operations on the verified `PIPMonitor` and `PIPThread` graph. Section 6.2.3 shows how the required operations have been implemented and why these are correct with respect to the PIP specification in 6.2.2 . And finally, section 6.2.4 presents the verified class `PIPNode`.

The next section introduces the necessary basic knowledge to understand the implementation of the verified class `PIPNode`.

6.2.1 Tracking Monitor and Thread Dependencies

The PIP enters the stage when threads are synchronized with resources based on monitor technology. A thread is *blocked by* a monitor if the thread does not have access to it. We say that a monitor is *owned by* a thread if the thread is currently operating within the monitor. Figure 6.2 shows the most general case: Solid arrows denote *"blocked_by"* relationships. Threads 1 to n are blocked by monitor 1, threads n+1 to m by monitor 2 and finally thread k is blocked by monitor n. Dashed pointers denote *"owned_by"* relationships. Monitor 1 and 2 are owned by thread k while Monitor n is owned by an unspecified thread. Four invariants hold on to the monitor / thread dependency graph in figure 6.2:

- Invariant 1: A thread is blocked by zero or one monitor

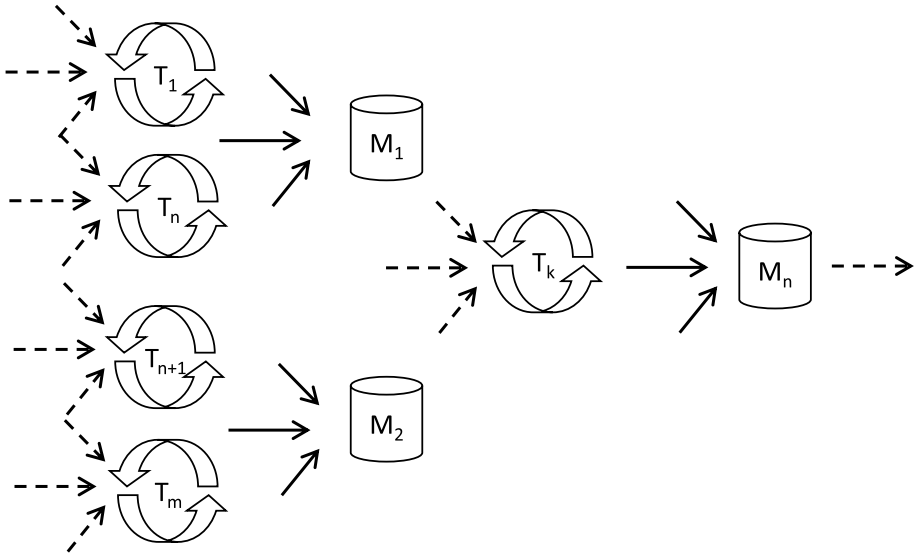

Figure 6.2: General thread / monitor dependency chain

- Invariant 2: A monitor may block an unbounded number of threads.
- Invariant 3: A monitor is owned by zero or one thread.
- Invariant 4: A thread may own an unbounded number of monitors.

Obviously, a thread is blocked by a monitor only if the monitor is owned by another thread.
Each thread has a property called *"static priority"* assigned with it. It gets its static priority at creation time and keeps it during its entire lifetime.
Another property a thread has is called *"current dynamic priority"* or short *"current priority"*. This priority is equal to the maximum of its static priority and the current priorities of all threads that are directly or indirectly blocked by the thread itself. There are two events potentially invalidating the current priority of a thread: Either a "blocked_by" or "owned_by" relation is added or removed. Note that a thread does not have to be involved directly in such a state transition. The next three subsections explain how the "current priority" property is updated whenever relations are created or removed. Understanding this will help to understand the core of the provably correct PIPNode implementation in listing 6.4.

If a runtime system is requested to determine the accurate runtime priority a thread is supposed to run at any point in time, then it must keep track of the "blocked_by" and "owned_by" relationships introduced in the previous section.

Formally, the goal is to keep the property “current priority” assigned to each single thread up to date, as a prerequisite to reason about priority inversion scenarios.
The illustrating examples that will be used in the coming paragraphs are simplified without a loss of generality. Especially the set of available thread priorities is reduced to High, Medium and Low, for the sake of simplicity.

Tracking “blocked_by” Relations

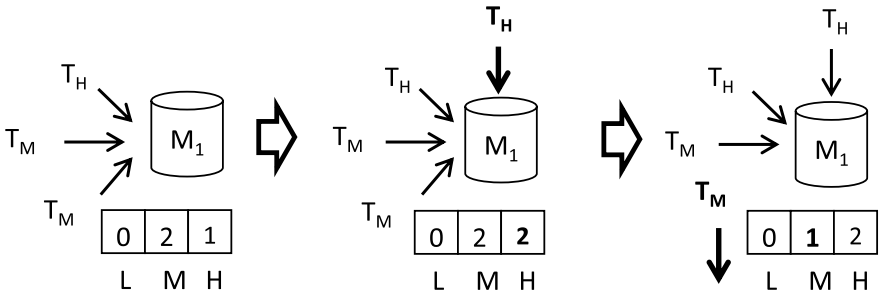

Figure 6.3: Tracking “blocked_by” relations

Each pending locking request must be tracked. The multiset depicted below the resource symbol keeps track of the number of high, medium and low prioritized threads that are currently blocked by the monitor. Initially, the monitor shown leftmost in figure 6.3 blocks two threads of priority “Medium” and one thread of priority “High”. When supposedly a fourth thread of priority “High” also tries to acquire the monitor, the counter “H” is simply incremented by one. Thus we have:

- Invariant 5: The multiset’s cross sum is equal to the number of threads blocked by the monitor.

When a blocked thread has succeeded to enter the monitor or failed due to a timeout, the respective counter is decremented (figure 6.3 rightmost). Thus Invariant 5 is restored. If all “blocked_by” relations are tracked carefully, corollary 5 follows immediately:

- Corollary 5: The total number of threads blocked by a particular monitor is known at any point in time.

An additional property has been assigned to each monitor instance: The *"current priority" of a monitor* is defined as the maximum current priority of all threads currently trying to enter into the monitor. Monitor 1 in figure 6.3 always has a "current priority" value "High".

Tracking "owned_by" Relations

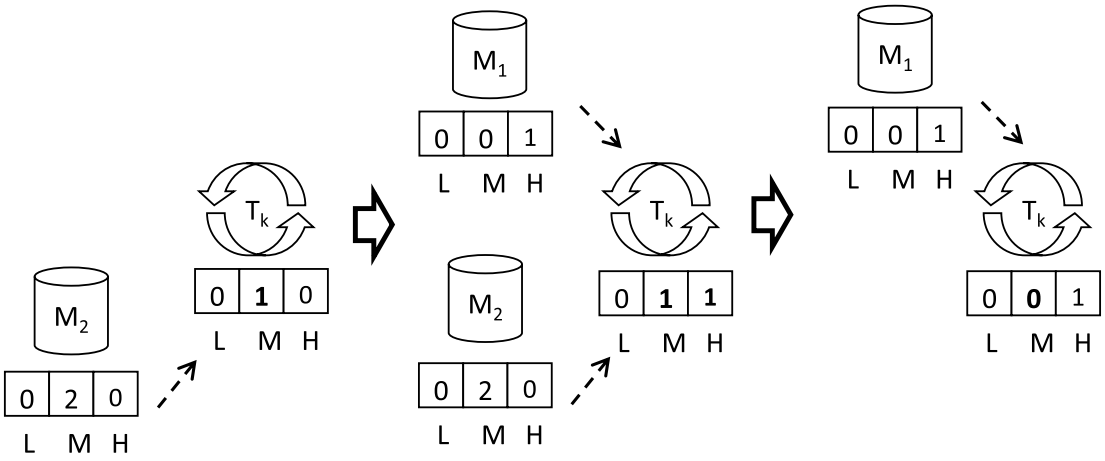

Figure 6.4: Tracking "owned_by" relations

Monitors acquired by unblocked threads also have to be tracked. For instance, monitor 2, left in figure 6.4, locked by thread k and monitor 1 in the middle, is also locked by thread k. The multiset depicted below the thread symbol keeps track of how many high, medium and low prioritized monitors are currently owned by the thread itself. On the left, the counter "M" is incremented by one when the thread acquires a monitor of current priority "Medium". Then the counter "H" is incremented by one when the task acquires a monitor of current priority "High". And finally the counter "M" is decremented by one when the task exits from the monitor of current priority "Medium" again.
The "current priority" of a thread could also be defined as the maximum of the threads's static priority and the current priorities of all monitors owned by the thread itself.

If all "owned_by" relations are tracked carefully, invariant 6 and corollary 6 follow:

- Invariant 6: the multiset's cross sum is equal to the number of monitors currently owned by the thread.

- Corollary 6: All monitors owned by any particular thread are known at any point in time.

Corollary 6 implies a key conclusion: the “current priority” of the thread is determinable at every point in time. This is a precondition for implementing the PIP.

Transitive Priority Propagation

Since priorities are propagated transitively via monitor and thread chains, each time an “owned_by” or “blocked_by” relation is established or eliminated, all affected threads have to be identified.

Figure 6.5 shows such a chain of three threads and two monitors. Thread n+1 runs on priority “High”, thread k and thread q on priority “Medium”.
The only way a priority inversion anomaly may occur, is by establishing a new “blocked_by” relationship as indicated by the fat solid arrow from thread n+1 to monitor 2 in figure 6.5. If thread n+1 runs at priority “High” and is blocked by monitor 2, which in turn is owned by thread k running at priority “Medium”, the priority inversion issue between thread n+1 and thread k must be resolved by increasing thread k’s priority to “High”. Unfortunately, these inversion anomalies happen recursively: By increasing thread k’s priority to “High”, a new priority inversion issue has been created between thread k and thread q. Hence in a second step, the priority of thread q must be incremented to “High” also. Obviously priority inheritance is a transitive phenomenon.

The propagation operation along a directed graph, like the one shown in figure 6.2, is supported by the auxiliary book keeping multisets introduced previously. When the current priority of an inner node is altered, the out-dated current priority is removed and the new priority is added to the multiset of the targeted node element. This operation is iterated transitively on the graph until a fixed-point is reached.

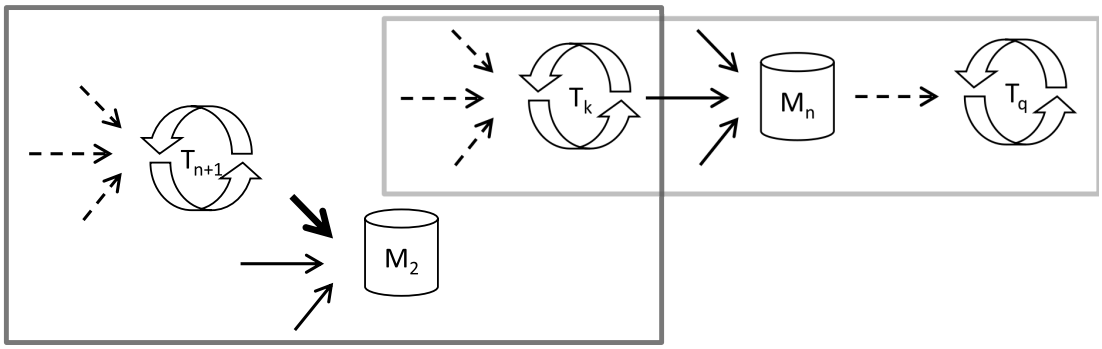

Figure 6.5: Priority Inversion Cascade

Initialization

Adding sentinels simplifies the implementation in 6.4 significantly. Sentinels make sure that the multisets used for bookkeeping are always in a defined state. Hence two virtual relations are introduced: The static priority each

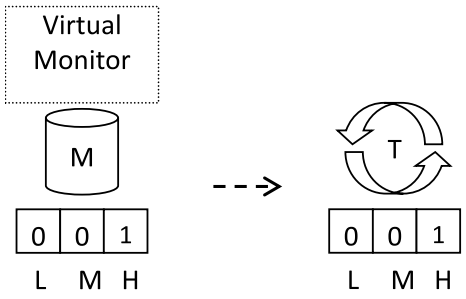

Figure 6.6: Process Initialization

thread instance gets at creation time is in fact a lower bound for the thread's current priority. To get around an explicit "dynamic_priority" field, a virtual "owned_ by" relation is created at a thread's creation time. Thus the static priority is inherited from a virtual monitor as if the process would operate in such a monitor during its whole lifetime. This virtual "owned_by" relation is never removed. Thus the thread's current dynamic priority will never go below the static one.

A monitor is initialized as if the idle thread, running at the lowest priority possible, would try to acquire it. This virtual "blocked_by" relation also persists during the monitors's whole lifetime.

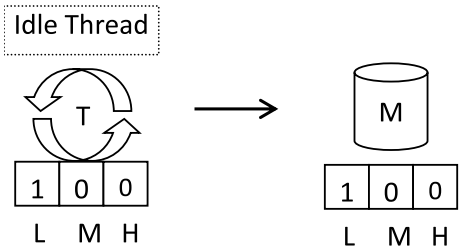

Figure 6.7: Resource Initialization

6.2.2 The Priority Inheritance Protocol Specification

As mentioned in 4.2.2, each thread strictly obeys the three step protocol "blocked" - "entered" - "left". While a thread is blocked (indicated by a solid arrow in figure 6.8), it propagates its priority consecutively to threads operating within the monitor. Once the blocked thread has entered the monitor (indicated by a dashed arrow), it inherits the priorities from all the blocked threads and when the thread leaves the monitor (indicated by a dotted arrow), the thread renounces the inherited priorities. These specifications are in accordance with the general understanding of priority inheritance.

There are two invariants for monitors:

$$I1 : \{monitor\_owned\_by\_one\_thread\} \vee \{monitor\_not\_owned\_by\_a\_thread\} \tag{6.1}$$

$$I2 : \{monitor\_owned\_by\_one\_thread\} \vee \neg \{threads\_blocked\_by\_monitor\} \tag{6.2}$$

Invariant I1 tells us that a monitor cannot be owned by two threads or more. Invariant I2 states that threads may not be blocked while no other thread owns the monitor.

These invariants are in accordance with the general understanding of monitors. Hence the informal specification of the PIP and the notion of monitors we use may be considered as validated.

There are two events when the state of the monitor has to be reevaluated, either when a thread acquires the monitor or when a thread leaves the monitor. The former event is triggered by transition (1), the latter by transitions (5) or (6).

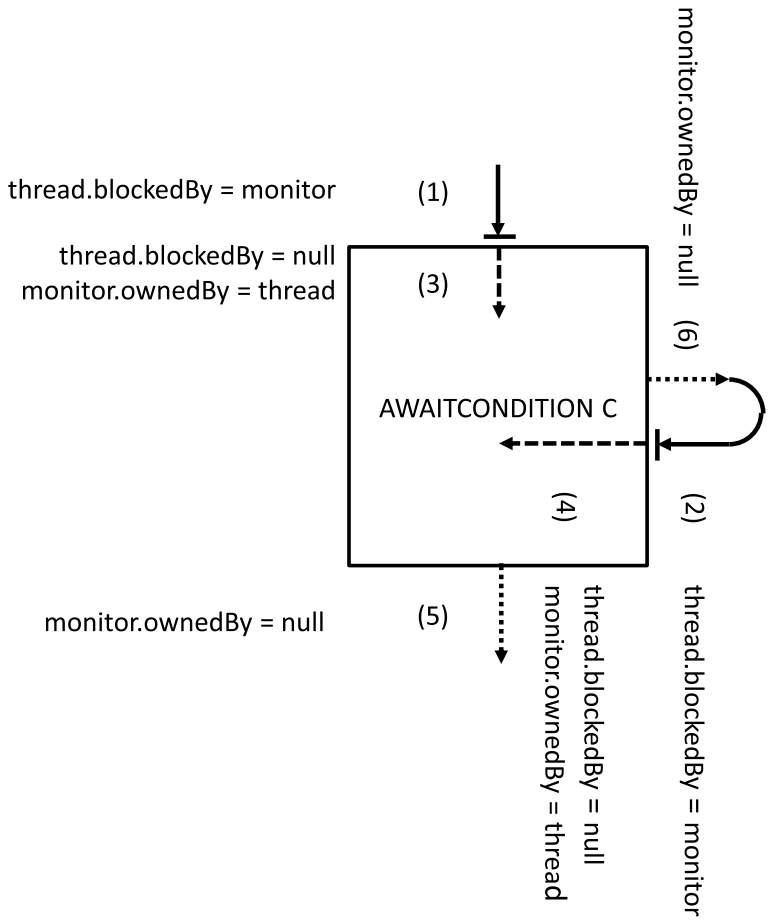

Figure 6.8: Priority Inheritance Protocol

Hence an implementation of the proposed PIP must provide three handlers. These handlers must re-establish the invariants I1 and I2.

Let us translate the protocol transitions in figure 6.8 into relations between threads and monitors. This is useful for implementing the required PIP handler. By assumption, a thread object possesses a “blocked_by” member that references the monitor it is blocked by and a monitor object possesses an “owned_by” member that references the thread it is owned by. When a thread is blocked by a monitor (cases (1) or (2)), a “blocked_by” relationship between thread and monitor is established. If the thread gains access to the monitor (cases (3) or (4)), the previously established “blocked_by” relationship is canceled and an “owned_by” relationship between monitor and thread is established. Finally, when the thread is about to leave the monitor (cases (5) or (6)), the previously established “owned_by” relationship is canceled.

These four operations on relationships are invariant for all possible paths through the monitor. No matter whether the thread takes (1) ⇒ (3) ⇒ (5), (1) ⇒ (3) ⇒ (6) or (2) ⇒ (4) ⇒ (5). It is evident that the four operations on relationships are also invariant if more than one `AWAITCONDITION` statement is located within the monitor.

The next challenge is to show that the concrete implementation always respects the protocol and always re-establishes the two invariants stated above.

6.2.3 Affected System Calls

As mentioned in the previous subsection, there are three system calls required to handle the PIP, namely;

- "Lock", which is invoked by a thread requesting the monitor,
- "Unlock", which is invoked by a thread about to leave the monitor, and
- "AwaitCondition", which is invoked by a thread blocked on an unestablished condition.

The next three subsections explain the implementation of these three system calls in detail. The code samples are simplified and just focus on PIP related issues. The reason why the system calls are treated in detail here is to show how the adaption of the priority inheritance concept sketched in figure 6.8 is mapped to OS primitives. It will be shown precisely where the code is located that implements the six transitions in 6.8. It will also be explained how the invariants are re-established. All code samples are in C# notation.

A monitor object possesses an "owned_by" member that references the thread it is owned by. Since one member variable can only refer to one object or to null, Invariant 1 is true per default. Hence only invariant 2 has to be tracked from now on.

Lock

Listing 6.1: Lock

```
public static void Lock(PIPMonitor monitor)
```

```
{
    if (monitor.owned_by == null)
    {

        // not yet locked
        /*1   runningThread.blocked_by = monitor;*/
        /*2   runningThread.blocked_by = null; */
        /*3*/ monitor.owned_by = runningThread;
    }
    else
    {
        // already locked by another thread
        /*4*/ runningThread.blocked_by = monitor;

        runningThread.state = AwaitingLock;
        monitor.awaitingLock.put(runningThread);
    }
}
```

The procedure `Lock` consists of exactly two execution paths.

The path with the statements `/*1*/`, `/*2*/` and `/*3*/` is executed if the monitor is not yet owned by a thread. This path correctly implements transitions (1) and (3) in figure 6.8. Note that the statements `/*1*/` and `/*2*/` are uncommented because the latter is the inverse of the former. Invariant 2 remains established because the monitor is owned by the running thread.

The path with statement `/*4*/` is executed if the monitor is already owned by a thread. Thus the running thread is blocked by the monitor. This execution path correctly implements transition (1) in figure 6.8. Invariant 2 remains established because the monitor continues to be owned by an unknown thread.

Unlock

Listing 6.2: Unlock

```
public static void Unlock(PIPMonitor monitor)
{
    /*1*/ monitor.owned_by = null;

    PIPThread c = FindTrueCondition(monitor.awaitingCond);
    if (c == null)
    {
```

```
            PIPThread t = monitor.awaitingLock.
                getElementWithHighestCurrentPriority();
            if (t != null)
            {
                //lock transfer to Thread t
                /*2*/ monitor.owned_by = t;
                /*3*/ t.blocked_by = null;

                t.state = Ready;
                EnterInReadyQueue(t);
            }
        }
        else
        {
            //lock transfer to Thread c
            /*4*/ monitor.owned_by = c;
            /*5*/ c.blocked_by = null;

            c.state = Ready;
            EnterInReadyQueue(c);
        }
    }
```

The procedure `Unlock` consists of exactly three execution paths. All paths execute statement `/*1*/`. Statement `/*1*/` correctly implements transition (5) in figure 6.8, the running thread leaves the monitor.

If neither a thread `c` stuck on an established condition nor a thread `t` trying to acquire the monitor is found, then no additional statement is executed. In this case, Invariant 2 remains established because no threads are blocked by the monitor.

The path with the additional statements `/*4*/` and `/*5*/` is executed if a thread `c` stuck on an established condition is found. The monitor is transferred to this thread `c`. This path correctly implements transitions (5) and (4) in figure 6.8. Invariant 2 remains established because the monitor is owned by the thread `c`.

The path with the additional statements `/*2*/` and `/*3*/` is executed if no thread `c` is stuck on an established condition but a thread `t` trying to acquire the monitor is found. The monitor is transferred to this thread `t`. This path correctly implements transitions (5) and (3) in figure 6.8. Invariant 2 remains established because the monitor is owned by the thread `t`.

AwaitCondition

Listing 6.3: AwaitCondition

```
public static void AwaitCondition(PIPMonitor monitor)
{
    assert(monitor.owned_by == runningThread);

    //check condition, if true then return

    /*1*/ monitor.owned_by = null;

    PIPThread c = FindTrueCondition(monitor.awaitingCond);
    if (c == null)
    {
        PIPThread t = monitor.awaitingLock.
            getElementWithHighestCurrentPriority();
        if (t != null)
        {
            //lock transfer to Thread t
            /*2*/ monitor.owned_by = t;
            /*3*/ t.blocked_by = null;

            t.state = Ready;
            EnterInReadyQueue(t);
        }
    }
    else
    {
        //lock transfer to Thread c
        /*4*/ monitor.owned_by = c;
        /*5*/ c.blocked_by = null;

        c.state = Ready;
        EnterInReadyQueue(c);
    }

    /*6*/ runningThread.blocked_by = monitor;

    runningThread.state = AwaitingCond;
    monitor.awaitingCond.put(runningThread);
}
```

The procedure `AwaitCondition` consists of exactly three execution paths. All paths execute the statements `/*1*/` and `/*6*/`. Statement `/*1*/` correctly implements transition (6) in figure 6.8, statement `/*6*/` correctly implements transition (2). The running thread leaves and is blocked by the monitor due to the unestablished condition.

If neither a thread `c` stuck on an established condition nor a thread `t` trying to acquire the monitor is found, then no additional statement is executed. In this case, Invariant 2 remains established because no threads are blocked by the monitor.

The path with the additional statements `/*4*/` and `/*5*/` is executed if a thread `c` stuck on an established condition is found. The monitor is transferred to this thread `c`. This path correctly implements transitions (5) and (4) in figure 6.8. Invariant 2 remains established because the monitor is owned by the thread `c`.

The path with the additional statements `/*2*/` and `/*3*/` is executed if no thread `c` is stuck on an established condition, but a thread `t` trying to acquire the monitor is found. The monitor is transferred to this thread `t`. This path correctly implements transitions (5) and (3) in figure 6.8. Invariant 2 remains established because the monitor is owned by the thread `t`.

6.2.4 The Verified Classes

This subsection fills the gap to the formally correct implementation. As mentioned, Rudich et al. have provided the PIPNode implementation in listing 6.4. A PIPNode has a `currentPriority` field and a `link` pointing to another PIPNode instance. This `link` field is used for either expressing an “owned_by” or “blocked_by” relation. The protected array `priorities` is used as a multiset to perform the bookkeeping operations as outlined in section 6.2.1. Additionally, there is an `acquire` and `release` method.

Listing 6.4: PIPNode

```
public abstract class PIPNode
{
    protected PIPNode link; // represents either an "owned_by" or
        a "blocked_by" relation
    protected int currentPriority; /* 0 ... Kernel.
       NumOfPriorities - 1*/
    protected int[] priorities = new int[Kernel.NumOfPriorities];
```

```
public PIPNode(int defaultPriority)
{
    link = null;
    for (int i = 0; i < priorities.Length; i++) { priorities[
       i] = 0; }
    priorities[defaultPriority]++; //
    this.currentPriority = MaxPrio(); // == defaultPriority
       of course
}

private  int MaxPrio()
{
    int i = priorities.Length - 1;
    while ((i > 0) && (priorities[i] == 0)) { i--; };
    return i;
}

private void updatePriorities(int from, int to)
{
    int oldCurrentPriority;

    oldCurrentPriority = currentPriority;
    if (from >= 0) priorities[from]--;
    if (to >= 0) priorities[to]++;

    currentPriority = MaxPrio();

    if((link != null) && (oldCurrentPriority !=
       currentPriority)) {
        link.updatePriorities(oldCurrentPriority,
           currentPriority);
    }
}

public void release(PIPNode n)
{
    n.link = null;
    updatePriorities(n.currentPriority, -1);
}

public void acquire(PIPNode n)
{
    if (n.link == null) {
        n.link = this;
```

```
                updatePriorities(-1, n.currentPriority);
            } else {
                this.link = n;
                updatePriorities(-1, this.currentPriority);
            }
        }
    }
```

It has been proven that priority inversion will never occur in graphs of PIPNodes [45].
The gaps in between the system calls, Lock (6.1), Unlock (6.2) and AwaitCondition (6.3), and the PIPNode implementation is closed by the PIPThread (6.5) implementation and PIPMonitor (6.6) implementation. Both are derived from the PIPNode. Hence the three system call implementations have been mapped to the provably correct PIPNode implementation and will therefore themselves inherit correctness.

Listing 6.5: Thread

```
public class PIPThread : PIPNode
{
    public int state = Ready;

    public PIPThread(int defaultPriority):base(defaultPriority)
        {/../}

    public PIPMonitor blocked_by
    {
        get { return ((PIPMonitor)link);}
        set {
            if (value == null) link.release(this); else value.
                acquire(this);
        }
    }
}
```

Listing 6.6: Monitor

```
public class PIPMonitor : PIPNode
{
    public Queue awaitingCond;
    public Queue awaitingLock;
```

```
        public PIPMonitor(int defaultPriority):base(defaultPriority)
            {/../}

        public PIPThread owned_by
        {
            get { return ((PIPThread)link); }
            set {
                if (value == null) link.release(this); else value.
                    acquire(this);
            }
        }
    }
```

6.2.5 Conclusion

Given a graph of `PIPMonitor` and `PIPThread` instances that excludes priority inversion, we have verified that the implemented PIP operations on this graph are correct with respect to a validated PIP Specification in section 6.2.2. Hence our PIP implementation is provably correct.

6.3 Decoupling Threads with Lock Free Data Structures

According to chapter 5.3.2, lock free data structures are used to isolate real-time threads from non real-time threads. This chapter shows how lock free buffers and queues are implemented. Listing 6.7 shows an implementation of a bounded buffer based on a linked list. The buffer, derived from Herb Sutter [50], is meant to be operated by exactly one producer and one consumer thread. This article also explains why no data races occur.

The implementation by Sutter implies polling. The procedure “Remove” may return a void value. Hence the buffer must be polled to get elements out. Considering resource consumption, extensive polling is not a feasible option. The improvement compared to Sutter’s implementation is a solution that gets around polling by using the signals introduced in chapter 3.4.3. The implementation shown in figure 6.7 synchronizes the consumer and the producer.

After an item has been appended, the lease counter assigned to `signal` is incremented by one. This allows the consumer thread to run once through the procedure `Remove`. Thus the lease counter assigned to `signal` represents the number of available items in the buffer. The consumer will be blocked when trying to consume from the empty buffer.

Using a boolean condition for synchronizing a producer and a consumer would be an alternative. But boolean conditions are bound to monitors. A real-time producer would propagate its priority to a non-real-time consumer when trying to enter the locked monitor, clearly, an undesired side effect of the PIP in the context of real-time processing since it could render the real-time thread schedule unpredictable.

A generalized concurrent lock free queue [51] for more than one producer and consumer can be implemented in a similar way, since raising and catching operations on signals are atomic.

Listing 6.7: Lock Free Buffer

```
BoundedBuffer = OBJECT
 VAR first, divider, last: Item; signal : SYSTEM.SIGNAL;

 PROCEDURE & Init();
 BEGIN
    NEW(first);
    first.val := dummy; divider := first; last := first;
 END Init;

 PROCEDURE Append(x: Item);
 VAR tmp : Item
 BEGIN
   (*trim unused nodes in linked list*)
  WHILE (first # divider) DO
    tmp := first;
    first := first.next;
    pool.recycleInstance(tmp);
  END;
  last.next := pool.getInstance();
  IF last.next # NIL THEN
    last := last.next; (*publish it *)
    AosKernel.Signal(signal,1);
  ELSE
    (*drop item, no memory available *)
  END;
```

```
  END Append;

  PROCEDURE Remove(): Value;
     VAR result: Value;
  BEGIN
   result := NIL;
   AWAITEVENT SYSTEM.SIGNAL, signal;
   result := divider.next.val;
   divider := divider.next;
   RETURN result
  END Remove;

END BoundedBuffer;
```

6.4 Elastic Garbage Collection

Garbage collection in embedded systems is constrained by limited resources. Therefore our implementation strategy pays particular attention to the following three issues.

- **Impact on coexisting threads**. Since the number of processor cores is usually still very limited in embedded systems, the garbage collector needs to share the cores with all other threads. Allocating a dedicated core is not an option, since this creates interference and has to be resolved somehow.
- **Efficient use of RAM.** Every garbage collector strategy creates more or less an overhead in terms of memory requirements. Unfortunately RAM memory in embedded systems is even more precious than in traditional PCs because there is often no secondary mass storage available.
- **Overall runtime costs.** Additional trade offs are the overall costs a garbage collection iteration causes. This is a real issue for battery powered embedded devices.

The implemented garbage collector belongs to the class of tracing mark and sweep collectors. A copying collector that divides the heap into two disjointed semi-spaces called “from-space” and “to-space” is not an option on embedded

devices, because half of the heap would become unavailable . Both Tracing and Sweeping are interruptible, but not incremental.

6.4.1 Scheduling the Garbage Collector

Chapter 5.3.3 gives on overview on how the garbage collector thread coexists with the rest of the thread ecosystem. This chapter follows up on this topic. The term “elastic” in the title refers to how memory allocating threads pull the garbage collector’s runtime priority up to their own priority . Garbage Collection has no “first-class citizen” status in the system but obeys exactly the same rules as all other threads, especially regarding the PIP. The garbage collector implementation is an interesting case in which the PIP not only fixes a scheduling anomaly but takes the role as the thread orchestrating concept, as mentioned in the list of contributions (1.4).

Memory allocating threads are synchronized with the garbage collector via a boolean condition called `memoryAvailable` . The desired side effect is that the garbage collector inherits the highest priority of all threads currently waiting for heap blocks. If the statement `Heap.CollectGarbage()` was not included in the `EXCLUSIVE` section, garbage collection would only run with the collector’s default priority.

Listing 6.8: Garbage Collector

```
GarbageCollector = OBJECT
    VAR memoryAvailable : BOOLEAN;

    PROCEDURE Trigger();
    BEGIN{EXLUSIVE}
        memoryAvailable := FALSE;
        AWAITCONDITION(memoryAvailable);
    END Append;

BEGIN{ACTIVE}
    WHILE (TRUE) DO
        BEGIN{EXCLUSIVE}
            (* runs potentially with inherited priority *)
            AWAITCONDITION(~memoryAvailable)
            Heap.CollectGarbage();
            memoryAvailable := TRUE;
        END;
```

```
        (*runs on static priority*)
    END;
END GarbageCollector;
```

The two possible triggers for a garbage collection iteration are:

1. In order to reduce fragmentation, a proactive iteration is started when either the memory utilization rate hits a predefined threshold or a predefined maximum amount of time has elapsed since the previous iteration.

2. In order to satisfy a heap memory request when a heap block greater or equal to the requested block size is no longer available. A garbage collection iteration is done to find a fitting block while the requesting thread is passivated. If no fitting block is found, then the requesting thread remains passivated until a future iteration frees a fitting block. Note that there is potential for the memory allocating thread to starve.

6.4.2 Performance Considerations with regard to Stack Tracing

During the mark phase, the collector is supposed to visit each heap object at least once, starting from a so called “root set”. The root set consists of all module instances and thread instances in the system. These instances are the origin from where all other object instances are reachable. Since heap objects may also be anchored in stacks, stacks must be traced during the mark phase as well.
There are two basic strategies to locate pointers on stacks. Either the runtime system keeps track of all stack locations containing pointers or the garbage collector makes a conservative guess as to where on the stack valid pointers might be located. The process that the garbage collector implements to guess is called a *“stack pointer heuristic”*. At first glance, the former strategy looks more promising than just guessing where the pointers are, but empirical studies done with A2 [39] have shown that this is not necessarily the case. The reason why in practice a guessing stack heuristic is competitive with a precise lookup strategy lies in the architecture of today’s processors. Iterating linearly over a consecutive memory block like a stack and checking each machine word for plausibility is cheap because of the very effective cache prefetching capa-

bilities of modern processors. The general problem with guessing is that the computational complexity shifts from O(n) with n equal to the number of heap objects, to O(m) with m equal to the overall stack length. This is not intuitive to the application programmer who is assuming predictability in the order of O(n).

The implemented mark and sweep collector uses a marking stack with overflow handling [25]. During the mark phase, all thread stacks are inspected consecutively. Each word-aligned stack element is tested for plausibility to represent a pointer to a heap object. The decision about possible candidates is taken upon a conservative heuristics making sure that no pointer is missed. Missing a pointer candidate would cause a referenced heap object to be collected by mistake, which would of course severely harm the system's integrity. All candidates are then tested against the set of existing memory blocks. If candidates really do point to such an existing memory block, then they are traced recursively.
This verification process, of checking whether a candidate is a valid pointer or not, is computationally expensive and therefore influences the overall runtime characteristics of the mark and sweep collector. If the applied stack pointer heuristic produces only very few false positives, then the overall runtime behavior would not shift that much towards O(size of all stacks).

Our trick to minimize the number of false positives is adding a pointer to itself in the meta-data part of all heap objects. Figure 6.9 shows a generic heap object together with its meta-data. The type tag pointer (pointing to the type descriptor) is always aligned to address = 28 (mod 32). Three pieces of information are encoded at address = 24 (mod 32). The least significant bit is the 'M'ark bit. This bit is set during the mark phase and reset during sweep. The second least significant bit is the 'F'ree bit. It is set when a block is linked into the free list and reset when removed from the free list. All other bits are reserved for mirroring the 'self' pointer. This pointer is aligned with 0 (mod 32), 8 (mod 16), or 16 (mod 32).

A well-educated heuristical guess as to whether a pointer on the stack is really pointing to a heap block is performed in three steps.

1. Does the pointer refer to the currently allocated heap memory range?
2. Is the pointer aligned with 0 (mod 32), 8 (mod 16), or 16 (mod 32)?
3. Is the pointer mirrored at 24 (mod 32)? As a fortunate side effect, this test also fails when the mark or free bit is already set because valid pointers

are always machine word aligned, meaning 0 (mod 4) on a 32 bit machine.

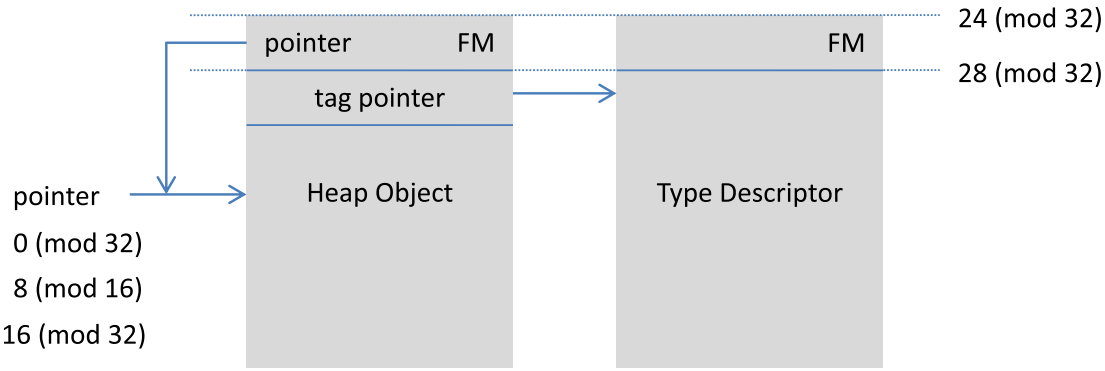

Figure 6.9: Memory Layout

Let us now discuss the rate of false positives based on the heuristics just explained. Two assumptions are made: First, the heap occupies the whole memory range and second, the heap and all stacks are filled with uniformly distributed random bits. Figure 6.10 shows the probability tree. Steps two

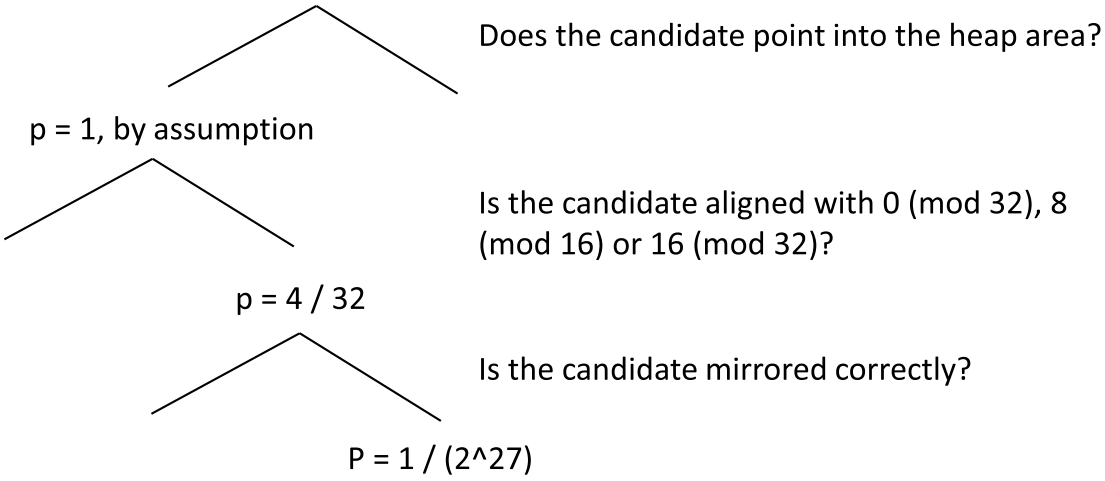

Figure 6.10: Decision Tree

and three are statistically dependent, the five least significant bits are already considered at stage two, thus the probability that a candidate is mirrored correctly by coincidence is just 1 divided by two to the power of 27 and not to the power of 32. Hence a pointer candidate is judged false positive only by about 0.0000000007 % of the time. This means that virtually no overhead is caused

by checking false positive candidates. Thus garbage collection's runtime complexity is proportional to the number of heap objects instead of to the overall amount of memory allocated by the different thread stacks.

6.4.3 Performance Considerations with regard to Write Barriers

Write barriers are necessary to avoid *"behind the back"* pointer assignments during the mark-phase of the garbage collector process. Theoretically, there are four different pointer assignment operations; 1) Assigning a heap object to a pointer located in another heap object, 2) assigning a heap object to a pointer located on the stack, 3) assigning an object allocated on the stack to a pointer also located on the stack, and 4) assigning an object allocated on the stack to a pointer anchored in a heap object. The latter operation does not exist in reality. Stacks are not explicit objects that can be referred to, rather they are invisible for heap objects. The stack to stack case can also be safely ignored, since it does not affect the heap. Thus only two cases remain.

Stack Pointer Assignment

When a heap object is assigned to a pointer located on the stack while garbage collection is in progress, no write barrier is launched. Instead the stack of the corresponding thread will simply be reinspected again. This saves runtime overhead at the expense of the garbage collector thread.

Heap to Heap Object Assignment

When a heap object is assigned to a pointer located in a heap object while garbage collection is in progress, then a write barrier is launched. Listing 6.9 below shows the write barrier logic formulated in pseudo code, which is compiled to an assignment operation like `rootObject.rootPtr := targetObject.targetPtr;`.

Listing 6.9: Mark Test

```
IF gcBusy THEN
    IF IsMarked(rootObject) & ~IsMarked(targetObject) THEN
```

```
            RegisterCandidate(targetPtr);
        END;
    END;
```

If the garbage collector has been interrupted *and* the `rootObject` has already been visited *and* the `targetObject` has not yet been visited, then the `targetPtr` needs to be traced again.

A few remarks on the penalty due to the additional write barrier if the garbage collector is not running. The first four assembler instructions in listing 6.10 show an ordinary pointer to pointer assignment operation. First, the target pointer address is loaded. Second, the target pointer value is loaded into register 2. Third, the root pointer address is loaded. And finally, the target pointer value in register 2 is stored to the root pointer address.
The latter four instructions implement part of the write barrier. First the global "gcBusy" flag located at address 200000H is loaded and compared to `FALSE`. The program counter skips the write barrier block if "gcBusy" is not `TRUE`.

The overall number of processor cycles necessary to compute an assignment operation with a write barrier is determined by the number of cache misses. According to listing 6.10, there are four potential cache misses possible.

The best scenario in terms of number of processor cycles is: If no cache miss arises at all, then the number of processor cycles needed to compute the assignment operation with a write barrier is determined by the number of executed instructions. Since four instructions are used for the core assignment operation and four instructions for checking the "gcBusy" flag, the overhead added by the write barrier is 100%.
The Worst scenario is: If all four cache misses occur, then the number of processor cycles needed to compute the assignment operation with a write barrier is determined by the cache misses. The overhead added by the write barrier is 33%, since three cache misses are at the expense of the pointer assignment operation.
The most likely scenario: The "gcBusy" flag is either cached because it is frequently used or because the according processor cache line has been locked down. Hence the overhead added by the write barrier is negligible since the number of processor cycles needed to compute the assignment operations is determined mainly by the three potential cache misses triggered by the first

three load operations.

Listing 6.10: Mark Test in Assembler

```
LDR R1, [FP,#8H]          (*potential cache miss*)
LDR R2, [R1] ]            (*potential cache miss*)
LDR R3, [PC, #-64CH] ]      (*potential cache miss*)
STR R2,[R3]
MOV R6, 200000H
LDR R7,[R6] ]             (*potential cache miss*)
CMP R0,R7
BEQ 20
```

While the garbage collector is interrupted, the write barrier's penalty is an order of magnitude beyond the previously outlined worst case scenario. The additional overhead is caused by the `IsMarked` and `RegisterCandidate` calls in listing 6.9.

Chapter 7

Use Cases

The three use cases from the medical IT field presented in this chapter belong to the dependable systems category and confront system developers with a significant challenge. Their discussion shows the substantial improvements made possible by the concepts presented in earlier chapters of this thesis.
The goal was to develop a wearable device able to cope with *multiple sensor types* and adaptable to a *variety of purposes*. It was supposed to synthesize different sensor types and to perform demanding high level analysis. An additional requirement was to provide a notification service over the air in case of relevant medical events. Diagnostic robustness was another important design goal.
The result shows convincingly that stationary diagnostic devices nowadays used by hospitals could, at least partially, be substituted by wearable devices. The presented device allows monitoring patients continuously, regardless of their location. A research prototype has been designed and produced in collaboration with XAI medica, a spinoff company of the Aeronautics University of Kharkov, Ukraine.
A project with a similar concept as the presented one is AMON by [2]. AMON proposes a wearable wrist device that correlates multiple parameters like blood pressure, ECG, body temperature, oxygen saturation and others in order to detect and report events, but only rudimentary high level analyses are performed on the device itself.
The only wearable diagnostic devices frequently used in current medical daily routine are Holter Electrocardiograms. Holter ECGs are conceptually single purpose and single parameter devices aimed to monitor the myocardial activity of a patient from a few hours up to several days.

Figure 7.1: Copyright by Maurice Grünig, Zürich

As previously mentioned, an important design goal was to provide a system with ultra robust detection capabilities, minimizing false positive alarms and of course also false negative classifications. The primary medical goal was not to provide detailed analysis of a patient's health state but to be able to reliably decide whether a patient is in a serious condition or not. In order to detect such potentially lethal conditions, inputs from different kinds of sensors are considered. A physiological system like a human shifting into a critical state often shows a variety of symptoms in parallel, which are redundantly detectable. Redundancy in sensing a human's health condition improves the quality of the prediction significantly. The proposed device is able to monitor two important physiological subsystems: Respiration and blood circulation. The former is monitored by measuring the thorax impedance, an electrophysiological measure. It could be complemented by the oxygen saturation. The latter is monitored by measuring the electrocardiogram of the heart plus the central and peripheral heat flow, the amount of heat (energy) diffusing from the body to its environment per time and surface unit. Especially, the first deriva-

tive of the difference from central to peripheral heat flow is of interest: It might be a hint of circulatory shock for instance if the distribution of the blood flow is altered in disfavor of the extremities. In addition, the acceleration of the device in three dimensions is measured. This allows detecting whether a patient is in a kinetically steady state to discard artifacts in the acquired data related to short and abrupt movements.

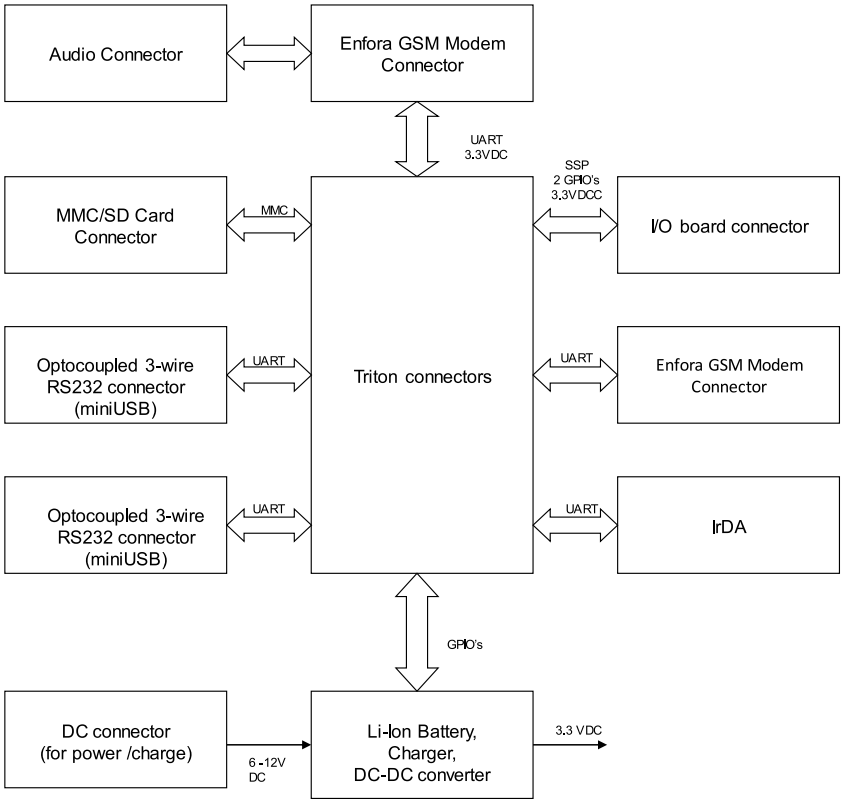

Figure 7.2: Board Schema

The schematic description above was taken from [49]. It shows all external input and output connectors and how they are connected to the central processing unit. GPIO stands for “general purpose I/O”, SSP for “Synchronous Serial Port” and MMC for “Multimedia Card”. All other abbreviations are self-explanatory.

7.1 Evaluation of the User Interface

Practical experience has shown that the device is lacking a simple way of indicating any state. Unfortunately, there are no means to provide any user feedback such as the battery's charge state, a warning if some sensors have not been connected correctly or an estimate about the remaining capacity of the data memory.
Also, Bluetooth instead of IrDA would have been the preferred option for near field communication.

7.2 Real-time Monitoring

Real-time data visualization aims at giving an immediate overview of the state of a physiological system. For this purpose a simple viewer application has been implemented by Alexey Morozov. It allows zooming and recording live feeds tapped from the device's serial connector. Chapter 7.2.1 lists all channels that are visualized. Figure 7.3 shows a demonstrator that has been im-

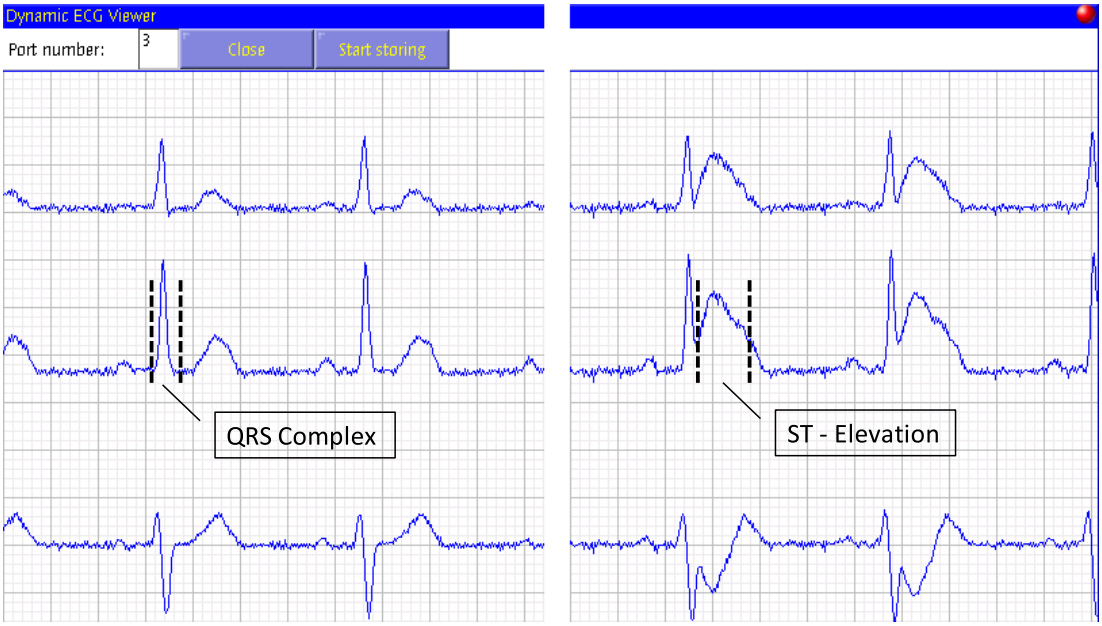

Figure 7.3: Screenshot Viewer

plemented on the occasion of the twenty-fifth anniversary of the computer science department at ETH Zurich [8]. In order to demonstrate the power of the

concept, Alexey Morozov has developed a simulator that injects a so called ST-elevated myocardial infarction into a healthy electrocardiogram. The left hand side in figure 7.3 shows a normal, synthetic sinus rhythm. On the right hand side, the same sinus rhythm is shown but this time superposed with an artificial ST elevation. Visitors were given an opportunity to play with the simulation by toggling the hazardous ST-overlay on and off on their own ECG with the black button on the front of the device.

This viewer tool is also used for checking if the sensors have been applied correctly and the acquired data is of the expected quality.

7.2.1 Data Channels

The medical I/O board depicted in figure 7.2 delivers sixteen data channels in parallel at a rate of five hundred samples per second where the sampling rate is determined by the high frequency ECG data characteristics. For other signal types, five hundred samples per second is hard to justify, but for the sake of simplicity, all channels are delivered uniformly.

7.2.2 Evaluation

When a physician senses a patient's pulse with his or her finger tips, the data displayed on the monitor shall correlate with what the physician feels. Thus, any latency caused by the pipelined data processing setup is problematic. The I/O board buffers the acquired data for a quarter of a second before it feeds it into the processing pipeline, which leads to a delay beyond what is acceptable in practice. A counter measure would have to include dynamic batching, depending on a latency versus power efficiency tradeoff.

Table 7.1: Data Channels

Channel	Remarks
ECG	Eight ECG channels are acquired; I, II, V1, V2, V3, V4, V5 and V6. There are four channels missing for a fully-fledged standard twelve lead ECG. Channels III, aVR, aVL, and aVF are simple linear combinations thereof and computed on the fly. III := II - I; aVR := (-I - II) / 2; aVL := (I - III) / 2; aVF := (II + III) / 2. The resolution of the ECG signal is 0.005 [mV].
Respiration	Two channels are dedicated to the body impedance acquired across the thorax. The first channel delivers a low pass filtered signal, the second channel delivers the difference from the first channel to the actual signal.
Acceleration	The acceleration in the X, Y and Z directions is delivered by three channels in parallel. Plus minus 2 [g] is the measurable range.
Heat Flow	Two additional channels are allocated for the central and peripheral heat flow sensors. The input values range from 0 to 400 [W / m2].
Battery state	A 16 bit count down indicates the battery's current charge.

7.3 Data Stream Recorder

Recording data over a longer period of time is best practice for detecting potentially seldom events like cardiac arrhythmias for instance. A Holter ECG is state of the art for detecting and quantifying such cardiac arrhythmias. A patient wears a device that records only two or three ECG leads for at least 24 hours. Data analysis is done later off line on a workstation. There are fully automated off line tools available for detecting and quantifying interesting events. The presented monitor could easily substitute such a Holter ECG. It stores data and records it on a standard FAT32 formatted fully interoperable MMC memory card. In addition, it could well be used for sophisticated high level analysis like detecting sleep apnea, where the body impedance combined with heart rate and kinetic movements derived from the built-in acceleration sensors would give the big picture about the dynamics of the different sleep phases.
The device has been used by Prof. Dr. Dr. Stephan Marsch at the University

Hospital Basel for determining the stress level of medical trainees involved in simulated critical care situations.

7.3.1 Implementation

The MMC driver by Thomas Kägi-Trachsel was originally implemented for Minos [24] and adopted to Realtime Oberon. Interrupt handling and conditional synchronization had to be reviewed and ported according to section 3.4.1. The adaptions necessary to be compatible with the FAT32 File System [9] borrowed from [39] have been rather invasive. The Framework was adapted according to the design pattern presented in chapter 5.4.2 in order to avoid deadlocks. Another issue was finalization. As introduced in chapter 3.4.4.

7.3.2 Evaluation

The amount of raw data stored per hour was equal to approximately 57 [Mbyte]. (16 channels multiplied by 2 Bytes multiplied by 500 samples per second multiplied by 3600 seconds per hour). Different data reduction strategies have been applied: the battery charge state has been dropped and the two heat flow channels have been down sampled to 10 samples per second. All other data channels were compressed using a lossless Huffman based compression method proposed by the European standardization committee [1]. This has reduced the amount of data significantly. The device is now usable as a 24 hour recorder thanks to its battery capacity of 1 [Ah] and commercially available flash memory cards of 1 [GByte] capacity.

7.4 Hazardous Event Notifier

The third use case finally shows the full power of the concept. The device's functionality has been upgraded to perform autonomously rich analyses in realtime. It scans the data stream for a set of predefined events and emits alerts via a built-in GSM modem.

This chapter presents a setting for detecting and reporting ventricular tachycardia events. The chosen tactics to detect such events is simple and straight

forward: scanning the ECG data stream for heart rates above 120 beats per second. There are several reasons why heart rates rise beyond 120 beats per second, the most prominent reason is heavy physical activity. Thus false positives cannot be excluded with our approach. In return, false negatives are unlikely because a tachycardia always manifests itself in the form of a rapid rhythm of the heart.

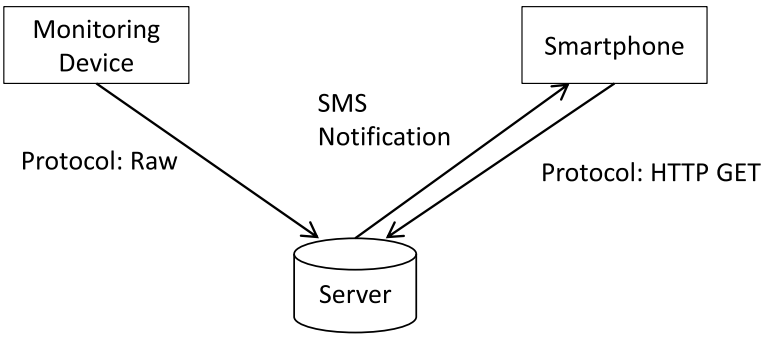

Figure 7.4: Event Notification Path

Figure 7.4 shows the physical system setup: An A2 [39] based webserver is used as a data relay. The monitoring device transmits the event related data to the server, and the server in turn sends an SMS to the receiver's smartphone. The SMS encodes the URL of the data that is supposed to be fetched by the smartphone. Figure 7.5 shows three screenshots of the smartphone application developed for the Windows mobile platform. The application parses all received SMS for a particular tagged SMS and extracts the required URL for downloading the data by means of an ordinary HTTP GET request.

The viewer application's user interface has been kept very simple. It shows the three most recent alerts, allows zooming, makes simple measurements and calls back a dedicated care team.

7.4.1 Implementation

Figure 7.6 shows the structure of the threads and blocking buffers used to perform the requested notification services.

This case study relies on two key software frameworks: a QRS detector developed by Alexey Morozov and a Support Vector Machine implemented by Bruno Koller [26]. Let us first discuss the QRS detector.

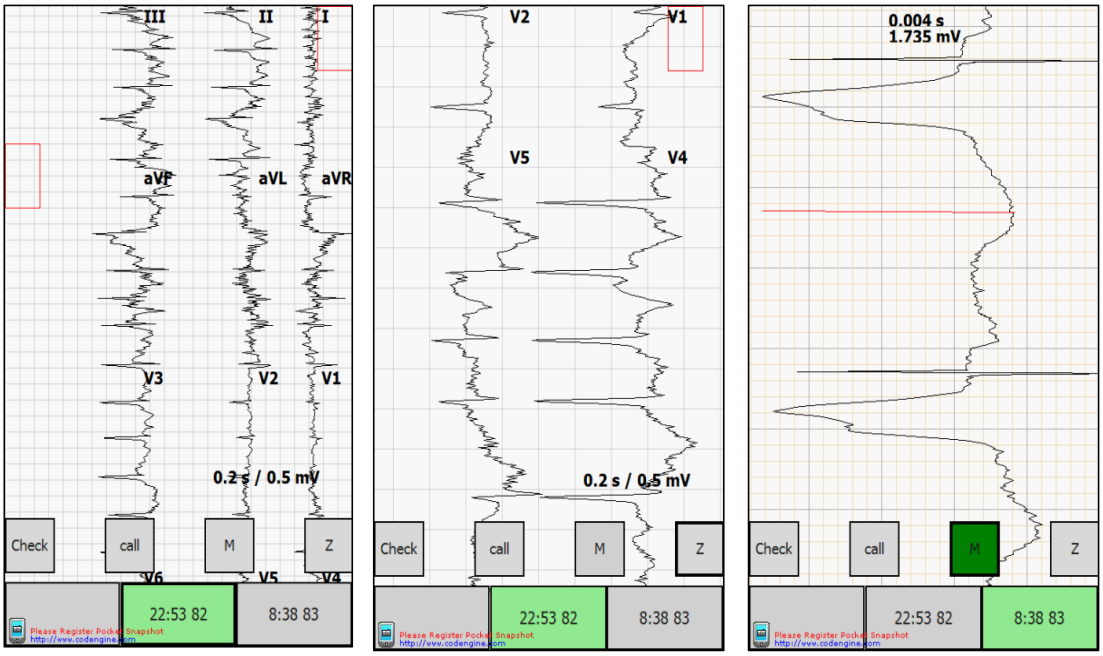

Figure 7.5: Mobile Phone based Receiving Application

QRS Detector

Analyzing an ECG signal first requires identifying individual heartbeats encoded in the signal. A heartbeat detector scans an ECG data stream for so called QRS segments. Figure 7.3 highlights such a segment on the left. The following quote is taken from [31]. It describes the heartbeat detector originally developed for the presented use case and later reviewed and refactored for validating a Process-Oriented Streaming System Design Paradigm for FPGAs.

> To detect ECG heart beats in a single channel we use a variation of the algorithm proposed by Pan and Tompkins [40]. The algorithm is based on linear digital filtering combined with a non-linearity. As a result, the detector returns the position of a probable ECG beat in a given single channel signal. Although the single channel QRS detection algorithm performs well on standard ECG records from the MIT/BIH database, its quality can be insufficient for a practical multichannel ECG monitoring system. A multichannel QRS de-

> tection can lead to improved performance over the single channel approach by the use of more information about the overall spatio-temporal signal. Here we chose a strategy proposed in [LJC94], which is based on combining preliminary QRS detection results from multiple channels [...] with some adaptation to real-time processing.

Once a QRS segment has been detected, the heart beat is analyzed. ECG wave analysis results in the extraction of multiple features, such as type of peak sequence, onset, offset and magnitude of the waves within the beat. For this a derivative-based wave boundary delineation algorithm has been used as presented in [28].

At this stage, all operations are performed using fixed point arithmetic. The ECG wave analyzer is capable of extracting about fifty different features:

- *RR - distance:* The peak to peak distance in between the current and previous QRS complex expressed in milliseconds. As a side effect, the median RR distance over a configurable sliding window is also tracked.
- *Amplitude:* The current QRS' amplitude is measured in millivolts. Calculated for the non-redundant channels I, II, V1, V2, V3, V4, V5, V6; the median amplitude over a configurable sliding window is tracked for each channel.
- *QRS length:* The current QRS' length is measured in milliseconds. Also calculated for channels I, II, V1, V2, V3, V4, V5, V6; additionally, the median QRS length over a configurable sliding window is tracked for each channel.
- *ST - Elevation:* The current ST - elevation is measured in millivolts. Calculated for channels I, II, V1, V2, V3, V4, V5, V6; the median ST - elevation over a configurable sliding window is tracked for each channel.

A configurable subset of the eight channels are tracked by default. According to the enumeration above, a maximum of fifty features are extracted each time a heartbeat is detected. Namely the RR distance *plus* its median *plus* eight channels *times* the current and median value *times* the amplitude, QRS length and ST - Elevation. Thus the maximum length an ECG feature vector may attain is fifty elements.

Support Vector Classifier

The second important software framework used is a Support Vector Classifier implemented by Bruno Koller [26]. Among the popular machine learning techniques, the Support Vector Machine approach has been chosen because of its modest runtime requirements. The algorithmic complexity classifying a feature vector is O(n) with n equal to the feature vector's length.
[26] has also provided a generic framework for implementing similarly conditioned problems aiming at pipelined data processing.

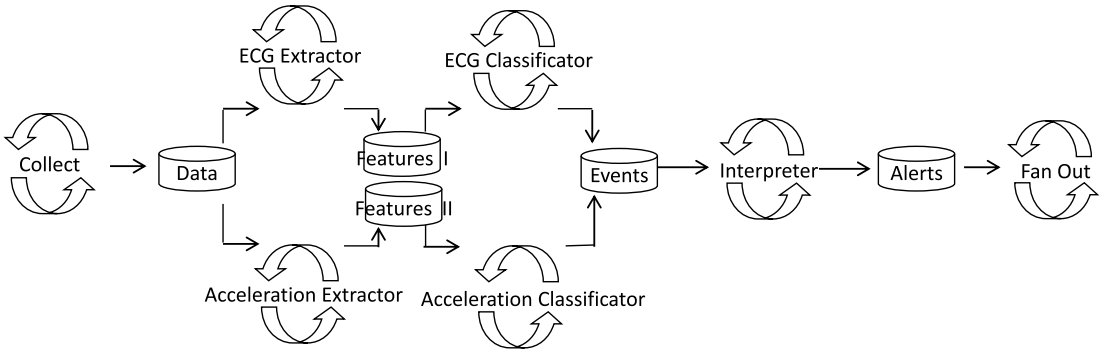

Figure 7.6: Processes and Interconnects

The two feature buffers depicted in figure 7.6 are each fed by a separate feature extracting thread. Both threads store feature vectors of a specified length into these buffers. Each element of such a vector contains a feature such as the heart rate or the maximum acceleration magnitude for instance.
Let us first focus on the upper data path in figure 7.6. Whenever a QRS complex is detected by the ECG feature extraction thread, it compiles a new feature vector with only one element, namely the median RR distance. The downstream classification thread processes data vectors from the feature vector buffer and classifies these vectors into event categories, which is straight forward in the case of linearly separable single element vectors. The median RR - distance is classified against a threshold, two kinds of events are generated, namely "below ventricular tachycardia RR - distance threshold" and its complement. The events are stored in the "Events" buffer.

Some explanations regarding the lower data path in figure 7.6: the feature extraction thread handling acceleration data evaluates a binary feature expressing the degree of kinetic activity. It is used to reset the ECG features

when the patient is not in a kinetically steady state. The thread continuously calculates a synthetic signal feature by adding the absolute values in the X, Y and Z directions and by down-filtering it by a factor of four with a wavelet filter. This flattens the aggregated signal without loss of generality. The newly computed synthetic feature signal's magnitude is compared against a threshold; two kinds of features are extracted and stored in the buffer "Features I", namely "below threshold" and "above threshold". The downstream classification process uses these feature vectors for classification according to "magnitude above threshold" and "magnitude since five seconds below threshold".

An interpretation thread consumes the shared event buffer and generates alerts, which in turn are then forwarded to a receiver's smartphone by the "Fan Out" thread via the GSM modem connected to UART port 1. The interpretation thread correlates the different events detected by the two upstream classification threads. An alert is only issued if positive ventricular tachycardia events have been reported continuously over a five second period and if the acceleration magnitude has not hit the threshold during this time span. The mentioned "Fan Out" thread takes care of sending reports in case an alert has been raised. It gathers the related raw data, compresses it and sends the data to the built in GSM modem. The thread also takes care to not send redundant reports to the receiver.

7.4.2 Evaluation

On behalf of Prof. Dr. med. Paul Erne a real patient was supervised during four nights in October 2008 at the Kantonsspital Lucerne. The patient refused to stay at the intensive care unit and therefore had to be supervised remotely. Reporting lethal ventricular tachycardia was the primary task the device was expected to perform. During four nights no event was signalled, and one false positive alert was reported.

The battery capacity delivers a sufficient amount of energy to perform the described computation for about twenty four hours.

Empirical studies by Bruno Koller [26] on the event detection capabilities of the presented ECG analyzer have shown that the extracted set of features mentioned in the previous section is insufficient for a holistic, fine grained and reliable ECG event detection. The following features in particular are missing: T-wave alternans, the QT-length and the rotation of the QRS complex.

However, there would be enough computing power available to enhance the feature extraction mechanism to meet the demands and their implementation on top of the presented system / language architecture even by non-system-programming experts in a reasonable time and without compromising the final system's dependability.

Chapter 8

Evaluation

This chapter evaluates the presented generic software framework for implementing dependable data driven embedded systems with other state of the art systems mentioned in chapter 1.2. The treated aspects are inspired by the Real Time Specification for Java [5] where Scheduling, Memory Management, Synchronization, Resource Sharing, Asynchronous Event Handling and Asynchronous Transfer of Control have been identified as the key aspects for implementing real-time systems. All these topics will be treated by the next sections.

8.1 Scheduling

Runtime Predictability requires a suitable scheduling model. Optimality, uniformity, and power awareness are the issues treated in the next paragraphs.

8.1.1 Optimality

First, a few remarks on *optimality*. A scheduler is called “optimal” if, and only if, it always finds a feasible schedule whenever one exists in theory. A well-known example is the earliest deadline first (EDF) scheduler that, however, requires a preemptable runtime system. Unfortunately optimal schedulers perform poorly in the case of computational overload and predictability is lost in general [30]. However, predictability is a very important property a scheduler is expected to feature. EDF schedulers therefore perform *acceptance tests* whenever a new

task shows up. A task is only included in the current working set if it is compliant with all previously accepted tasks. Acceptance decisions are taken based on worst case execution time, release time and the deadline of the new thread. More sophisticated tests also take a thread's urgency into account. At first glance the concept of acceptance test looks promising because it optimizes a cost function, for instance the number of threads missing their deadlines, based on the properties of all threads belonging to the current working set plus the properties of the new candidate. However acceptance decisions are local decisions taken by a greedy algorithm merely considering the close past and the near future. This leads to only local optimal decisions, a greedy acceptance test will most likely not find a global optimal solution in the case of overload.
However, optimality has not been a concern of priority in the context of Realtime Oberon. Realtime Oberon comes with a non-optimal fixed priority scheduler with predictable behavior in the case of overload. The actual price for non-optimality will depend on the specific use-cases.

8.1.2 Uniformity

The second issue is about uniformity. Importantly, the priority inversion protocol has been smoothly integrated into the scheduling concept. It is uniformly applied to all activities. Including interrupt handling. This is conceptually clean and easy to deal with.
LynxOS for instance, comes up with a patented solution called *Priority Tracking* to get around priority inversion. According to LynuxWorks [18] it works as follows:

> [Interrupt handling] kernel threads begin their existence with a very low priority as created by a driver. When a user thread opens the device, the kernel thread promotes its own priority and "inherits" the priority of the user thread opening the device. If another user thread of higher priority opens the device, the kernel thread bumps its priority up to match the other thread; when the I/O is complete the kernel thread returns to the next pending thread's priority level, or to its starting level.

It is immediately clear that if the interrupt handling kernel thread runs at the same priority as its client thread does, no priority inversion will occur. However, running the interrupt handling thread at the same priority as the client thread is not always beneficial. A counterexample: A high prioritized interrupt handling thread copying data from a hardware register into a bounded buffer, which in turn is consumed by a low prioritized thread, is sound and very common. The PIP just has to make sure that the high priority producer is not uncontrollably blocked when putting data into the bounded buffer. The ability to make precise distinctions between consumers and producers threads regarding their runtime behavior is a key requirement for mapping real world constraints. LynxOS does not allow this with its simplistic Priority Tracking approach.
Other real-time systems are completely priority inversion agnostic when handling interrupts. The ThreadX [29] system for instance and many other kernels *do not participate in the overall priority scheme although it governs it. Interrupts are handled by extensions to the kernel called "drivers", which also function outside normal prioritization [18].*

8.1.3 Power Awareness

A third issue regarding the scheduling strategy is power awareness, which is an aspect of crucial importance in a runtime system used for operating battery powered devices. Time slice interrupts that periodically reevaluate the running process are avoided, because if the reevaluation process takes no action, then the consumed computing cycles simply have been wasted. Scheduling is driven by raised events instead. Hence pointless context switches are eliminated.
This is not the case in Java, even if the Java Runtime would do without time slicing. The generic Java code fragment in 8.1 taken from the Java Documentation [7] shows why. When a waiting thread on `obj.wait()` is notified, it must reevaluate the condition. If another competing thread has falsified the condition in the meantime, the thread is going to wait again. The result is two pointless context switches. In Realtime Oberon, the condition is guaranteed to be true right after `AWAITCONDITION`.

Listing 8.1: Conditional Synchronization in Java

```
synchronized (obj) {
        while (<condition does not hold>)
```

```
            obj.wait();
        // Perform action appropriate to condition
}
```

8.2 Memory Management

In principle there are three different memory domains a thread operates on; the heap, the stack, and on statically allocated global data. The next three sections evaluate the real-time tradeoffs.

8.2.1 Heap

Hard real-time threads using a managed heap pose a challenging problem. There are two basic strategies to cope with this challenge: Either the heap management overhead is shielded from hard real-time threads, or hard real-time threads are completely decoupled from the managed heap.
An implementation aiming at the first option has been presented by Bacon [4]. Its garbage collector strives to deliver short deterministic pauses of a predefined maximum length, thus preserving a minimum utilization rate for real-time threads over a sliding time window. This implementation exhibits two drawbacks: a) the granularity of the short deterministic pauses might be inappropriate for a particular application and b) extra heap space is needed in addition to the heap space required by the application.
In contrast, the decoupling approach is based on introducing an unmanaged heap that allows real-time threads to allocate and release memory explicitly [39], [5]. The disadvantages of this approach are a) potential memory leaks and dangling pointers due to mismanagement and b) references from the managed part of the heap pointing to the unmanned part. These are not straight forward to handle since unmanaged heap memory can be released arbitrarily by its owner. As a general invariant, a heap object should never be referenced by an object with a potentially longer lifetime. Different countermeasures are popular. The unmanaged heap could be declared as immortal. In this case, a persistent object may be referenced by any other object due to its particular lifetime property.

Realtime Oberon runs a single heap memory in combination with a non-transparent garbage collector that obeys exactly the same rules as all other threads, especially also regarding the PIP. It was influenced by related work on Scheduling Garbage Collection in Embedded Systems done by Henriksson [15]. His work has heavily influenced Oracle's commercial implementation of the Real-Time Specification for Java [5]. *The garbage collector runs as one or more threads. These threads run at a priority that is lower than all instances of the hard real-time threads so that critical threads may preempt the collector. In this way, critical threads are shielded from the effects of garbage collection. One of the tuning parameters is the maximum priority of the garbage collector. By default, the garbage collector runs at its initial priority, which is below that of the noncritical real-time threads. But as memory grows short, the virtual machine will boost or raise the priority of the collector to the maximum configured priority [12]*.

The proposed Realtime Oberon solution outlined in listing 6.8 also aims to shield hard real-time threads from the effects of garbage collection, similar to Henriksson's GC [15]. An improvement to Henriksson's scheduling strategy has been proposed. Whereas in [15] heuristics take care of steering the collector's priority within a preconfigured band, our solution postulates a very natural way of adjusting the priority: the garbage collector inherits the priority from those threads that currently depend on the garbage collector's progress freeing memory.
If the garbage collector's priority is limited to a reconfigured band, priority inversion is likely, whereas our solution avoids priority inversion by design.

8.2.2 Stack

The stack is predestinated to host structures of a known and limited lifetime that do not need to be shared. Allocation and deallocation of stack memory is very efficient and predictable, which makes it an attractive option particularly for hard real-time threads.
In Java or C#, compilers optimize behind the scenes what data is stored on the stack and what in the heap. Programmers cannot express their preferences. Therefore the designers of Real-Time Java [5] had to reintroduce so called *scoped memory areas*. Each area is assigned to exactly one real-time thread. Scoped memory areas are intended to accommodate objects with a known

lifetime and are uncollected. They are reclaimed at the end of their thread's lifetime. The analogy of scoped memory areas with ordinary stacks is evident. In Realtime Oberon we make use of the non-optimizing compiler that supports a fully-fledged stack and allows programmers to decide where which data will be allocated.

8.2.3 Global Data

Analogous to Java where global data is stored in static class members, global data is tightly coupled to Oberon modules. A module is an entity of code and data, namely the module variables. A variable's lifetime starts when the hosting module is loaded and then persists. Working with module variables does not cause any costs at the expense of (hard real-time) threads, given the module has been loaded or statically linked and the variables are not shared.

8.3 Synchronization

Synchronization includes two aspects: mutual exclusion and conditional synchronization.

8.3.1 Mutual Exclusion

Many programming languages originally designed for general purposes have later been instrumented for real-time programming. Unsurprisingly they lack expressiveness regarding time constraints.

Java provides a `synchronized(){}` statement to protect critical code sections. Unfortunately, `synchronized{}` does not allow to declare a timeout that a thread is trying to enter the critical section. This is suboptimal in the case of real-time threads. There is no way to express the maximum amount of time a thread waits to enter a critical section.

In order to improve the time predictability of programs written in Realtime Oberon, all language statements that potentially passivate the invoking thread allow specifying time constraints. Notably these statements are `EXCLUSIVE`

(3.2), AWAITCONDITION (3.3) and AWAITEVENT (3.4). These timely constrained statements facilitate reasoning on worst case execution times of real-time threads.

8.3.2 Conditional Synchronization

A popular way programming languages / runtime systems support conditional synchronization is in terms of some wait() and notify() statements / API calls. Figure 8.2 shows a Java sample.

Listing 8.2: Conditional Synchronization in Java with Timeout

```
synchronized (obj) {
        while (<condition does not hold>)
             obj.wait(timeout);
        ... // Perform action appropriate to condition
}
```

`wait(int timeout)` is used to passivate a thread and `notify()` or `notifyAll()` to resume threads. Java threads are supposed to reevaluate the boolean condition they are waiting for after they have been resumed because the condition could have been falsified in the meantime. The maximum amount of time a thread is going to be passivated is specified by the optional parameter `timeout` of method `wait()`.

The improvement Realtime Oberon comes up with in the context of conditional synchronization is an extended PIP adaption. Combining priority inheritance with conditional synchronization is unique to the best of our knowledge. This combination has been made possible by Realtime Oberon's atomic AWAITCONDITION statement.

Furthermore, to the best of our knowledge, no Java Virtual machine propagates priorities while a thread is passivated on `wait()`. The invoking thread implicitly releases the monitor by calling `wait()` and is not going to propagate its priority to a lower prioritized thread that acquires the monitor in order to establish `condition`. Since passivating the thread and reevaluating the boolean condition are spread over two different statements (while / wait), a runtime system has little chance of ensuring that another thread manipulating the condition is running at least at the same priority as the waiting thread is. There

are other reasons for calling `wait()` within a monitor, in addition to waiting for a boolean condition to be established. Whereas the `AWAITCONDITION` statement clearly expresses the correlation between the boolean condition and the expected runtime behavior, which makes a holistic PIP adaption feasible.

8.4 Resource Sharing

Sharing resources among threads brings up the priority inversion problem as introduced in chapter 4.2. A few concepts ([48], [33]) to cure this anomaly have been proposed and implemented. Obviously it is hard to get it right: It has been shown that widely used real-time operating systems are substantially flawed and already fail with test cases of modest complexity [52]. Zöbel and Polock have also published work on conformance testing of Priority Inheritance Protocols [53].

The other issue treated in this chapter is the applicability of the PIP implementation to application programming.

8.4.1 Correctness of the Priority Inheritance Protocol Implementation

Various priority inheritance implementations have been reviewed by Dieter Zöbel and David Pollock [52]. They have shown that implementing the PIP is obviously not straight forward, since popular implementations have been identified as heavily error prone. To get an idea of the complexity hidden in the PIP we take a closer look at an example taken from [52]. Figure 8.1 needs to be read as follows: Process P1 runs initially at priority 10. It requests critical section *d* at t = 02 and gains access to it at t = 03. At t = 04 it requests critical section *a* and gains access before it is preempted by process P2 at t = 06, because P2 runs at priority 12 and so forth. The test suite used to unfold faulty inheritance was configured for five processes with basic priorities ranging from 10 to 18 and critical sections from *a* to *d*. The, at the time popular, QNX 6.2 realtime OS has unveiled significant weaknesses. “During the time units 29 - 31 inversion is possible in that process P2 should have priority 18. The reason for this error possibly arises from a non-transitive implementation of inheritance, in this state from process P5 via P3 to P2” [52]. Process P2 owns critical sec-

tion *a*, which is requested by process P3 owning critical section *c*, which is in turn requested by process P5. Hence process P2 is supposed to run at P5's priority.

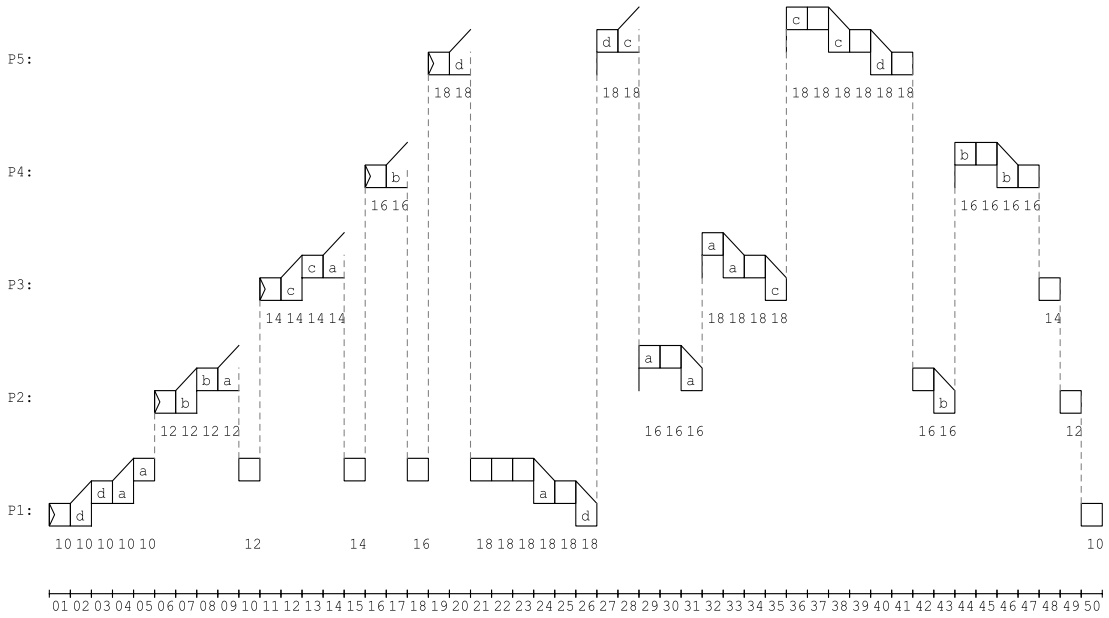

Figure 8.1: QNX 6.2

QNX 6.2's predecessor was also error prone, interestingly due to different defects. The bugfix from version 6.1 to 6.2 introduced new defects, thus there is strong evidence that getting the PIP implementation right is far from trivial. Realtime Oberon's contribution is a provably correct PIP implementation (6.2). This is a clear step ahead of conformance testing [53].

8.4.2 Applicability of the Priority Inheritance Protocol Implementation

The second issue beside correctness is applicability: Within interdisciplinary projects, innovation is generally not exclusively driven by computer scientists but by specialists in applications, with a potentially weak background in computer science. Therefore OS artifacts like priority inheritance should be com-

pletely hidden from application programmers.
A well-known priority inversion related disaster happened with the Mars Exploration Rover Mission, although the underlying runtime system was supporting primitives for coping with the inversion problem [21]. In that particular case, application programmers overlooked or misunderstood functionality and options offered by the Runtime System's API to cope with the inversion problem.

Realtime Oberon provides a sufficiently abstract programming model that hides priority inversion issues. This has been achieved by hiding the priority inheritance implementation behind the `EXCLUSIVE` and `AWAITCONDITION` programming language statement. The application programmer does not have to be aware of the priority inversion issues, everything is handled implicitly behind the scenes. There are systems that do not include such a seamless priority inheritance implementation: An example of an inconsistent API for solving the inversion problem is the one provided by the ThreadX [29] real-time multithreading runtime environment. It provides different building blocks for synchronization like mutual exclusion and semaphores, but only mutual exclusion is enabled to handle priority inversion by inheritance. There are no conceptual reasons for restricting priority inheritance to a subset of the implemented synchronization primitives. These kinds of implementation restrictions often confuse users rather than support application programmers.
In Real Time Java, the PIP is implemented by a particular class that an application programmer has to deal with, whereas the protocol is completely transparent to the programmer in Realtime Oberon.

8.4.3 Relevance of a correct and applicable Priority Inheritance Protocol Implementation

Last but not least, a few considerations about the relevance of a correct and applicable PIP Implementation. Relevance of the PIP is only given for (pseudo parallel) time sharing systems. On a true hardware parallel system where a physical processor is available for every thread ready to run, priority inversion will never arise. The reason for this is that a thread operating within a mutually exclusive section will never be preempted because all other threads ready to run have an available processor by definition. Propagating priorities to other threads would not have any effect since all threads ready to run will run anyway. Thus the PIP becomes redundant in that particular case.

However, as soon as there is a lack of one or more physical processors to schedule all threads ready to run, the PIP takes effect. In the best case, a thread will be blocked, at most, as if there is an unlimited number of physical processors.

However, although time sharing systems are more complex from the scheduling point of view, they offer better utilization of hardware, especially when computational load fluctuates a lot over time. Hence a correct and applicable PIP implementation will remain of importance for a wide variety of systems and applications.

8.5 Asynchronous Event Handling

Realtime Oberon provides the AWAITEVENT statement for asynchronous event handling, whereas event handling is not supported in Java on the programming language level. The Java runtime invokes delegate callbacks for event handling. In summary, the arguments in favor of including event handling in the language are:

- AWAITEVENT completely eliminates call backs like interrupt handling routines, timer routines and finalizers and thereby replaces inverse programming with direct programming. This is an important benefit regarding the construction of deadlock free systems as outlined in chapter 5.4.

- The built in AWAITEVENT statement helps to keep the kernel application programming interface lean. All traditional callback services like timer and interrupt handling services are no longer needed.

- AWAITEVENT underlines the duality of boolean conditions and events. It is a natural complement to the AWAITCONDITION statement.

Let us now focus on a particular class of asynchronous events: hardware interrupts. *"Interrupt latency is the time from the assertion of a hardware interrupt until the first instruction of the device driver's interrupt handler is executed. The [QNX micro kernel] leaves interrupts fully enabled almost all the time, so that interrupt latency is typically insignificant."* [19] This strategy of dealing with first level interrupts is very common and thus not specific the cited OS.

We have chosen a different approach with Realtime Oberon. It leaves interrupts only partially enabled almost all the time according to chapter 6.1.1. If the kernel leaves interrupts fully enabled almost all the time, then the running thread is interruptible almost all the time. This clearly harms predictability. Our approach is more sophisticated. It leaves interrupts only enabled if the corresponding second level interrupt handling thread preempts the running thread due to a higher priority. Thus non-effective interruptions and context switches are avoided.

8.6 Asynchronous Transfer of Control

The idea behind what the Real Time Specification for Java [5] calls *Asynchronous Transfer of Control* is that thread A may interrupt another thread B and thread B resumes at a predefined execution point. The idea is to let thread B react with minimum latency on external events. Asynchronous Transfer of Control is implemented with exceptions. Thread A raises an exception that thread B catches as opposed to the usual case where the same thread rises and catches an exception.

Asynchronous Transfer of Control is conceptually promising but tricky to deal with in practice. For instance, a thread is not interrupted when it is in a critical section. This voids the concept to a certain extent. Additionally, the programmer has to make sure that the thread is always in a consistent state when the thread is interruptible.

Realtime Oberon does not provide a special concept for Asynchronous Transfer of Control. A work around is to let determine the thread at specific execution points whether it should abort its operation or not. At first glance, injecting these checks into the code at specific execution points looks more costly than just catching an exception as in the Java case. But also the Java solution does not spare the programmer from carefully thinking about execution consistency. The drawback with the Realtime Oberon work around is the latency that a thread reacts to. The latency is certainly not minimal, it is given by the time in between check points. The actual impact has to be evaluated by considering the use case.

Chapter 9

Conclusion

To conclude, we summarize conceptual and technical results and propose future work.

9.1 Summary

Conceptual Results

- The scheduling model presented in chapter 4 is *invariant to all activities* in the system, especially to hardware interrupt threads. All activities obey two precise and concise rules. These two simple rules allow anticipating the application's timely behavior.

- Realtime Oberon includes a built in `AWAITEVENT` statement to overcome *inversion of control*. Up-calls like interrupt handler and finalizers are substituted by ordinary threads invoking `AWAITEVENT`. Getting rid of inversion of control helps, for instance, to build systems free of deadlocks by mutual exclusion (chapter 5.4).

- The two code samples 5.2 and 6.8 show how to employ *priority inheritance as a thread structuring concept* rather than just as a means to overcome a scheduling anomaly.

Technical Results

- Chapter 6.2 outlines a *provably correct PIP implementation*. A formal proof is a step ahead of conformance testing.
- We have implemented three data driven *sample applications* based on Realtime Oberon. These medical IT applications have been successfully field tested.

9.2 Future Work

Conceptual Improvements

- The *asynchronous control of transfer* challenge mentioned in chapter 8.6 remains unsolved. When a thread is interrupted, it is supposed to immediately stop but at the same time leave the system in a consistent state in order to resume its operation at a different execution point. This is somehow a conflicting goal. A programmer should get a tool to express when a thread is interruptible.

Technical Improvements

- Automated tools to statically check system properties are missing. Instead of checking by hand if no up-calls in the execution path are possible, an automated tool could take over that task. Such a tool would significantly simplify reasoning on the execution path in order to avoid deadlocks by mutual exclusion.

List of Figures

Listings

Bibliography

[1] *Health informatics - Standard communication protocol - Computer - assisted electrocardiography*, 2002.

[2] U. Anliker, Jamie A Ward, Paul Lukowicz, Gerhard Troster, F. Dolveck, M. Baer, F. Keita, E.B. Schenker, F. Catarsi, L. Coluccini, A. Belardinelli, D. Shklarski, M. Alon, E. Hirt, R. Schmid, and M. Vuskovic. Amon : a wearable multiparameter medical monitoring and alert system. *IEEE Transactions on Information Technology in Biomedicine*, 8(4):415–427, 2004.

[3] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.*, 7(1):5:1–5:49, December 2007.

[4] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38:285–298, January 2003.

[5] Greg Bollella, Ben Brosgol, Steve Furr, David Hardin, Peter Dibble, James Gosling, and Mark Turnbull. *The Real-Time Specification for Java*. Addison – Wesley, 2000.

[6] Roberto Brega. *A Combination of System Software Techniques Aimed at Raising the Run-Time Safety of Complex Mechatronic Applications*. PhD thesis, ETH Zürich, 2002. Diss. ETH No. 14513.

[7] Oracle corp. Java platform standard ed. 7. `http://docs.oracle.com/javase/7/docs/api/index.html`, 2012.

[8] ETH Zürich Departement für Informatik. Die welt zwischen 0 und 1, 25 jahre informatik an der eth zürich. http://www.25jahre.inf.ethz.ch/ausstellung/fact_sheets/fact_sheets/021_Der_Arzt_in_der_Westentasche.pdf, 2006.

[9] Bernhard Egger. Simple installation and windows interoperability for eth oberon. Semester project.

[10] Bernhard Egger. Development of an aos operating system for the dnard network computer. Master's thesis, Department of Computer Science, ETH Zürich, 2001.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[12] Brian Goetz and Robert Eckstein. How to handle java finalization's memory-retention issues. http://java.sun.com/developer/technicalArticles/Programming/rt_pt2/, July 2007.

[13] Ary Golberger. Non-linear dynamics for clinicians: chaos theory, fractals, and complexity at the bedside. *Lancet*, pages 1312–1314, May 1996.

[14] Jürg Gutnecht and Niklaus Wirth. *Project Oberon-The Design of an Operating System and Compiler*. Addison – Wesley, 1992.

[15] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, 1998.

[16] C. A. R. Hoare. Monitors, an operating system structuring concept. *Communications of the ACM*, 17(10), 1974.

[17] Galen Hunt. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[18] LynuxWorks Inc. Lynuxworks patented technology speeds handling of hardware events. http://www.lynuxworks.com/products/whitepapers/patentedio.php3, 2011.

[19] QNX Inc. The qnx microkernel. http://www.qnx.com/developers/docs/6.4.0/neutrino/sys_arch/kernel.html#INTERRUPTHANDLING, 2012.

[20] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *8th International Conference on Formal Engineering Methods, ICFEM, Macao, China, Proceedings, volume 4260 of LNCS*, pages 420–439. Springer, 2006.

[21] Mike Jones. What really happened on mars rover pathfinder. `http://catless.ncl.ac.uk/Risks/19.49.html#subj1`, 1998.

[22] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[23] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[24] Thomas Kaegi-Trachsel and Jürg Gutknecht. Minos - the design and implementation of an embedded real-time operating system with a perspective of fault tolerance. In *International Multiconference on Computer Science and Information Technology - IMCSIT*, pages 649–656, 2008.

[25] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley Professional, 3 edition, July 1997.

[26] Bruno Koller. Support-vektor maschine für aos bluebottle. Master's thesis, Department of Computer Science, ETH Zürich, 2007.

[27] Christian Kurmann. *Zero-Copy Strategies for Distributed CORBA Objects in Clusters of PCs*. PhD thesis, ETH Zürich, 2002. Diss. ETH No. 14950.

[28] Jané R. Laguna, P. and P. Caminal. Automatic detection of wave, boundaries in multilead ecg signals: validation with the cse database. *Comput. Biomed. Res. 27*, pages 45–60, 1994.

[29] Edward L. Lamie. *Real-Time Embedded Multithreading*. CMP Books, 2006.

[30] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[31] Ling Liu and Alexey Morozov. A process-oriented streaming system design paradigm for fpgas. In *2010 International Conference on Reconfigurable Computing and FPGAs*, 2010.

[32] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *In 26th IEEE Real-Time Systems Symposium*, 2005.

[33] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *In 26th IEEE Real-Time Systems Symposium*, 2005.

[34] MSDN Microsoft corp. Hardware specifications for windows phone. `http://msdn.microsoft.com/en-us/library/ff637514(v=VS.92).aspx`, 2011.

[35] David Monniaux. Verification of device drivers and intelligent controllers: a case study. In *EMSOFT*, pages 30–36, 2007.

[36] MSDN. Object.finalize method. `http://msdn.microsoft.com/en-us/library/system.object.finalize.aspx`, December 2010.

[37] MSDN. Task parallel library (tpl). `http://msdn.microsoft.com/en-us/library/dd460717.aspx`, March 2013.

[38] Pieter Muller. *The Active Object System, Design and Multiprocessor Implementation*. PhD thesis, ETH Zürich, 2002. Diss. ETH No. 14755.

[39] ETHZ Native Systems Group, D-INFK. A2 distribution. `http://www.bluebottle.ethz.ch/download.html`, February 2010.

[40] J. Pan and W. J. Tompkins. A real-time QRS detection algorithm. *IEEE Trans Biomed Eng*, 32(3):230–236, March 1985.

[41] David L. Parnas, John van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–647, June 1990.

[42] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. In *Proc. of ACM/SIGCOMM'94*, pages 387–396, 1994.

[43] Mike Printezis. Garbage collection and the sun java real-time system (java rts). `http://java.sun.com/developer/technicalArticles/javase/finalization/`, September 2008.

[44] Patrik Reali. *Using Oberon's Active Objects for Language Interoperability and Compilation*. PhD thesis, ETH Zürich, 2003. Diss. ETH No. 15022.

[45] Arsenii Rudich. *http://e-collection.library.ethz.ch/view/eth:5571*. PhD thesis, ETH Zürich, 2011.

[46] Miro Samek. *Practical UML Statecharts in C/C++*. Elsevier Inc., 2009.

[47] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, UK, 2001. Springer-Verlag.

[48] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[49] V. Shulgin. *Technical Documentation and Specification*. Kharkov, Ukraine, 2005.

[50] Herb Sutter. Writing a generalized concurrent queue. *Dr. Dobb's*, 2008.

[51] Herb Sutter. Writing lock-free code: A corrected queue. *Dr. Dobb's*, 2008.

[52] Dieter Zöbel and David Pollock. Priority inversion revisited. In *12th International Conference on Real-Time Systems*, pages 411–414, 2004.

[53] Dieter Zöbel, David Polock, and Andreas van Arkel. Testing for the conformance of real-time protocols implemented by operating systems. *Electr. Notes Theor. Comput. Sci.*, 133:315–332, 2005.