UNIVERSIDAD POLITÉCNICA DE VALENCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Flexible Real-Time Linux

A New Environment for Flexible Hard Real-Time Systems

Tesis doctoral

Dirigida por:
Dra. Dña. Ana García Fornes y
Dr. D. Vicente Botti Navarro

Presentada por:
Andrés Martín Terrasa Barrena

Valencia, 2000

UNIVERSIDAD POLITÉCNICA DE VALENCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Flexible Real-Time Linux

A New Environment for Flexible Hard Real-Time Systems

Andrés M. Terrasa Barrena

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor en Informática

This work has been co-supervised by
Dr. Ana García Fornes and Dr. Vicente Botti Navarro

Valencia, 2000

A Ana, por su eterna (im)paciencia,
y en memoria de nuestro amigo Lucas

I have never begun a novel with more misgiving.
If I call it a novel it is only because I don't know
what else to call it. I have little story to tell and
I end neither with a death nor a marriage.

W. SOMERSET MAUGHAM:
The razor's edge

```
/*
 * Buddy system.  Hairy.  You really aren't
 * expected to understand this
 *
 * Hint:  -mask = 1+~mask
 */
```

LINUS TORVALDS:
`/usr/src/linux/mm/page_alloc.c`

Abstract

This thesis proposes a new general framework for building *flexible hard real-time systems*, that is, systems featuring both hard timing constraints and flexible behaviour. The proposed framework is capable of integrating tasks with different criticality levels as well as combining different scheduling paradigms into the same system. As a result, the framework completely guarantees the timing requirements of hard tasks, while also allowing for an adaptive and intelligent scheduling of non-hard tasks. The framework is divided into a computational model, a software architecture and a set of services. The computational model proposes building a flexible real-time application as a set of tasks, with each task being structured as a sequence of mandatory and optional components. The software architecture proposes separating the execution of task components into two scheduling levels, one level for scheduling the task mandatory components by means of a hard real-time policy and another level for scheduling the task optional components by means of a sort of utility-based policy. The set of services actually includes both a facility for communicating mandatory and optional components inside an application and a group of mechanisms for explicitly detecting and handling run-time timing exceptions.

The proposed framework is also shown to be realizable in practice. In particular, this thesis presents the design and implementation of a real run-time support system which supports all the features proposed by the framework. This run-time system, called Flexible Real-Time Linux (FRTL), has been developed by enhancing the original capabilities of the Real-Time Linux (RT-Linux) operating system.

Finally, an exact timing characterization of the FRTL run-time system and actual measurements of its overhead are also provided. The timing characterization allows for the development of a complete feasibility analysis of the entire system, which can be then used for verifying the timing constraints of any FRTL-based application. The overhead measurements show that the FRTL has been designed and implemented very efficiently. Overall, FRTL is shown to be both predictable and efficient, which actually proves that it can be useful in building real flexible real-time systems

The work presented in this thesis has been developed in the context of the following re-

search projects, which have been funded by the Spanish government: *"Entornos de sistemas basados en el conocimiento de tiempo real"* (TAP94-0511-C02-01), *"ARTIS: herramienta para el desarrollo de sistemas inteligentes en tiempo real aplicados al control de robots móviles"* (TAP97-1164-C03-01), *"ARTIS: herramienta para el desarrollo de sistemas inteligentes en tiempo real estricto"* (TAP98-0333-C03-01) and *"Soporte de ejecución para sistemas empotrados distribuidos"* (TIC99-1043-C03-02). Another related research project, which is funded in this case by the Valencian Government, is called *"Desarrollo de un entorno de ejecución para la realización de sistemas de tiempo real estricto"* (GV98-14-76).

The main contributions of this thesis have been published in the following magazines and conferences: an implementation of an ARTIS prototype was presented in [Gar96b, Gar97a]. This prototype, which was the predecessor of FRTL, was implemented in Solaris by using POSIX facilities. Both papers presented a complete timing characterization of the prototype's kernel. An early implementation of this prototype in RT-Linux was presented in [Ter98]. In [Esp98], the problem of sharing memory predictably in the prototype was addressed. Finally, the actual framework proposed in this thesis and the corresponding FRTL run-time system have recently been published in [Ter99, Terr00a, Terr00b].

Acknowledgments

First of all, I would like to thank my two co-advisors, Dr. Ana García Fornes and Dr. Vicente Botti Navarro, for their constant dedication and their efforts in helping to make this work into a doctoral thesis. Without them, this thesis would never have been finished.

I would also like to thank the rest of the colleagues in my research group, *Grupo de Tecnología Informática/Inteligencia Artificial* (or simply GTI/IA) for their support. The never-ending discussions that we normally have in the group meetings (week after week) have undoubtedly brought indispensable ideas related to this work. Specially, I would like to thank Carlos Carrascosa for our philosophical discussions about ARTIS and Ignacio Pachés for his help in debugging and measuring the kernel source code. I would also like to give special mention to Agustín Espinosa, a brilliant fellow who is always so fast in finding weak points in my work as in helping me to solve them. I want to thank them all for making my job a daily enjoyment (this includes our particular way of testing the network capabilities... well, they know perfectly well what I'm talking about).

Part of the work of this thesis was developed while I was visiting the New Mexico Institute of Mining and Technology (NMT), which is located in a tiny town called Socorro, in New Mexico (USA). In my one-year-long stay there (from February, 1999 to February, 2000), I had the privilege of working with Dr. Victor Yodaiken. Besides his technical support and suggestions, he taught me that good research should be a synonym for *useful* research. I'll try not to forget that. I also made many friends in Socorro. With their help, I managed to feel at home within the beautiful (and sometimes too quiet) landscapes of New Mexico. Thanks to Paulo Cortés, Pablo Altamirano, José Guilberto, Axel Bernal, Carlos López Mariscal, Pablo Velázquez, Juan Muñoz, Mike Martínez, Niruj Mohan, Amy Bartlet, Sukesh Ganda, Justin Jayne, Jen Rinard, Lee Paprocki, Bill Davenport, and many, many others. Regardless of whether I talked to them in Spanish or in English, I shared very good times with all of them. I sometimes wish I were in the *Cap* again having a beer with all of you.

I also had the opportunity of discussing some of the ideas of this thesis with people who do not belong to my university or the NMT. This is the case of Guillem Bernat, who helped me in some parts of the thesis. I would like to thank him for his comments and for sharing

his thesis' latex style with me. He made me promise to say that this style was designed by his wife.

Last, but by no means least, I want to thank my family (Ana, my parents Andrés and Micaela, my sister Silvia and my brother-in-law Vicent) for their constant support. It seems that writing a thesis inevitably makes you angrier than usual, and they have suffered the worst part of it with great patience.

Contents

Abstract i

Acknowledgements iii

Contents v

List of Figures ix

List of Tables xi

Glossary of Acronyms xiii

Nomenclature xv

1 Introduction 1

1.1 Purpose 1

1.2 Objectives 3

1.3 Organization 3

2 An Overview of Flexible Hard Real-Time Systems 5

2.1 Real-Time Systems 5

2.2 Flexible Real-Time Systems 7

2.3 Task Models for Imprecise Computations 10

2.3.1 The Milestone Method 10

2.3.2 Sieve Functions 10

2.3.3 Multiple Versions 11

2.3.4 The Prologue-Optional-Epilogue Model 11

2.4 An Overview of Fixed Priority Preemptive Scheduling 12

2.5 ARTIS 15

2.6 Summary and Discussion 16

3 An Overview of Real-Time Operating Systems **19**

3.1 Introduction 20
3.2 The Realtime Extensions of POSIX 21
3.2.1 POSIX facilities 21
3.2.2 POSIX profiles 22
3.2.3 Discussion 23
3.3 Real-Time Kernels underneath GPOSs 24
3.3.1 Slotted Priorities 24
3.3.2 Real-Time Linux 25
3.3.3 Systems based on Windows NT 27
3.4 Summary and Discussion 27

4 A New Framework for FRTS **29**

4.1 Introduction 30
4.2 The Task Model 31
4.2.1 Tasks 31
4.2.2 Components 31
4.2.3 Synchronization 35
4.2.4 Dynamic Components 35
4.2.5 Set of Constraints Imposed over the Model 36
4.3 Formalizing the Task Model 36
4.3.1 Tasks and Components 36
4.3.2 Dynamic Components 38
4.3.3 Model Constraints and Properties 39
4.4 The Software Architecture 40
4.4.1 The Real-Time Level 40
4.4.2 The Non-Real-Time Level 52
4.5 The FRTL Run-Time System 55
4.6 The Set of Interface Functions 56
4.7 Summary and Contributions 58

5 Synchronization Facilities **61**

5.1 Introduction 62
5.2 Related work 64
5.2.1 Priority Ceiling Protocol 64
5.2.2 Ceiling Semaphore Protocol 65
5.2.3 Slack Stealing Algorithm and the PCP 67

5.2.4 Problems Detected in this Work 68
5.3 Integration of the Priority Ceiling Protocol 69
5.3.1 The PCP at the Real-Time Level 69
5.3.2 The PCP and the Slack-Stealing Algorithm 70
5.3.3 The PCP at the Non-Real-Time Level 70
5.4 Integration of the Ceiling Semaphore Protocol 71
5.4.1 The CSP at the Real-Time Level 71
5.4.2 The CSP and the Slack-Stealing Algorithm 72
5.4.3 The CSP at the Non-Real-Time Level 73
5.5 Interface Functions 75
5.6 Evaluation of the Two Protocols 76
5.6.1 High-Level Evaluation 76
5.6.2 Run-Time Evaluation 78
5.7 Summary and Contributions 79

6 Timing Exception Support 81
6.1 Introduction 82
6.2 Previous Work 84
6.3 Design Principles 85
6.4 Support for *wcet* Exceptions 86
6.4.1 Detection of *wcet* Exceptions 87
6.4.2 Handling of *wcet* Exceptions 91
6.4.3 Introducing User-Defined Handlers into the Task Model 92
6.5 Support for Deadline Exceptions 93
6.6 Interface Functions 94
6.7 Summary and Contributions 95

7 The FRTL Run-Time System 97
7.1 Overall Description of FRTL 98
7.1.1 FRTL's Real-Time Level 98
7.1.2 FRTL's Linux Level 102
7.1.3 The Storage 104
7.2 The Real-Time Level 105
7.2.1 Interrupt-Driven Kernel 105
7.2.2 Common Entry and Exit Points 106
7.2.3 The Scheduler's Layered Structure 107
7.2.4 Taxonomy of System Calls 111
7.2.5 Implementation Details 112
7.3 The Linux Level 116

7.3.1 Internal Design of the Second-Level Scheduler 116
7.3.2 Synchronization Between the Two Schedulers 118
7.3.3 Implementation Details 118
7.4 System Tools 121
7.4.1 Collection of Run-Time Data 121
7.4.2 Off-line Analysis of the Collected Data 123
7.5 Summary and Contributions 125

8 Complete Feasibility Analysis **127**
8.1 Introduction 128
8.2 Previous Work 129
8.3 Theoretical Test Adapted for the Framework 130
8.4 Developing a Complete Test 132
8.4.1 Kernel Design Issues 133
8.4.2 Inclusion of Interrupt Handling 133
8.4.3 Inclusion of Implicit System Calls 134
8.4.4 Inclusion of Explicit System Calls 135
8.4.5 Inclusion of Timing Exception Support 136
8.4.6 Final Equation 137
8.5 Overhead Measurements 139
8.5.1 Experiment Design 140
8.5.2 Measurements of the Kernel Constants 142
8.6 Summary and Contributions 145

9 Conclusions and Future Work **153**
9.1 Conclusions and Contributions 153
9.2 Future Lines of Work 156

Bibliography **159**

List of Figures

4.1 Example of a task structure. 33
4.2 The software architecture. 41
4.3 Schedule function of the first-level scheduler. 43
4.4 Example of slack scheduling 49
4.5 Main loop of the second-level scheduler. 53
4.6 Set of interface functions at the real-time level. 56
4.7 Set of interface functions at the non-real-time (Linux) level. 57

5.1 CSP anomaly at the non-real-time level. 74
5.2 Set of synchronization functions. 75

6.1 Example of the compensation mechanism. 89
6.2 Set of exception-handling interface functions. 95

7.1 The Flexible Real-Time Linux system. 99
7.2 The internal layered design of the FRTL's real-time level. 107
7.3 Sequence of calls when processing a system event 110
7.4 The `rt_task_runner` function. 113
7.5 The `rt_timer_handler` function. 115

8.1 Calculation of T_k^h 132

List of Tables

4.1 Types of components in the task model. 34

5.1 Measured values of the cost of lock and unlock operations. 78

7.1 Trace information collected by the first-level scheduler 124

8.1 Kernel constants introduced in the feasibility test. 140
8.2 Description of the hardware platforms. 141
8.3 Attributes of tasks included in test applications. 141
8.4 Internal structure of tasks in the test application (tasks 1 to 8) 143
8.5 Internal structure of tasks in the test application (tasks 9 to 12) 144
8.6 Overhead measurements for a 200 Mhz. Pentium processor 147
8.7 Overhead measurements for a 233 Mhz. Pentium II processor 148
8.8 Overhead measurements for a 450 Mhz. Pentium II processor 149
8.9 Overhead measurements for a 500 Mhz. Pentium III processor 150
8.10 Overhead measurements for a 600 Mhz. Pentium III processor 151

Glossary of Acronyms

ARTIS	An Architecture for Real-Time Intelligent Systems.
CSP	Ceiling Semaphore Protocol.
DASS	Dynamic Approximate Slack Stealing algorithm.
DM	Deadline Monotonic.
DSS	Dynamic Slack Stealing algorithm.
EDF	Earliest Deadline First.
FRTL	Flexible Real-Time Linux.
FRTS	Flexible Hard Real-Time Systems.
GPOS	General-Purpose Operating System.
PCP	Priority Ceiling Protocol.
PIP	Basic Priority Inheritance Protocol.
POSIX	Portable Operating System Interface.
RM	Rate Monotonic.
RTAI	Real-Time Artificial Intelligence.
RTOS	Real-Time Operating System.
RTS	Real-Time System.
wcet	Worst-case execution time.

Nomenclature

Definition of the Task Model

$\mathcal{A}$	Flexible hard real-time application, consisting of a sequence of tasks $\{\tau_1, \tau_2, \ldots, \tau_i, \ldots, \tau_N\}$ plus a set of dynamic components, $\{\gamma_1^d, \gamma_2^d, \ldots, \gamma_j^d, \ldots, \gamma_M^d\}$.
τ_i	Application task i, $\tau_i = (T_i,\, D_i,\, O_i,\, C_i,\, B_i,\, H_i,\, M_i,\, \Gamma_i)$.
γ_j^d	Application dynamic component, $\gamma_j^d = (Y_j^d, D_j^d, M_j^d,\, \Phi_j^d)$.
T_i	Period of task τ_i.
D_i	Deadline of task τ_i.
O_i	Initial offset of task τ_i.
C_i	Worst-case execution time of task τ_i.
B_i	Worst-case blocking time of task τ_i.
H_i	Handler of task τ_i, $H_i = (Y_i^h,\, C_i^h,\, P_i^h)$
M_i	Number of components of task τ_i.
Γ_i	Sequence of components of task τ_i, $\Gamma_i = \{\gamma_{i1}, \gamma_{i2}, \ldots, \gamma_{ij}, \ldots, \gamma_{iM_i}\}$.
γ_{ij}	jth component of task τ_i, $\gamma_{ij} = (Y_{ij},\, S_{ij},\, M_{ij},\, \Phi_{ij})$.
Y_i^h	Type of task τ_i's handler.
C_i^h	Worst-case execution time of task τ_i's handler.
P_i^h	Priority of task τ_i's handler.
Y_{ij}	Type of component γ_{ij}.
S_{ij}	Slack fraction of component γ_{ij}.
M_{ij}	Number of versions of component γ_{ij}.
Φ_{ij}	Set of versions of component γ_{ij}, $\Phi_{ij} = \{c_{ij1}, c_{ij2}, \ldots, c_{ijk}, \ldots, c_{ijM_{ij}}\}$.
c_{ijk}	Worst-case execution time of the kth version of component γ_{ij}.

Y_j^d	Type of dynamic component γ_j^d.
D_j^d	Deadline of dynamic component γ_j^d.
M_j^d	Number of versions of dynamic component γ_j^d.
Φ_j^d	Set of versions of dynamic component γ_j^d.

Definition of the Slack Stealing Algorithm

γ_{ij}^*	List of contiguous optional components headed by component γ_{ij}.
S_{ij}^*	Combined slack fraction for list of optional components γ_{ij}^*.
ES_{ij}^*	Effective slack fraction for list of optional components γ_{ij}^*.
$AS_{ij}^*(t)$	Available slack for list of optional components γ_{ij}^* at time t.
$x_k(t)$	Time of the next release of task τ_k after time t.
$SI_{ij}^*(t)$	Slack interval to be programmed for list of optional components γ_{ij}^* at time t.
$d_{ij}^*(t)$	Absolute deadline for list of optional components γ_{ij}^* at time t.
$I_k(t_1, t_2)$	Exact interference caused by task k in the interval $[t_1, t_2]$.

Definition of the Feasibility Analysis

R_i	Worst-case response time of task τ_i.
$man(i)$	Set of mandatory components of task τ_i.
$hand(i)$	Set of tasks having a mandatory handler with a priority higher than or equal to i.
T_k^h	Minimum inter-arrival time of task k's handler.
C_i^{real}	Real worst-case execution time of task τ_i, including the kernel overhead directly related to it.
$C_{release}$	Kernel cost of releasing an application task.
C_{gain}	Kernel cost of processing the gain time after executing a task mandatory component.
C_{opt}	Kernel cost of releasing a task optional component.
C_{end}	Kernel cost of ending a task release.
C_{lock}	Kernel cost of processing a mutex lock request.

C_{unlock}	Kernel cost of processing a mutex unlock request.
$C_{dynamic}$	Kernel cost of processing a dynamic mandatory component release request.
C_{except}	Kernel cost of processing a timer interrupt which signals a task timing exception.
C_{end_hand}	Kernel cost of ending a mandatory handler release.
C_{max}	Longest non-interruptible section of kernel code.

1

Introduction

1.1 Purpose

A *real-time system* (RTS) can be defined as a computer system on which the correctness of its response depends on both the logical correctness of the results and the moments at which these results are generated. In other words, results have to be provided within well-defined intervals of time in order to be useful. In real-time systems, this requirement is typically expressed as a set of *timing constraints* that the system has to meet at run time. *Hard real-time systems* form a subset of RTSs on which a single failure in one timing constraint may lead to a catastrophe. For example, a self-guided robot failing to detect an obstacle may crash into this obstacle. The strict necessity of avoiding any timing failure has focussed traditional hard real-time systems on guaranteeing that their timing constraints are always met. In order to fulfill this goal, simplistic theoretical models have been used. These models are almost exclusively driven by timing characteristics, such as the *deadline* of each application task, and require extensive *a priori* information of the problem or environment being solved. Thus, traditional real-time applications, which are built according to these models, can only deal with environments which are stable, completely specified and which are not very complex.

Many research efforts have emerged in the last few years in order to overcome these limitations. These efforts share the common goal of adding *flexibility* and *intelligence* to hard real-time systems. Systems of this new generation are intended to solve complex problems, to deal with dynamic environments, to work with incomplete or uncertain data, etc. As a result of these efforts, the concept of *flexible real-time system* has progressively become more rele-

vant. However, this concept has not yet been accurately defined: it seems to be applied to a large and heterogeneous group of methods, architectures, techniques, computational models, scheduling policies, etc., which try to incorporate intelligent, adaptive and flexible behaviour to systems with real-time constraints. This is specially difficult in hard real-time systems, since any added feature must completely preserve strict timing constraints. Systems of this kind, here referred to as *Flexible Hard Real-Time Systems* (FRTS), are the topic of this thesis.

The first main aim of this work is to try to provide a general framework for building FRTS. As a minimum, such a framework has to provide two main features. On the one hand, the framework has to have the ability to combine tasks with different criticality levels into the same application. Hard tasks are required to execute time-critical actions and thus they normally implement simple, fast and predictable algorithms. Less critical tasks can be used for implementing more complex and time-consuming algorithms, solving less critical aspects of the problem and improving the response achieved by hard tasks. On the other hand, the framework also has to provide a scheduling policy which is more sophisticated than traditional policies for hard real-time systems. This scheduling policy must completely ensure the execution of hard tasks while also scheduling non-hard tasks in the best possible manner. The expression "best possible manner" implies that the scheduling process itself can help in improving the system response quality. In fact, the scheduling of non-hard tasks is one of the key aspects in incorporating flexibility and intelligence to FRTS. This scheduling process could not only maximize some quality function of task execution (as it proposes the *best-effort paradigm* [Loc86]) but it could also incorporate specific knowledge of the problem being solved, adapt the system to the current status of the environment, dynamically choose among different scheduling strategies, etc.

The state-of-the-art approaches on FRTS have made contributions to either the system's computational model or to the run-time scheduling policy. However, both aspects are not normally considered at the same time. This thesis proposes a framework that tries precisely to incorporate flexibility from both perspectives. First, the framework incorporates a computational model in which tasks are made up of components, which can be either mandatory or optional, with the mandatory components being guaranteed to execute. Second, the framework proposes the combination of two hierarchically related scheduling policies, one for scheduling the mandatory components and the other for scheduling the optional components. The framework establishes the policy for the mandatory components and the relationships between both policies, but does not constrain the optional component's policy. This policy can be selected, adapted or even completely implemented by the application designer, depending on the requirements of the application.

The second aim of this thesis is to implement the framework for FRTS as a real run-time system. In this sense, the thesis presents the design and implementation of Flexible Real-Time Linux (FRTL). FRTL is a run-time support system (or kernel) which has been developed to support all the framework's features. FRTL has been implemented by enhancing the original functionality of the Real-Time Linux (RT-Linux) version 1 operating system [Yod99]. RT-Linux leads a new trend in real-time operating systems in which a General-Purpose Operating

System (GPOS) is provided with hard real-time capabilities by adding a small and predictable real-time layer between the hardware and the GPOS kernel. The thesis also presents a complete feasibility test of FRTL as a very important part in the realization of the framework. This complete test incorporates all the relevant features of the framework's task model and the particular run-time behaviour of FRTL. The availability of such a complete test is necessary if the kernel is to be useful. The designer can only prove the system's feasibility by studying the schedulability of the entire system (i.e., the application tasks plus the kernel).

This work has been developed as a result of several years of research on Real-Time Artificial Intelligence (RTAI) by the research group GTI/IA, or *Grupo de Tecnología Informática/Inteligencia Artificial* (Group of Computer Science Technology/Artificial Intelligence), of the Technical University of Valencia. In particular, this work is directly related to the ARTIS architecture [Gar96a]. In many senses, this architecture is the theoretical foundation of the framework proposed in this work. In fact, this thesis can be seen as both an evolution and a realization of ARTIS.

1.2 Objectives

The overall goal of this thesis is to analyze and identify the requirements of FRTS and to provide both a framework and a corresponding run-time system which are appropriate for systems of this type. This global goal can be divided into the following specific objectives, which actually constitute the sequential steps of the work presented in this thesis:

- To identify the key features which are normally required by FRTS. These features lead to the definition of the requirements that a framework for FRTS should provide.

- To propose a new framework for building FRTS according to the requirements previously identified. The framework has to include a computational model, a software organization or architecture and a set of services which can support these requirements.

- To design and implement an actual run-time support system according to the proposed framework, to be provided to the application developer for implementing FRTS. The run-time system is required to be predictable and efficient.

- To develop a generic, complete schedulability analysis of the run-time system. The test has to include all the framework's characteristics and the particular timing behaviour of the run-time system. By using this complete test, the application developer can accurately study the timing feasibility of the entire system.

1.3 Organization

This thesis is divided into nine chapters. The first chapter presents the aims, objectives and organization of the thesis.

Chapters 2 and 3 present the state-of-the-art results of FRTS and Real-Time Operating Systems (RTOSs), respectively. In the final conclusions of these chapters, the features that FRTS should include are presented from the viewpoint of both the framework and the run-time system.

Chapter 4 introduces a new framework for building FRTS. This framework first proposes and formalizes a task model on which tasks are defined as a sequence of mandatory and optional components. This chapter also proposes a software architecture in which the application components are separated into two scheduling levels. The chapter details how such an architecture could be designed. Finally, this chapter briefly introduces the FRTL run-time system and presents the set of interface functions (related to the task model) which are available for building an application in FRTL.

Chapter 5 introduces two synchronization protocols which have been adapted for the framework in order to allow mandatory and optional components to safely share memory. The chapter introduces the original definition of each protocol and shows how each can be adapted to the framework. The end of the chapter is devoted to comparing both protocols in order to determine which one is more appropriate for the framework.

Chapter 6 characterizes the possible types of timing exceptions that may occur at run time and discusses the situations in which they may occur. Then, appropriate mechanisms for supporting these situations are proposed. These mechanisms have been developed with the goal of being predictable as well as providing a highly flexible support to the designer. In particular, the detection mechanism has been integrated within the run-time system in order to maximize the effectiveness of the real-time scheduling policy. The handling mechanism allows the designer to implement specific timing handlers for each application task. The formal task model introduced in Chapter 4 is extended in order to include these user-defined handlers.

Chapter 7 describes the design and implementation of the FRTL run-time support system in detail. The chapter discusses the most relevant features of this system, specially emphasizing the aspects which affect its run-time timing behaviour.

Chapter 8 introduces a system's complete feasibility test, which takes into account the characteristics of both the theoretical framework and the FRTL run-time system. This chapter presents a novel technique for analyzing the impact of running the kernel with the interrupts disabled on the schedulability conditions of tasks.

Finally, Chapter 9 summarizes the contributions and conclusions of this thesis and introduces some future lines of work.

2

An Overview of Flexible Hard Real-Time Systems

2.1 Real-Time Systems

A *real-time system* is commonly defined as a computer system in which the correctness of the system depends not only on the logical result of computation (logical correctness), but also on the times at which results are produced (timing correctness) [Ram94]. This second type of correctness is normally expressed as a set of *timing constraints* that the system has to meet at run time. According to these constraints, real-time systems may be broadly classified into two categories: *hard* and *soft*. Hard real-time systems require all their constraints to be met under any circumstances, otherwise catastrophic results may occur. Many examples of hard systems can be found within computer systems where a timing failure can cause an intolerable cost (in terms of human lives, equipment damage or economic loss) such as avionics, robotic systems, nuclear or chemical plants, etc. Soft real-time systems allow some of their constraints to be occasionally lost, producing a degradation of the system response. A good example of a soft system is a multimedia application whose video/audio output loses quality whenever the system is not able to process all the input frames on time. This thesis is within the context of hard real-time systems.

Hard real-time systems are normally identified as computing systems which interact with (or control) a process or *environment* in the real world. In this sense, a real-time system con-

tinuously executes a loop in which it inputs data from the environment (perceives), computes solutions based on the entry data (thinks), and outputs the solutions to the environment (acts), potentially changing the environment status. Then, the system repeats this cycle again. All these actions have to be performed in a timely fashion, with specific timing restrictions imposed by the environment. The multiplicity of processes present in real-world environments and the inherent parallelism of these processes usually lead to designing and implementing a hard real-time system as a set of cooperative, concurrent tasks. In this kind of design, each application task solves a particular subset of the problem and it is assigned a specific timing constraint. This constraint is typically expressed as a *deadline* which the task has to meet each time it is executed. Hence, the general goal of timing correctness in hard real-time systems is achieved in practice when the application tasks are scheduled and executed in such a way that they always meet their deadlines. For this reason, most of the research efforts in the field of hard real-time systems are centered on developing *scheduling paradigms* which are appropriate for achieving this execution goal. Normally, all other topics related to building these systems (such as requirement specifications, design methodologies, programming languages, operating system facilities, etc.) are geared towards the specific scheduling paradigm used by the system.

Scheduling paradigms for hard real-time systems are focussed on guaranteeing the system timing constraints *a priori*. In other words, a hard real-time system must be known to keep its timing constraints *before* actually being executed. This guarantee is normally expressed in terms of *feasibility*. Feasibility corresponds to the knowledge of the system being able to meet its timing requirements. Feasibility, in turn, is based on *predictability*, which is defined as the complete knowledge of the specific actions that the system will take under any given run-time situation. In order to be able to provide such predictable behaviour, scheduling paradigms impose several restrictions on the system. These restrictions are typically expressed in three forms:

a) The *computational model*, also known as the task model, is the set of characteristics and constraints imposed on the application tasks.

b) The *pre-execution configuration* is normally related to certain dependencies which are statically imposed on the tasks. The term *statically* means that the dependencies do not change at run time.

c) The *run-time scheduling policy*, which establishes the actual criteria used by the run-time system for determining which task of all the runnable tasks must be executed.

The task model enforces strict rules for implementing the application tasks, such as "tasks have to be periodic or else they must have a minimum inter-arrival rate", "tasks must have a known maximum execution time", "task cannot voluntarily suspend themselves", etc. The task model also requires that several *timing attributes* about tasks be specified at design time, such as the period, deadline, worst-case execution time, initial offset, etc. On the other hand, the pre-execution configuration and the scheduling policy vary in importance depending on

the particular paradigm the system is using. For example, in the static table-driven paradigm (which includes the so-called *cyclic executive* scheduling), the pre-execution configuration is the most important activity. This activity generates an explicit schedule or calendar which determines which task has to be executed at any particular instant. In this paradigm, the run-time policy just follows this static calendar. In the fixed priority preemptive paradigm, the pre-execution configuration assigns a fixed priority to each task; at run time, the scheduling policy dynamically assigns the processor to the runnable task with the highest priority. Finally, in the dynamic priority preemptive paradigm, the pre-execution configuration is nonexistent and the run-time policy selects the ready task with the earliest absolute deadline for execution.

Each scheduling paradigm has its own method for proving the system's feasibility. For example, in systems using the static table-driven scheme, feasibility is achieved by construction. Conversely, priority-driven paradigms (with either fixed or dynamic priorities) require an explicit *feasibility analysis*. This analysis is based on a test that mathematically checks whether all tasks can meet their deadlines. This test is expressed by equations reflecting both the timing attributes of tasks and the particular run-time behaviour imposed by the scheduling policy.

To sum up, the general goal of guaranteeing timing correctness a priori imposes two main requirements when building a hard real-time system. The first requirement is to have a complete knowledge of the environment in order for the application to be able to react to environmental changes in a timely fashion. The second requirement is to enforce several restrictions on the design of the application tasks and on the run-time system. Restrictions on the task model includes, for example, knowing the maximum computation and resource requirements of every application task in advance. Restrictions on the run-time system basically imply always scheduling tasks according to the same, fixed criteria. As a result, hard real-time systems normally implement simple applications dealing with well-defined, predictable environments.

2.2 Flexible Real-Time Systems

As discussed in the previous section, traditional hard real-time systems have typically featured restrictive task models and rigidly defined scheduling policies, in order to achieve predictability. In this context, some research efforts have taken place with the general aim of adding flexibility to hard real-time systems. *Flexible Hard Real-Time System* (FRTS) is the term used in this thesis to denote a real-time system featuring both hard guarantees and a flexible run-time behaviour. This section reviews the main contributions on this topic.

The first issue to be noted here is that there is not a clear, commonly accepted definition of what *flexibility* is in the context of real-time systems. Nevertheless, there seems to be a general agreement on that flexibility means a real-time system should be able to do some of the following:

- To improve the system utilization, making the best use of the processor's spare capacity in order to run algorithms that enhance the system's response quality.

- To deal with complex, highly dynamic, unpredictable environments, effectively adapting the system to the different environment situations.
- To work in an uncertain or hostile environment. In environments of this kind, the information available may be incomplete or inaccurate.
- To incorporate unbounded algorithms. These algorithms cannot be directly introduced into a hard real-time system because they do not have a known maximum computation time or because this time is too pessimistic[1].
- To incorporate intelligent scheduling, which takes into account not only the timing attributes of tasks, but also some quality aspects and the environment situation.
- To deal with faults. A drastic change in the environment status may create a transient overload in the system or force a task to overrun its maximum computation time because of an unexpectedly large amount of input data or other factors.

Different techniques have been proposed in order to incorporate these new features, normally only addressing one of them. These techniques have been developed by two different research communities, the real-time community and the artificial intelligence community[2]. Research efforts from the real-time community are mainly focussed on incrementing the flexibility of a hard real-time system; these efforts include the Imprecise Computation model and many related research works [Liu91], the best-effort scheduling paradigm [Loc86], the dynamic planning-based scheduling paradigm [Sta98], etc. Results in RTAI are, in general, oriented towards achieving a complex, intelligent system which offers some degree of real-time response, such as the design-to-time scheduling [Gar93], the *satisficing cycle* scheduling [Hay90], etc. Overall, these different efforts may be broadly categorized in two groups, although some solutions actually make use of elements from both groups:

a) Efforts to enhance the *computational model*. The typical task model for a hard real-time system is rather restrictive. In particular, every hard task is supposed to have a known maximum computation requirement named *worst-case execution time* (*wcet*). This usually makes these tasks implement simple algorithms. In order to overcome this restriction, two main results have been proposed: the concepts of *optional component* [Aud96] and *anytime algorithm* [Bod94]. Optional components are software components implementing complex, typically unbounded algorithms, which are not strictly required to execute, but that may be run if sufficient spare capacity is detected at run time. This concept has been consolidated from earlier results such as the Imprecise Computation models [Liu91] or the Approximate Processing paradigm [Les88]. Anytime algorithms are iterative refinement algorithms that can be interrupted and asked to provide a result at any time. This is useful when tasks have real-time constraints, since any task can

[1] This means that the worst-case computation time for the algorithm is much larger than its average computation time, making the feasibility test reject a task that can be successfully scheduled in most cases.

[2] There is an actual trend called Real-Time Artificial Intelligence (RTAI) [Mus95].

be potentially allowed to run to its deadline and then be interrupted, providing the best possible result at the last possible moment. However, the main drawback of anytime algorithms is that a problem cannot always be solved by means of such algorithms.

b) Efforts to develop new algorithms for *flexible scheduling*. Traditional real-time scheduling only uses quantitative (timing) attributes of tasks in order to obtain timeliness. Traditional scheduling approaches in artificial intelligence try to achieve the best quality solution, normally without any real-time restriction, by typically using search algorithms and heuristics. Efforts in this group are oriented towards scheduling techniques which are able to include quality features into the scheduling of tasks while letting these tasks meet certain real-time constraints. Results are very heterogeneous, but may be categorized in the following groups: first, algorithms specially developed for achieving this twofold goal, such as best-effort scheduling [Loc86], planning-based scheduling [Sta98], flexible computation [Hor91], deliberation scheduling [Bod94], compilation of anytime algorithms [Zil95, Zil96], design-to-time scheduling [Gar93] and the satisficing cycle scheduling inside the AIS architecture [Hay90, Hay95]. These scheduling algorithms cannot typically provide a priori hard real-time guarantees on tasks not using anytime algorithms. The second approach corresponds to *combining* a real-time and a non-real-time scheduling algorithm, each of which works at a different level and deals with different parameters. Examples of this group are the scheduling approach of architectures such as ARTIS [Gar96a] or CIRCA [Mus93]. Finally, the third approach corresponds to *mixing* different scheduling paradigms into a single-level framework, as in [Aud96], where fixed priority preemptive scheduling is used for hard tasks and some utility-based criteria are used for scheduling optional tasks. In general, results in both the second and third approaches are able to provide hard real-time guarantees to all the critical tasks of the application.

Overall, this classification groups very different results which, in most cases, only deal with part of the problem, leading either to soft real-time systems or to hard systems without much flexibility. The rest of the chapter discusses some of the results which are directly related to the proposals of this thesis in more detail. In particular, the following section reviews the computational models developed for the Imprecise Computations paradigm. Section 2.4 discusses the main contributions in the field of single-processor fixed priority preemptive scheduling, since this is the paradigm proposed in this thesis for achieving hard real-time guarantees. Section 2.5 presents a summary of the ARTIS architecture, as it is the theoretical foundation of the framework proposed in this work. Finally, Section 2.6 presents a summary of the actual characteristics that we have identified as necessary for any framework intended to support FRTS.

2.3 Task Models for Imprecise Computations

This section first reviews the three original computational models developed by Liu et al. under the generic name of *algorithms for Imprecise Computations* [Liu91]. These models are the *Milestone Method*, the *Sieve Functions* and the *Multiple Versions*. The section then presents a more generic model introduced by Audsley et al. [Aud96], which subsumes these three models. A similar model has also been developed within the context of the ARTIS architecture [Gar96a], which is presented in Section 2.5.

2.3.1 The Milestone Method

This model normally applies to *monotone tasks*. A task is said to be monotone if the quality of its response does not decrease as the task executes. The Milestone Method is typically applied to monotone tasks which can produce (imprecise) results before completely finishing (i.e., they can be aborted at any time and still produce a consistent result). However, tasks of this type usually need some initial time before which they can produce valid results or else they are required to provide a minimum quality solution. As a result, tasks in this model are broken down into two sequential parts: a *mandatory part* and an *optional part*. The mandatory part executes first, obtaining a minimum quality solution in a predictable time. The optional part runs afterwards, improving this solution as it executes.

In the Milestone Method, the feasibility test must only take into consideration the mandatory part of each task, verifying that these parts will always execute. At run time, the more execution time the system can offer to the optional part of a task, the higher the solution quality this task will provide, at least theoretically.

This model is useful when application tasks can be implemented as monotone algorithms, which is the case of numerous algorithms for numerical computation, statistical approximations or heuristic search. However, a problem cannot always be solved by using algorithms of this kind.

2.3.2 Sieve Functions

Under this approach, a task is broken down into a sequence of computation units or *steps*, in such a way that the execution of each one refines (*sieves*) the solution provided by the previous unit. A task designed within this model normally allows some of the units to be skipped while still producing valid results. Thus, the units belonging to a task which are required to be executed are considered as *mandatory* and the rest are considered as *optional*. Mandatory parts must have known worst-case execution times in order to be considered in the corresponding feasibility test, which guarantees their execution. Optional parts will only execute if possible, and therefore they may implement unbounded algorithms.

This model is appropriate only when the solution of a problem can be produced in terms of successive refinements. For this reason, just as for the Milestone Method, the model can only be applied sometimes.

2.3.3 Multiple Versions

In the Multiple Versions model, a task always consists of two *versions*: a *primary* version and an *alternate* version. The primary version produces the highest quality solution of the task, but it normally uses an unbounded algorithm. On the contrary, the alternate version produces a minimum-quality solution in a bounded and known execution time. At run time, each task always starts running its primary version. If the system can offer the task enough time to finish its execution, the optimum response is produced. However, if the running task executes beyond a security threshold (at which it is detected that the task will not finish on time) the primary version is discarded and the alternate version is then executed until completion. From the feasibility test's point of view, only alternate versions are mandatory and thus required to be considered.

This task model is actually more general than the models presented in the two previous sections, since primary and alternate versions are not constrained to be of a certain type of algorithm. However, there are important issues which are not explicitly addressed in the model. For example, how does the system know at run time that a certain primary version will not finish on time. Or what happens if this version must be abandoned just when it was executing a critical section. Or, in this latter case, what happens with the partial results already computed (and possibly written) by the primary version. All these issues would need to be solved for a valid implementation of this model.

2.3.4 The Prologue-Optional-Epilogue Model

In [Aud96], Audsley et al. present a computational model designed for supporting the three task models for Imprecise Computation presented above. The task model is based on breaking down each application task into three sequential parts, named the *prologue*, the *optional* and the *epilogue* parts. The prologue and the epilogue parts are mandatory (and hence they must be guaranteed) while the optional part executes between them, if possible. The work in [Aud96] shows how the model can be used to build any of the models for Imprecise Computations and directly supports a mandatory final part for each task, which is not present in the other models. A mandatory final part is useful, for example, for ensuring that the solution computed by the task is actually communicated to the environment.

This work has two weak points: first, the model uses *fixed* offsets to delay the execution of each epilogue part with respect to its related prologue. This produces a rigid behaviour on the amount of time available for the optional components. And second, the scheduling of optional components are exclusively based on an acceptance test (proving that their execution will not jeopardize the schedulability of mandatory parts) and by a threshold function that checks whether or not the *utility* value achieved when running a particular optional component is enough. This kind of scheduling is based on the best-effort paradigm, which proposes some utility-based scheduling criteria. Besides producing significant overhead, the problem with the best-effort approach is that it is actually fixed: the scheduling of optional tasks only uses a single value per task to decide which tasks should be executed. Which value is used

and how it is computed is decided at design time. A more interesting behaviour would be achieved if, for example, specific knowledge about the problem being solved was introduced into the scheduling process, or if this process could automatically adapt itself to changes in the environment.

2.4 An Overview of Fixed Priority Preemptive Scheduling

In the early 1970's, a paper by Liu and Layland [Liu73] introduced two scheduling techniques, the Rate Monotonic (RM) algorithm and the Earliest Deadline First (EDF) algorithm in the context of real-time systems. In both algorithms, the run-time scheduling policy is not based on a pre-configured calendar but on task priorities. In RM, each task is assigned a priority off-line. This priority is inversely proportional to its period. At run time, the ready task with the highest priority is always selected first. In EDF, the run-time scheduler always chooses the task with the closest absolute deadline for execution. Although the task model originally proposed for both algorithms was rather restrictive (only pure periodic, independent tasks with deadlines equal to their periods and exact computation times), this work made a great contribution to the real-time community, since the authors provided a mathematical feasibility analysis for each algorithm.

The EDF algorithm evolved into the more general dynamic priority preemptive scheme. Within this scheme, task priorities are recomputed at run time as a function of some dynamic attribute (such as its closest deadline), so that the priority of a given task may change during its runnable periods, depending on the value of that attribute. This scheme is not discussed further because it is outside the scope of this thesis.

The RM algorithm turned into the fixed priority preemptive scheme, which nowadays is one of the standards in developing hard real-time systems. The fixed priority scheme maintains the original RM's run-time scheduling policy, where the fixed task priorities are the only criterion for selecting the running task. However, many features have been added to the original RM algorithm, making the scheme more complete or, in other words, less restrictive. On one hand, the original Rate Monotonic Analysis (RMA) (based on processor utilization) has been replaced by a more general feasibility test based on calculating the worst-case response time of each task. One of the benefits of this new test is that it no longer imposes how task priorities should be calculated. These priorities can be freely assigned by the application designer.

The scheme evolution has progressively relaxed the constraints that RM imposed over the task model. As expressed above, the original RM task model was very constraining, only permitting a system to be formed by a set of pure periodic, independent, hard tasks, with the deadline of each task being equal to that task period. Obviously, applications developed using that model were rather simple. Over the years, features added by several authors have made it possible to remove or relax many of the restrictions enforced by the original task model. The following discusses some of the main contributions in this field, in the context of single-

processor systems.

Sporadic tasks

Sporadic task is the name given to a non-periodic task having a hard deadline. Although Liu and Layland did not include tasks of this kind in their original task model, further work has demonstrated that any task which executes *at most* once each period can be considered in the original test for the RM algorithm. As a result, sporadic tasks with a known minimum inter-arrival rate can be directly introduced in the RM feasibility analysis.

Synchronization

In any concurrent application, resource sharing is a major topic. Tasks often need to communicate with other tasks, either via messages or by means of reading and writing in shared memory. In any case, the requirement of mutual exclusion in some regions of code has to be explicitly introduced within the test. Many approaches deal with this problem, such as atomic operations, lock-free data structures and semaphore-guarded critical sections (this is the most popular technique). When real-time tasks use semaphores for controlling mutual exclusion, specific protocols have to be used in order to know the worst-case time that each task may be preempted by a lower priority task, due to blocking. Protocols such as the Priority Ceiling Protocol [Sha90] or the Ceiling Semaphore Protocol [Raj89] bound (and calculate) these factors, thus making it possible to introduce them into the test.

Deadlines different from periods

The restriction that each task deadline be equal to its period has also been removed from the task model. The introduction of the Deadline Monotonic (DM) algorithm [Aud91] allowed for the definition of tasks with deadlines less than or equal to their periods, but it also imposed the method of calculating task priorities (a higher priority for a smaller deadline). This priority assignment is actually not necessary if the feasibility test mentioned above (which is based on the calculation of the worst-case response time of each task) is used. Nevertheless, this assignment has been shown to be optimal for tasks with deadlines less than or equal to their periods.

Other tests have been developed for systems whose tasks have *arbitrary* deadlines [Leh90, Tin93]. In systems where some tasks have deadlines greater than their periods, the DM priority assignment is not optimal; in this case, an optimal priority assignment is presented in [Aud93].

Soft tasks

In early hard real-time systems also containing soft tasks (tasks with soft, or null, time requirements), the traditional approach was to schedule the soft tasks in *background* in order to prevent them from jeopardizing hard task deadlines. The first attempt at improving the scheduling conditions of soft tasks while maintaining the application timing correctness was

the introduction of *aperiodic servers*[3]. The common goal of these algorithms is to add a special task, or *server*, to the application task set. This server in charge of executing soft tasks. The server is a regular periodic task, and can thus be considered by the feasibility test. In this way, the system actually reserves a guaranteed *bandwidth* to execute aperiodic tasks. Among the several existing techniques based on servers, the most relevant are probably the Polling Server, the Deferrable Server [Leh87], the Exchange Priority Server [Sto88] and the Sporadic Server [Spu90]. The Sporadic Server has recently been added to the POSIX standard.

There is a different family of soft task scheduling techniques, which is based on the idea of exploiting the processor spare time to execute soft tasks. These techniques are commonly called *slack stealing algorithms* [Leh92, Dav93a, Dav93b]. The spare time, named *slack time*, of each task is defined as the amount of time that this task may be delayed without missing its deadline. Slack stealing algorithms detect whether such slack time is available whenever the system has soft tasks to execute (typically, as soon as these tasks arrive to the system). If there are soft tasks and slack is available, then the system executes as many soft tasks as possible in the slack time. There are many types of slack algorithms (static and dynamic, exact and approximate, etc.), but in general they are simpler and more flexible than servers: first, no extra tasks have to be adjusted and introduced into the system so that the system remains feasible; and second, there is not an explicit off-line reservation of execution bandwidth but rather a dynamic usage of the available slack at any given moment.

Finally, there is still a third approach: the *Dual-Priority Algorithm* [Bur96]. This algorithm proposes executing each task at two different priority levels or *bands*. For each task, its upper band is above the priority level at which soft tasks are scheduled, while its lower band is below the soft task level. When a hard task is released, it starts running at its lower band. This allows soft tasks, if any, to preempt the hard task. When the task reaches a special time, called its *promotion time*, its priority is immediately raised to its upper band. From that moment on, it executes with no interference from soft tasks. The best feature of this algorithm is that the promotion time of each task is unique and can be calculated off-line. Thus, the run-time overhead related to the algorithm is practically nil. However, this algorithm has two main drawbacks: first, it increases the amount of required priority levels; and second, the system is not able to decide *when* it is necessary (or interesting) to run a soft task, since intervals of soft tasks execution are exclusively based on the (fixed) task promotion times.

Mode Changes

In some application domains, the system is made to work in different *modes*, potentially having a different set of running tasks in each mode. The typical example is a control system for an aircraft, which has at least three modes: take-off, regular flight and landing. In applications of this kind, the real-time system must not only have the guarantee that the task set for each mode is schedulable, but it must also ensure that hard deadlines are guaranteed within

[3] In early real-time systems, aperiodic tasks could not be considered as hard tasks in the feasibility test; for this reason, the term aperiodic task was used as an antonym of hard task.

the *transitions* between modes. An exhaustive review of this topic, as well as several recent contributions, can be found in [Rea00].

2.5 ARTIS

ARTIS (Architecture for Real-Time Intelligent Systems) [Gar96a] is an agent-based architecture which is focused on the development of intelligent real-time control applications with hard real-time constraints. An ARTIS application can be seen as an independent entity or *agent* which perceives its environment through a set of sensors, reasons about the perceived data and acts through a set of actuators, potentially changing the environment. An ARTIS agent can also communicate with other ARTIS agents which are working in the same environment.

There are two task models involved in the development of an ARTIS application. These models, called the *high-level task model* and the *low-level task model*, are now discussed.

The high-level task model is the model given to system designers for specifying the application in terms of their knowledge of the problem, that is, without having to be aware of the low level details of the running system.

This model offers a hierarchy of entities, with the *in-agent* (internal agent) as the entity at the top. An in-agent is an entity that *perceives* a subset of the environment, *reasons* about a subset of the problem and *acts* on (or communicates its solution to) the environment. These three activities are performed by the in-agent within a certain deadline and in a periodic manner. Normally, the control problem to be solved by the ARTIS agent is broken down into some smaller subproblems (which are not necessarily independent) and an in-agent is defined to deal with each subproblem.

An in-agent is naturally made up of *Multi-level Knowledge Sources* (MKSs) or sets of Knowledge Sources (KSs). The KSs are the architecture's computation individuals. In particular, an in-agent can have one or several *perception* MKSs, one or several *cognition* MKSs and one *action* MKS. Each MKS is defined as either critical or non critical. When a MKS is defined as *critical*, its first KS is considered as mandatory and the rest are considered as optional. Mandatory KSs are required to have bounded execution times, since their execution is guaranteed by the system; optional KSs can be unbounded, since they are only executed if possible. If the MKS is defined as *non critical*, all its KSs are considered as optional. For each in-agent, at least one perception and one cognition MKSs, and the action MKS are normally defined as critical. This is done in order to guarantee a useful (minimum-quality) response each time the in-agent is executed. At run time, the system will try to run as many optional KSs as possible, thus improving the agent's response.

The designer specification of an ARTIS agent, in terms of the high-level task model, is automatically translated into a real application, in terms of the low-level task model. In this second model, each in-agent becomes a *task*, made up of three consecutive parts: the *initial* part, the *optional* part and the *final* part. The initial part is built by adding the first (mandatory) KS of each perception critical MKS and each cognition critical MKS. The optional part is formed by the optional KSs of all perception and cognition MKSs. The final part is the

mandatory KS of the in-agent action MKS. Therefore, the initial and final parts of each task are only formed by mandatory KSs, and thus both parts will be guaranteed to execute (by means of a feasibility analysis). Optional parts will only be executed in moments when processor spare capacity is detected at run time; the detection of this capacity is done by using some slack stealing algorithm [Dav93a].

Another difference between mandatory and optional parts is that when a set of mandatory KSs forms an initial or final part, the KSs become a strict sequence. In fact, each initial and final part is considered as a *single* run-time entity. On the contrary, optional parts conserve their hierarchy of MKSs and KSs at run time. Thus, each optional KS can be independently chosen for execution. Furthermore, parts of each type are scheduled at a different level: mandatory parts are scheduled by the *first-level scheduler*, which applies a fixed priority preemptive policy; optional parts (that is, their KSs) are scheduled by a *second-level scheduler*, which normally applies some knowledge of the problem to its decisions. Because of this, this second scheduler is also referred to as the *Intelligent Server*.

It is worth noting that a great deal of the flexibility present in the ARTIS architecture derives from the fact of having two scheduling levels at run time. This allows for both a strict guarantee of mandatory parts and a flexible scheduling of optional parts. However, the ARTIS' low-level task model has limitations which could be relaxed. For example, although a task can have any of its parts missing, it cannot have a more complex structure than two mandatory parts separated by an optional part.

2.6 Summary and Discussion

Traditional research in hard real-time systems has focussed on developing scheduling paradigms which can provide a 100% guarantee on the system timing correctness. The guarantee is obtained when the application tasks can be mathematically proven to always keep their deadlines at run time. In order to ensure this predictable run-time behaviour, the scheduling paradigms typically impose several restrictions on real-time applications. The restrictions are mainly applied to two features: the computational model used for developing the application and the scheduling policy used at run time. Restrictions over the task model include, for example, knowing the maximum computation and resource requirements of each application task at design time. Restrictions over the run-time policy obligate the run-time system to always select the running tasks according to the same, fixed criteria. As a result of these restrictions, traditional hard real-time systems have implemented simple applications dealing with predictable environments.

As opposed to the simplicity of traditional hard real-time systems, this chapter has identified the abilities which FRTS are expected to have in order to feature a more intelligent, flexible and adaptive behaviour. According to these abilities, the chapter has then reviewed the main research efforts in this field. This review has presented that, in general, these efforts are focussed on either adding flexibility to the system computational model or introducing a more sophisticated run-time scheduling to the system. Taking both the requirements of FRTS

and the characteristics and limitations of the reviewed techniques into account, we consider that a framework for developing FRTS should at least support the following features:

- First of all, the framework must obviously support all the features currently available in traditional hard real-time systems. These features include tasks with hard deadlines, periodic and sporadic tasks, synchronization among hard tasks, etc. Applications developed by using the framework have to allow for a feasibility analysis which guarantees the deadlines of hard tasks.

- Since the framework has to provide a flexible response depending on the run-time situation, tasks with *different criticality* should be available. These include, as a minimum, hard tasks, soft tasks and firm tasks. Hard tasks are completely specified a priori and must be guaranteed off-line. Soft tasks do not have any constraints, and the system must try to execute them, typically in order to improve the quality of system response. Firm tasks have timing constraints but the system is not strictly required to always execute them; when a firm task is released, the system applies an on-line guarantee test to it, in order to check if its execution keeps its own constraint and maintains the guarantee of all hard tasks; depending on the result, the on-line test can accept or reject the firm task.

- The framework has to offer a homogeneous functionality to all the application tasks, in spite of being hard, firm or soft. In particular, at least two types of relationships should be provided. On the one hand, all types of tasks normally cooperate to achieve a common goal or result; hence, *synchronization* among tasks of any type must be provided. On the other hand, it is common in FRTS for some tasks to refine the solution achieved by other tasks; for example, a soft task implementing an unbounded algorithm is typically used for improving the outcome provided by a hard task, which normally implements a much simpler algorithm. In scenarios of this type, the *precedence relationship* among tasks also has to be addressed.

- The typical run-time scheduling policy in a hard real-time system takes into account the same, fixed criteria for choosing the running task, with these criteria normally being based on some timing attributes of tasks. For hard tasks, this is the only way to be able to guarantee the task timing constraints a priori. However, if the framework offers tasks with different criticality, more sophisticated scheduling can be offered to non-hard tasks. This scheduling can take into account the *quality* aspects of tasks, such as their importance, appropriateness, utility, etc.

- The scheduling policy (or policies) adopted by the framework have to schedule hard tasks in such a way that they always meet their deadlines as well as schedule non-hard tasks in an intelligent and adaptive manner. The scheduling has to achieve the overall goal of adapting the system response to the current scheduling conditions (from low-load to overload situations) and to the current status of the environment. In this sense, an interesting feature of the scheduling policy is to be able to include particular

knowledge about the problem being solved. By means of this knowledge, the system can dynamically focus on executing some tasks rather than others.

- Since FRTS may deal with complex, dynamic and unpredictable environments, the framework must have fault tolerance capabilities. These capabilities are needed, for example, when a hard task overruns because of an unexpectedly large data input.

3

An Overview of Real-Time Operating Systems

The previous chapter has explained that the development of a hard real-time system requires a special theoretical *framework* in order to be able to guarantee the system timing correctness. This framework (normally related to a scheduling scheme) imposes some restrictions on the application tasks and enforces a specific run-time behaviour in aspects such as scheduling policy, periodicity of tasks, synchronization mechanisms, task priorities, etc. A run-time support system (or *operating system*) for hard real-time systems has to offer such functionality, while also meeting requirements of reliability, efficiency and predictability. Therefore, no matter how good the framework is or how carefully the application design has followed it, without this special run-time support it is not possible to successfully implement a hard real-time system. Unfortunately, all these properties are usually not available in regular operating systems. Thus, a special system, called *Real-Time Operating System* (RTOS), is required. Many RTOSs have been developed over the years, with quite different capabilities. This chapter reviews the current trends of RTOSs and their suitability in supporting FRTS. In particular, two types of RTOSs are discussed: RTOSs based on the real-time extensions of the POSIX standard and systems combining a real-time kernel and a General-Purpose Operating System (GPOS).

3.1 Introduction

According to [Ram94], there have been three traditional approaches for developing RTOSs: small, fast proprietary kernels; GPOSs which have been enhanced with some real-time features; and RTOSs developed within research groups. However, another classification of RTOSs which is closer to the purposes of this chapter is presented here. This classification places RTOSs in to groups according to their functionality: small kernels for simple embedded systems and bigger RTOSs offering more sophisticated support[1].

The first group is generally formed by commercial products such as C executive, CMX, eCos, pSOS, etc., mainly oriented towards the embedded industry. These kernels are normally provided as a run-time support system included in a cross-compiling (or *host/target*) development environment. In this kind of development environment, the application is coded in the *host* computer, normally a full-featured system with compilers, debuggers and all sorts of Graphical User Interface (GUI) tools. The application is then compiled by the cross compiler and linked with the run-time kernel, in order to produce a binary executable system for the *target* computer, where it will be executed. The functionality of these kernels is usually very limited because their aim is to be small (in memory size), fully predictable and efficient, in order to implement tiny hard embedded applications. These products are normally focussed on providing a good set of development tools in the host environment and to support a great variety of target hardware platforms, including special embedded boards.

The second group of RTOSs is formed by systems offering a greater number of features and more sophisticated support. These systems are oriented to soft and hard real-time applications of medium to high complexity. Typical features of these RTOSs are: several scheduling policies, network accessibility, synchronization/communication mechanisms, multi-processor support, etc. In the past, this higher level of support was provided by several proprietary solutions, including multi-featured RTOSs such as QNX, VxWorks, LynxOS, etc. However, these particular solutions are lately converging into a common trend: to offer the real-time extensions of the POSIX standard [POS90] as the application programming interface. The reason of this convergence is that the POSIX real-time extensions are commonly accepted nowadays as the set of real-time capabilities that a RTOS should offer. These real-time extensions of POSIX are also being adopted by several Unix-like GPOSs, such as HP-UX, Solaris, Irix, etc. These operating systems, which had formerly adopted the non-real-time part of the standard, have found the real-time extensions a very natural way of also providing real-time support[2].

The review presented so far deals with the support that RTOSs provide to regular real-time systems. The capabilities required by FRTS are discussed herein. Due to their special complexity, FRTS require very sophisticated run-time support, specially for managing different types of tasks and utilizing different scheduling paradigms. As shown above, the current al-

[1] Information about many RTOSs in this section has been extracted from [RTE] and the RTOSs' own world-wide-web pages.

[2] Actually, most of these systems cannot be used for implementing hard real-time applications, since they are excessively big and complex, and their original internal design has been optimized for the average case; as a result, they are not completely predictable.

ternative with the state-of-the-art RTOSs would be to implement FRTS by using the POSIX real-time extensions. However, there is another trend of RTOSs which is lately taking place within both the academic community and industry: to combine a hard real-time executive with a GPOS in order to provide both real-time capabilities to hard tasks and regular, full-featured support for non-real-time tasks. The natural combination of hard and soft tasks, each one with its own scheduling paradigm, brings this trend close to the requirements of FRTS. The rest of the chapter presents both alternatives and discusses their appropriateness for building FRTS.

3.2 The Realtime Extensions of POSIX

The POSIX standard [POS90], developed by the Institute of Electrical and Electronic Engineers (IEEE), is an attempt to standardize the Unix operating system at a programming level, that is, to define a common environment and interface for all Unix-like operating systems and to make the source code of programs directly portable to all these systems.

The first POSIX document did not contain any reference to real-time systems. However, two new documents describing the system real-time capabilities were soon added to the standard: the *Real-Time Extension* [POS93] and the *Threads Extension* [POS95]. Based on these documents, POSIX has also included the definition of four different *profiles* [POS97] or different levels of requirements, depending on the complexity of the system. The following subsections revise the general real-time POSIX facilities as well as the four profiles.

3.2.1 POSIX facilities

There are nine groups of facilities included in the standard:

1) **Semaphores, Mutexes and Condition Variables**. These minimal synchronization primitives have to be provided, being the base to the application for building more sophisticated mechanisms, if needed.

2) **Process memory locking**. This enforces the operating system to use only physical memory to store all running real-time processes, instead of using typical Virtual Memory (VM) mechanisms, which normally keep part of the process memory in some secondary storage device.

3) **Shared memory and memory-mapped files**. Shared memory allows various processes with independent memory spaces to share a portion of the physical memory, as a means of sharing large volumes of data. The memory mapping facility permits a process to access a file as if it were part of its memory. This latter facility is normally not allowed in hard systems, since the time involved in accessing a file inside a secondary storage device is difficult to bound.

4) **Priority scheduling**. The operating system must schedule concurrent processes with a preemptive priority-based policy, in order to allow the applications to decide in which

order ready processes may use the processor.

5) **Real-time signals**. Extension of the traditional Unix signal mechanism to asynchronously notify signals (or events) to the application in a reliable manner.

6) **Clocks and timers**. The traditional POSIX time management functions do not have resolution enough for real-time systems. This facility increases the ability of both the operating system and the application to measure and manage time.

7) **Message passing**. This provides a high-level mechanism for communication among processes or threads.

8) **Synchronized Input and Output (I/O)**. I/O operations of this kind permit the application to ensure that the information being handled is physically stored in secondary storage devices in order to provide data integrity.

9) **Asynchronous Input and Output**. This permits the application to queue requests to data transfer and to be notified when these operations are completed.

3.2.2 POSIX profiles

The POSIX Realtime Application Support (AEP) [POS97] includes the definition of four different real-time *profiles*, describing the real-time requirements of systems with different levels of complexity:

a) **Minimal Realtime System Profile**. This profile describes the simplest type of real-time (typically embedded) system. The programming model includes one single process, i.e. one address space, containing one or more threads. No file system (secondary mass storage device) is needed, and devices are operated by memory-mapped Input and Output (I/O) or specific I/O interfaces. This profile assumes one processor with no Memory Management Unit (MMU).

b) **Realtime Controller System Profile**. This is an extension of the minimal profile, in which a support for asynchronous I/O and a file system interface has been added. Mass storage devices are not strictly required, and file system may be implemented in memory.

c) **Dedicated Realtime System Profile**. This profile is an extension of the minimal profile on which support for multiple processes is added. A common interface for devices and files is required, but not a hierarchical file system. As many processes share memory, memory locking must be provided. This profile assumes one or more processors with or without MMUs.

d) **Multi-Purpose Realtime System Profile**. This profile includes all the functionality of the other three profiles. The purpose is to be able to run a mix of different real-time and non-real-time processes. The hardware assumed includes memory with MMUs, storage devices, displays, network support, etc.

3.2.3 Discussion

The POSIX real-time extensions are a fixed set of well-defined functions which are intended to be used for building real-time applications in a Unix-like RTOS. The system functionality established by the standard is actually divided in four groups or *profiles*, with the functionality of each group included within the more complex, next group. This division in profiles has been designed in order to allow the RTOS to incorporate a functionality level according to the complexity of the applications intended to be implemented on the system.

The POSIX real-time extensions (specially, those included in the *Minimal Realtime System Profile*) actually provide a good environment for developing simple and predictable real-time applications: the support for threads, priority-based scheduling, mutexes, timers, etc. are commonly required by almost any real-time application. For applications demanding a more sophisticated support, POSIX proposes the other three profiles which progressively establish a larger set of features available to the application. In this sense, it must be noted that these features are related to more complex *system services* (such as different synchronization facilities, memory-mapped files, asynchronous input/output, network accessibility, etc.), but they do not directly support a richer task model nor more sophisticated scheduling policies. On the other hand, as profiles increment their support, it becomes more difficult for the RTOS to provide this support in a completely predictable manner.

As a result, the POSIX real-time extensions are not appropriate for building FRTS in our opinion, since these extensions have not been designed to directly support the features stated at the end of Chapter 2, which include different task types, combination of different scheduling paradigms, synchronization of tasks of different criticality, support for timing exceptions, etc. Although a framework supporting some of these features could probably be implemented by using POSIX, this would have two serious drawbacks:

- The framework support would have to be provided to the application as a *custom kernel*, implemented at the top of the POSIX interface functions. Such a kernel would suffer from the limitations of having a pre-fixed set of services available for implementing a complex support. This would probably lead to a very inefficient kernel. Furthermore, the kernel would suffer from a double overhead: its own overhead plus the overhead of the POSIX RTOS it is using.

- Currently, not all the POSIX RTOSs provide detailed time measurements of each interface function for each possible target platform. Without these times (or without the RTOS source code) it is not possible to study the time behaviour of the *custom kernel*, and, thus, the system's feasibility analysis could not be performed. Therefore, the apparent advantage of having a POSIX portable code is limited in this case to having these measurements available on the system to be ported to.

As a conclusion, the alternative of building a general framework for FRTS by means of the POSIX real-time extensions may be interesting in systems where 1) these extensions permit the implementation of all the framework features, 2) the extensions are perfectly measured

and 3) the system overhead is not a major design topic. Until the current POSIX standard incorporates mechanisms closer to FRTS requirements, it seems more appropriate to support them *natively*, either by implementing a new kernel from scratch or by modifying an existing kernel.

3.3 Real-Time Kernels underneath General-Purpose Operating Systems

One of the traditional approaches of developing a RTOS, as shown above, is to incorporate real-time capabilities to a GPOS. This approach is rather problematic, since GPOSs are normally big operating systems whose design principles are completely opposite to real-time systems: fair sharing of resources, good average response time, FIFO queueing of requests, etc. Actually adding real-time features to them is an arduous labor which includes revising (and often re-implementing) a great part of the system's source code.

As a completely different approach to achieve the same goal (obtaining a RTOS by means of modifying a GPOS), a new idea has arisen: instead of changing a great part of the GPOS' source code, the new approach proposes placing a real-time executive (or kernel) *underneath* the GPOS; that is, between the hardware and the GPOS' kernel. This idea, formerly expressed in [Lyc78], has three significant advantages. First, it adds a second level of execution, meaning that the executive can run hard tasks separate from (and with preference to) GPOS' processes. Second, predictability is more easily achieved, since the executive isolates the GPOS from the hardware by intercepting all the hardware interrupts. Then, actions performed by the GPOS, even by its kernel, cannot cause delays to the executive's execution. And third, the GPOS itself is almost unchanged, meaning that it maintains its regular capabilities, which actually can support the designer in building the real-time application. With this approach the host system can be the target system as well.

The following three sections revise the most important systems developed by applying this idea: the Slotted Priorities architecture, Real-Time Linux and a few systems based on Windows NT.

3.3.1 Slotted Priorities

The Slotted Priorities (SP) architecture [Bol97] is a theoretical model describing how a Real-Time Kernel (RTK) can be placed below a GPOS in such a way that both systems share the same hardware. The RTK allows the execution of real-time threads or tasks in such a way that they always have preference of use (without delays) of all physical resources (this includes the use of the processor) with respect to the GPOS and its processes. To do this, the RTK accesses the hardware interrupts directly, but prevents the GPOS from doing so by providing a mechanism called *virtual interrupts*. In this mechanism, the GPOS is unable to disable the timer interrupt, which is programmed by the RTK scheduler to determine when to run the

real-time threads and when to run the GPOS. This achieves the necessary predictability for building hard real-time systems. In [Bol97], it is also shown how this model can be actually implemented above a real GPOS, the IBM MicroKernel.

Specifically, the SP architecture provides an execution model, a resource taxonomy and a programming model. The execution model establishes how to share the resource *processor* between the GPOS and the RTK. The model uses a kind of *cyclic executive* technique, by which the processor time is split into two fixed-duration, alternative intervals: the *real-time minor cycle*, on which the RTK schedules ready real-time tasks, and the *non-real-time minor cycle*, on which the GPOS schedules and executes its regular processes. The beginning of each cycle is signaled by a timer interrupt (programmed by the RTK). The resource taxonomy classifies physical devices in a hierarchy, depending on how they can be shared between the RTK and the GPOS. For each resource to be shared, the model proposes building an *executive*, which is in charge of making sure that the resource is available for the real-time tasks whenever they request it. The taxonomy also provides some guidelines for designing these executives, which can be very straightforward or really complex, depending on the device's characteristics. The programming model includes a set of interface functions to create real-time tasks, which can only be purely periodic, independent tasks. Each time a new real-time task is created, the RTK recomputes the duration of the real-time and non-real-time minor cycles, in order to be able to meet the deadlines of all tasks, including the new one. Inside the real-time minor cycles, the RTK uses a scheduling policy based on the Earliest Deadline First (EDF) policy.

The SP architecture is an interesting theoretical approach which describes how to have a hard RTOS and a GPOS sharing the same hardware. The work includes a special feasibility test for the system, by which the schedulability of applications using the architecture can be proved. However, the actual sharing between the RTK and the GPOS is rather inflexible, in the sense that the execution of real-time and non-real time tasks are confined inside fixed-duration intervals. This approach is valid for an EDF scheduling policy inside the real-time minor cycles, but it would be difficult to implement other real-time policies, like the fixed-priority preemptive policy.

3.3.2 Real-Time Linux

Real-time Linux (RT-Linux) [Yod99] is another example of a system with a real-time kernel underneath a GPOS, which is Linux in this case. Both Linux and RT-Linux have been released under the terms of the GNU's General Public License [GNU91], which means that they can be freely distributed and modified (while still preserving the original authors' copyright). In RT-Linux, the real-time executive provides support for running hard *real-time tasks* (rt-tasks) and turns the entire Linux into the lowest-priority task. All rt-tasks execute in the Linux kernel's address space and with kernel privileges.

Given that the theoretical foundations are the same in RT-Linux and the SP architecture, the former has some significant advantages over the latter:

- The design of the executive is simpler in RT-Linux: the Linux kernel is completely

unaware of the existence of the executive, and it just becomes the lowest-priority task for the RT-Linux scheduler.

- Scheduling in RT-Linux is more flexible, since the intervals of execution of the real-time and non-real-time sides are not fixed: Linux executes whenever no rt-task is ready, and an rt-task can be immediately ready to run at any moment, as a response to a hardware interrupt (normally, the timer interrupt). In addition, changing the scheduling policy of the RT-Linux kernel is straightforward.

- Although the theoretical foundation is the same in both cases, the RT-Linux's *soft interrupts* mechanism is more sophisticated and ambitious than the SP's *virtual interrupts* mechanism. The SP's mechanism simply prevents the IBM MicroKernel from disabling a particular interrupt which it does not use. On the contrary, the mechanism in RT-Linux permits the real-time kernel (and the rt-tasks) to catch *any* interrupt (whether or not in use by Linux), and the rest are still available for Linux.

- At a practical level, the most obvious advantage is that the GPOS is a free, full-featured, Unix-like operating system. A system like this has a full set of tools which makes it easier to develop the real-time application.

- In RT-Linux, the set of permanent modifications on the Linux kernel are very few. Most of the real-time support, including the real-time scheduler itself, is compiled as kernel modules that can be loaded and unloaded dynamically, just by typing the appropriate commands in a Linux shell. This mechanism is also used to run the real-time application.

The original support provided in RT-Linux v1 (version 1) includes a fixed priority preemptive scheduler, periodic and non-periodic (single execution) rt-tasks and a communication mechanism called RT-FIFO. RT-FIFOs are actually buffers (allocated in kernel address space) which allow bidirectional communication between Linux processes and rt-tasks. From the rt-tasks' point of view, RT-FIFOs are accessed by calling special reading and writing functions, which are atomic and non blocking. From the Linux processes' point of view, RT-FIFOs are accessed like traditional UNIX character devices. Communication between rt-tasks and Linux processes permits designing a real-time application by separating the activities that must be done in real-time (which are executed by rt-tasks) from the activities which do not have hard timing requirements (which can be implemented as Linux processes). This allows keeping the real-time system as simple (and fast) as possible, which is one of the major goals of RT-Linux.

Currently, the development of RT-Linux is going towards two main topics: the Simultaneous Multi-Processor (SMP) support and a POSIX interface according to the Minimal Realtime System Profile described in Section 3.2. On the other hand, probably the main deficiency in RT-Linux is not having a detailed feasibility test available with the costs of the services provided by the real-time scheduler. In this sense, the only way of knowing if an application will meet its timing constraints is to execute it, as was experienced in [Hum99].

3.3.3 Systems based on Windows NT

Some real-time extensions to the Windows NT® operating system[3], which are based on similar ideas present in SP and RT-Linux, have appeared over the last years. Besides being appropriate for adding real-time capabilities to a GPOS, in the case of NT it is the only possible approach, since the NT kernel's source code is not available. For this reason, the extensions can only be placed at the lowest level of the NT's architecture, that is, inside or at the same level as the NT's *Hardware Abstraction Layer* (HAL). This layer is in fact available in order to implement device drivers.

VentureCom's RTX [Car97], Radysis' INtime [Fis98] and Imagination's Hyperkernel [Hyp] are three existing commercial, NT-based, real-time kernels. These extensions provide both a real-time scheduler (normally implemented upon modifications on the HAL) in order to run real-time threads, and an extension of the Win32 system library in order to provide some real-time capabilities to regular NT threads, plus some means of communication between real-time and NT threads. On the other hand some specific support, different for each solution, is provided as well. Currently, the most sophisticated support is probably offered by the INtime system, which provides memory protection for real-time threads (running in user-mode), mailboxes, semaphores, shared memory between real-time and NT processes, access from the real-time threads to the NT's file system and to the network, etc.

The solutions presented in this section are all commercial products and hence their source codes are not available for further independent development. Nevertheless, their existence demonstrates the growing interest within the industry of having a widely-used, full-featured GPOS enhanced with hard real-time capabilities.

3.4 Summary and Discussion

This chapter has presented a review of real-time operating systems and has discussed their applicability for developing FRTS. In particular, systems offering the real-time extensions of the POSIX standard are appropriate for building traditional real-time systems, but they clearly lack the facilities for directly providing a sophisticated task model or for integrating more than one scheduling paradigm. Thus, implementing these features in POSIX would imply to build a custom kernel at the top of the POSIX facilities. This kernel would suffer from two drawbacks. First, it would be implemented by means of a pre-fixed set of facilities, which would probably lead to a inefficient implementation of some features and to a high overhead (produced by both the kernel and the POSIX facilities which are supporting it). Second, the a complete feasibility analysis would only be possible if accurate measurements of the costs of the POSIX facilities were available, which only a few POSIX RTOSs actually provide. As a result, FRTS built on top of POSIX can easily suffer from a great deal of overhead, which, in addition, may be very difficult to determine.

[3] Windows NT is a registered trademark of Microsoft Corporation. All other product names in this section are trademarks of their respective companies.

The second part of this chapter has reviewed some systems corresponding to a new trend in developing RTOSs. This new trend proposes incorporating a small, fully predictable real-time kernel underneath a GPOS, making both systems share the hardware. This preserves both the hard real-time capabilities of the kernel and the original capabilities of the GPOS. This philosophy is very close to the framework for FRTS proposed in this thesis, which is based on a two-level architecture that separates the application into two cooperative disjoint sides: a real-time side containing the part of the application with hard real-time constraints, and a non-real-time side containing all the optional activities. As a result, this framework has been implemented by enhancing the capabilities of one of these RTOSs, called Real-Time Linux.

4

A New Framework for Flexible Hard Real-Time Systems

This chapter introduces a new theoretical framework for FRTS supporting the requirements expressed at the end of Chapter 2. The framework is based on the original ARTIS architecture, to which new features have been added and some limitations have been relaxed or removed. In fact, the framework can be seen as an extended version (or an evolution) of ARTIS. In particular, the framework consists of three parts: a task model, a software architecture and a set of services available to the application tasks. The first two parts are described in this chapter, whereas the services currently developed are introduced in the two following chapters. In this chapter, the framework's task model proposes to design each application task as a sequence of components, which may be either mandatory or optional. The software architecture proposes to organize the system in two computational or scheduling levels, one for running the mandatory components and another for running the optional components, and to utilize a different scheduling paradigm for each level. Besides providing a high-level definition of the framework, this chapter also describes how this framework can be designed and implemented in a real run-time support system. This system is called *Flexible Real-Time Linux* (FRTL). A general description of FRTL is presented in this chapter, emphasizing the aspects directly related to the task model. A complete description of this RTOS may be found later in Chapter 7.

4.1 Introduction

In real-time systems, the *task model* (also called the computational model) is the framework specifying the type of tasks the application may define along with their characteristics and constraints. This model has to be consistent with the feasibility test, in order to be able to study the application timing constraints. According to the requirements for FRTS stated in Chapter 2, a task model for FRTS is normally more sophisticated than a model for a traditional hard real-time system. The reason for this is that a model for FRTS typically defines several types of tasks, with each type having a different criticality level.

A task model for FRTS must not only define the characteristics and constraints of tasks belonging to each type but also their possible interactions, providing an overall environment where all tasks can usefully cooperate and where timing constraints are never jeopardized. In particular, as soft tasks are normally used to enhance the results computed by hard tasks, the task model must at least address *synchronization* and *precedence* relationships among tasks. These relationships permit tasks to communicate their results (synchronization) and allow some tasks to be defined to execute after some other task(s) (precedence). It is also important for the task model itself to be as flexible as possible, in the sense that it allows for the definition (even inside the same application) of tasks belonging to several different types. These types should range from the most basic (periodic, independent) hard or soft task to a highly sophisticated task which can be defined in terms of cooperating hard and soft tasks forced to execute in a certain given order.

It is also common for a framework for FRTS to mix several cooperating scheduling policies. In such frameworks, the potential interactions among these policies have to be fully described. The problem of mixing or combining different policies is that the resulting policy should preserve the abilities of each one. At least, the scheduling policy for hard tasks must maintain its compatibility with the feasibility test, because, otherwise, the schedulability of the system could not be tested. An interesting feature in this context is the possibility of defining custom scheduling policies depending on the characteristics of each particular application. If the framework allows this, it should clearly establish how to incorporate these new policies to the system.

This chapter presents a new framework for FRTS, which takes into account both the general FRTS requirements specified in Chapter 2 and the issues discussed above. The organization of the chapter is as follows: Section 4.2 presents the framework's task model, which is based on the concept of *component*. Then, Section 4.3 formalizes the presented model. Section 4.4 introduces the framework's software architecture, which proposes organizing the application in two scheduling levels. This section describes each level in detail and specifies their possible interactions. Section 4.5 introduces the FRTL run-time system, emphasizing the implementation of the framework's scheduling levels. Then, Section 4.6 specifies the set of task-related interface functions available within this run-time system. Finally, Section 4.7 summarizes the chapter and discusses its contributions.

4.2 The Task Model

4.2.1 Tasks

The task model now presented proposes a real-time application as a fixed set of real-time tasks. The term *fixed* means that the full set must be determined before the application is executed, because it is intended to be verified by an off-line feasibility test. Therefore, the set of tasks is supposed to be verified at design time and no more tasks can be added at run time.

In this model, each task is characterized by the following basic timing attributes: a *period*, a *deadline*, an initial *offset*, a *worst-case execution time* or *wcet*, and a *worst-case blocking time* (derived from the use of a special synchronization protocol, introduced in Chapter 5). Each task is also assigned a fixed priority, which is called its *base priority*. All these attributes are *static*, in the sense that they are specified at design time and do not change at run time. A task can also be attached to a timing exception handler, which is a special function to be called if the task produces a timing exception. This possibility is further discussed in Chapter 6.

4.2.2 Components

This task model proposes each task to be composed of a *sequence* of *components*. A component can be described as a subpart of a task implementing a concrete set of actions and having a particular criticality. According to this criticality, components are classified in two categories:

a) **Mandatory components**. These components are required to execute under any circumstances, and for this reason they must have bounded and known computation times in order to be considered by the feasibility test. Thus, each mandatory component is characterized by its *wcet*, which has to be provided by the designer.

b) **Optional components**. These components are not strictly required to execute and, rather than being considered by the feasibility test, they are executed in the processor spare capacity. Each optional component is thus characterized by the portion of this capacity that the system should spend in executing the component, as discussed below.

The task model proposes determining the intervals of processor spare capacity by using a *slack stealing algorithm* (see Section 2.4), because algorithms of this sort naturally fit the scheduling requirements of optional components in two main aspects. First, slack algorithms permit exploiting the spare capacity in a flexible manner; that is, no bandwidth reservation is made off line; instead, the available spare capacity is dynamically obtained at run time when required (i.e., when optional components are ready to be executed). Second, the internal behaviour of slack time algorithms naturally maintains the precedence relationship among the components inside a task. Slack algorithms typically delay runnable hard tasks during the intervals where there is slack available, in order to run soft tasks. In this task model, each time a task wants to schedule an optional component, the slack algorithm is used for delaying

the task's following mandatory components (as well as all the mandatory components of all other ready tasks). In the context of slack stealing algorithms, the available slack time is normally computed at each priority level which, in practical terms, means *for each task*. For this reason, the run-time system also needs to know how to distribute this slack among the optional components belonging to each particular task. The task model thus characterizes each optional component by its *slack fraction*, which is a real number between zero and one expressing the component's portion of the total slack time available for the task it belongs to.

Figure 4.1 illustrates the task model concept of an application task. The task has a deadline of 29 time units, with all other timing attributes being irrelevant for the purposes of the example. The figure includes first the task structure, consisting of five components, and then depicts a chronogram showing a possible execution of the task. The task execution assumes that each mandatory component exactly consumes its *wcet* and that the task does not suffer any interference from other tasks. As a result, the slack time available for the task is 24 time units (calculated as its deadline (30) minus the sum of mandatory component *wcet*s (5)). Then, these 25 units of slack time are distributed between the task's two optional components according to their respective slack fractions: component 2 gets 20% of this slack (5 units) and component 4 gets the other 80% (20 units).

Due to this method of scheduling optional components, the amount of slack time scheduled for a particular component inside a task actually establishes the component's validity interval. Beyond this interval, the component cannot be allowed to continue execution because it would either jeopardize mandatory components or steal the slack time corresponding to other optional components. Therefore, if the slack time scheduled for an optional component expires before the component is able to completely finish, then this component has to be killed. In the example depicted in Figure 4.1, optional components 2 and 4 have a maximum of 5 and 20 time units, respectively[1]. If either of them is not able to finish within its respective interval, then it is killed. However, killing an optional component does not necessarily prevent it from producing valid results, as explained below.

According to a more detailed classification, optional components can be further divided into the three following types:

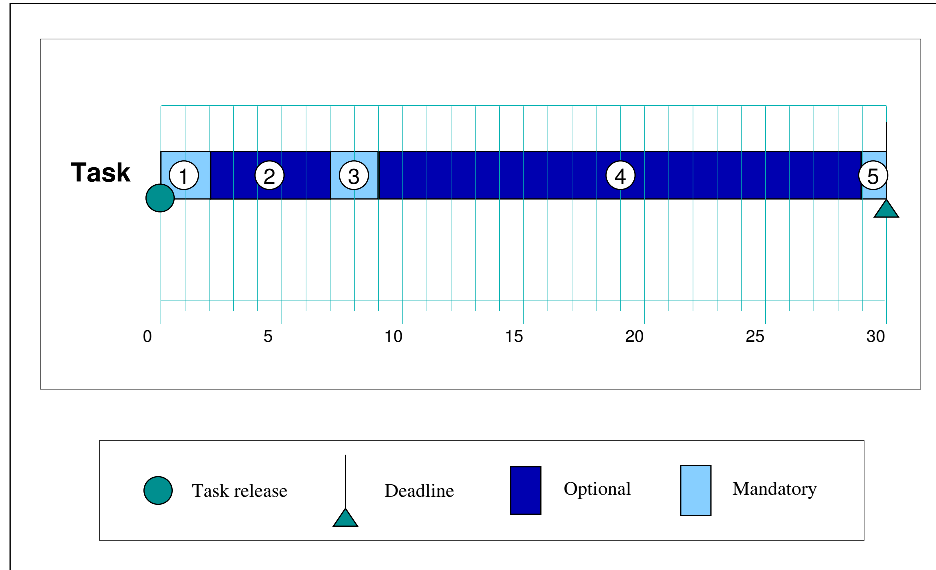
a) **Unique Version** (UV). This is the simplest type of optional component in the model, consisting of a single computational unit or *version*. UV components may be used to implement any algorithm (bounded or unbounded). In particular, *monotone algorithms*, also called *Anytime Algorithms* [Dea88], are specially suited because they produce valid results even if the component is killed before being allowed to finish.

b) **Successive Versions** (SV). A component of this type internally consists of a sequence of *versions*, which must be executed in the strict order specified in the sequence. SV components are appropriate for implementing a set of refinement algorithms, each one enhancing the solution computed by the previous one, like the *Progressive Deepening*

[1]At run time, the actual amount of slack time available for each optional component is dynamically calculated, depending on aspects such as mandatory components consuming their *wcet* or not, interference of other tasks, etc.

Component	Type	Wcet	Slack Fraction
1	mandatory	2	–
2	optional	–	20%
3	mandatory	2	–
4	optional	–	80%
5	mandatory	1	–

(a) Internal structure of the task.



(b) Execution of the task.

Figure 4.1: Example of a task structure.

algorithms [Win92]. If a SV component is killed, the result computed by its last finished version is used as the component's result.

c) Alternative Versions (AV). This type of component is made up of a set of versions, without any order. Each time the component is run, only *one* of them can actually be selected for execution. Thus, these components are provided in order to implement different approaches solving the same problem, normally with different degrees of accuracy and different (estimated) computation times, like in the *Multiple Methods* discussed in [Les88].

Thus, the task model allows the application developer to think of a task in terms of reusable smaller pieces or components, knowing that the run-time system always executes the mandatory components and offers the maximum possible amount of execution time (slack time) to the optional components. For this reason, mandatory components are intended to implement

predictable and simple algorithms with hard constraints and to access physical devices, while optional components are intended to implement complex, unbounded algorithms enhancing the outcomes computed by mandatory components. Table 4.1 summarizes the types of components along with their characteristics.

Name	Type	Guaranteed	Feature	Versions
Mandatory	mandatory	yes	wcet	1
Unique Version	optional	no	slack fraction	1
Successive Versions	optional	no	slack fraction	1 or more
Alternative Versions	optional	no	slack fraction	1 or more

Table 4.1: Types of components in the task model.

From the viewpoint of designing each application task, it is important to note that the task model does not constrain the amount of components forming a task nor the particular arrangement of these components inside the task. A task may be designed to consist simply of one component, either mandatory or optional, or to be formed by several components, with each one being of any of the defined types. Thus, the application developer is entirely in charge of deciding the structure of each task, depending on the activity to be performed by that task.

This model, based on the concept of component, offers the following advantages. First, the model guarantees that a minimum-quality solution will always be produced in a predictable time, by means of executing the application mandatory components. The model also helps in achieving the best quality enhancement at run time, by means of providing as much slack time as possible to the application optional components. Second, this model subsumes the four models for Imprecise Computations presented in Section 2.3 (the Milestone Method, Sieve Functions, Multiple Versions and the Prologue-Optional-Epilogue Model). As a result, each application task may be designed by following any of them. Third, the model favors code reusability, in the sense that a common part of several tasks may be coded as a separate component, and then inserted in whichever task needs it. Furthermore, the same component may be considered mandatory in one task and optional in another. And fourth, it allows a more complex task structure than any other previous model for FRTS. For example, a task can allocate one or more optional components after any mandatory component, in order to enhance the mandatory component solution; a task can also allocate a mandatory component after an optional component in the middle of the task structure, in order to immediately communicate the result computed by the task up to that moment.

4.2.3 Synchronization

The ability of sharing resources among tasks is a common requirement in real-time systems. In particular, memory sharing is often needed as a communication mechanism for tasks. For this reason, many real-time task models allow tasks to safely share memory by providing a special, real-time synchronization protocol which maintains both data consistency and task timing constraints. However, architectures mixing hard and soft tasks normally only permit synchronization among hard tasks. This model allows mandatory *and* optional components to directly (and homogeneously) share memory, by using special synchronization mechanisms. These mechanisms are introduced and evaluated in Chapter 5.

4.2.4 Dynamic Components

When real-time applications have to deal with changing environments, it may be necessary for the application to be able to execute algorithms that solve special and rare situations. In this case, since these algorithms are seldom (or maybe never) needed, it is not reasonable for the system to *a priori* reserve a bandwidth for their execution by including them in the feasibility test[2].

As a more reasonable approach for coping with this requirement, the task model permits the definition of special components. These components do not belong to any task nor are considered in the test, and they are not released until the application decides to do so, after detecting a special run-time situation. These components are called *dynamic components*, and can be defined as either optional or mandatory:

a) **Optional dynamic components** do not have any timing constraints and. because of this, they do not need any special scheduling mechanism to be defined in the task model. In particular, when one of these components is released, it simply becomes another ready optional component to be scheduled in slack time. Optional dynamic components can be defined as Unique Version, Successive Versions or Alternative Versions, just as regular optional components.

b) **Mandatory dynamic components** are characterized by a timing constraint, expressed by a *wcet* and a deadline, that the system must *try* to meet. When a component of this type is released, the system applies an *acceptance test* to it, in order to determine if it can be executed by its deadline while maintaining the deadlines of the application tasks. If the test is satisfactory, the component is accepted and the system is committed to execute it before its deadline. On the contrary, if the test fails then the component is rejected. These components are normally referred to as *firm tasks* in the literature [Dav95].

Overall, the concept of dynamic component completes the task model features by adding the ability of dynamically releasing execution units not included in the initial, fixed set of guaranteed application tasks.

[2]Obviously, unless they have to be 100% guaranteed. If so, these algorithms must be implemented as task mandatory components

4.2.5 Set of Constraints Imposed over the Model

Finally, the constraints enforced to the task model are now presented. These constraints are necessary in order to make the task model consistent with both the feasibility test and the software architecture introduced later in this chapter. In particular, there are three main constraints in the model:

- Mandatory components cannot suspend themselves. This is a common requirement in hard real-time systems, and it is derived from the inability of feasibility tests to verify tasks which exhibit this behaviour.
- The deadline of each task is assumed to be be less than or equal to the task's period, since the actual slack stealing algorithm used for scheduling optional components has been designed by assuming this constraint. Having arbitrary deadlines makes this algorithm more complex and inefficient.
- Sporadic tasks are not supported by the task model. The use of sporadic tasks worsens the behaviour of the slack stealing algorithm, as shown in [Dav93a]. Furthermore, sporadic arrivals of hard tasks make it impossible to exactly know the length of *slack intervals*, which is a key concept in the software architecture introduced in Section 4.4. Nevertheless, applications requiring sporadic tasks can implement them by means of periodic tasks, as explained in [Kle93].

4.3 Formalizing the Task Model

One of the main goals of this framework is to provide a feasibility test considering the particular features of the task model. In order to be able to study the feasibility of tasks, their characteristics must first be formally expressed. This section formalizes the model descriptively introduced in the previous section.

A real-time application $\mathcal{A}$ following this model is defined by a finite sequence of *tasks* $\{\tau_1, \tau_2, \ldots, \tau_i, \ldots, \tau_N\}$ plus a set of *dynamic components* $\{\gamma_1^d, \gamma_2^d, \ldots, \gamma_j^d, \ldots, \gamma_M^d\}$. The following three subsections formalize the set of tasks, the set of dynamic components and the model constraints and properties respectively.

4.3.1 Tasks and Components

The application sequence of tasks $\{\tau_1, \tau_2, \ldots, \tau_N\}$ is strictly ordered by following any criteria, which are not constrained by the model, with the priority of each task τ_i being the position i that it occupies in the sequence. In this sequence, a lower number expresses a higher priority. Hereafter, each task will be referred to as either task τ_i or simply task i.

Each task τ_i is defined by a tuple:

$$\tau_i = (T_i,\, D_i,\, O_i,\, C_i,\, B_i,\, H_i,\, M_i,\, \Gamma_i)$$

where:

- T_i is task i's period.
- D_i is task i's deadline, expressed as an interval of time relative to the moments at which the task is periodically released.
- O_i is task i's initial offset, that is, the interval of time from the moment the application starts running until the moment task i is released for the first time.
- C_i is the worst-case execution time for task i. The formula by which it is automatically calculated is shown at the end of the section.
- B_i represents task i's worst-case blocking time. This is the cost of the most time-consuming section of code belonging to lower priority tasks that can interfere (or *block*) task i. Its calculation is derived from the particular synchronization mechanism used by the system. The actual computation method is introduced in Chapter 5, when the framework synchronization mechanism is explained.
- H_i represents the exception handler that the application designer may attach to task i. Handlers are explained (and formalized) in Chapter 6.
- M_i is a finite positive integer number expressing the amount of components forming task i.
- Γ_i represents the sequence of task i's components. This is further examined below.

As is usual in real-time systems, all the timing attributes of tasks (in this case, T_i, D_i, O_i, C_i and B_i), are defined as non-infinite positive integer numbers.

In this model, each application task i can be broken down into a finite *sequence* of *components* or scheduling units. This sequence, denoted by Γ_i, is defined in the following terms:

$$\Gamma_i = \{\gamma_{i1}, \gamma_{i2}, \ldots, \gamma_{ij}, \ldots, \gamma_{iM_i}\}$$

Each component γ_{ij} (or simply component ij) belonging to task i is a tuple of the following form:

$$\gamma_{ij} = (Y_{ij},\, S_{ij},\, M_{ij},\, \Phi_{ij})$$

where:

- Y_{ij} is the component's type, which can be *mandatory*, *unique version*, *successive versions* or *alternative versions*. Please recall that the latter three types correspond to the types of optional components. Formally:

$$Y_{ij} \in \{\mathcal{M},\, \mathcal{UV},\, \mathcal{SV},\, \mathcal{AV}\}$$

- M_{ij} is the amount of *versions* belonging to the component. Versions are computation individuals that can be executed separately.

- Φ_{ij} is the finite set of *versions* belonging to the component. It can be expressed as:

$$\Phi_{ij} = \{c_{ij1}, c_{ij2}, \ldots, c_{ijk}, \ldots, c_{ijM_{ij}}\}$$

  Where c_{ijk} denotes the *worst-case execution time* of the kth version of the jth component of task i. The model enforces this value to be specified for each mandatory component. This value is not required for optional components but, if provided, it is considered as an *estimate* of the version's execution time, which may be useful to know at run time.

- S_{ij} is a real number between zero and one, representing the component ij's *slack fraction*, that is, the portion of task i's total slack that has to be used for scheduling this component. S_{ij} is zero for mandatory components, since they are not scheduled in slack time:

$$\forall \tau_i \in \mathcal{A},\ \forall \gamma_{ij} \in \Gamma_i, \quad \begin{cases} S_{ij} = 0 & Y_{ij} \in \{\mathcal{M}\}, \\ S_{ij} \in [0,1] & Y_{ij} \in \{\mathcal{UV}, \mathcal{SV}, \mathcal{AV}\}. \end{cases}$$

4.3.2 Dynamic Components

The application set of dynamic components $\{\gamma_1^d, \ldots, \gamma_M^d\}$ is defined as a group of components, on which each component γ_j^d is defined as follows:

$$\gamma_j^d = (Y_j^d, D_j^d, M_j^d,\ \Phi_j^d)$$

where:

- Y_j^d is the dynamic component's type which, as for the regular components, can be *mandatory*, *unique version*, *successive versions* or *alternative versions*:

$$Y_j^d \in \{\mathcal{M}, \mathcal{UV}, \mathcal{SV}, \mathcal{AV}\}$$

- D_j^d is the dynamic component's deadline. Its value is only required if the dynamic component is mandatory. Otherwise, this value is not relevant.

- M_j^d is the number of *versions* belonging to the dynamic component. Dynamic mandatory components consists of one version, as it occurs for dynamic optional components of type Unique Version.

- Φ_j^d is the finite set of *versions* belonging to the dynamic component. It can be expressed as:

$$\Phi_j^d = \{c_{j1}^d, c_{j2}^d, \ldots, c_{jk}^d, \ldots, c_{jM_j^d}^d\}$$

  Where c_{jk}^d denotes the *worst-case execution time* of the kth version of the dynamic component γ_j^d. Its value is only required for mandatory dynamic components.

4.3.3 Model Constraints and Properties

All applications $\mathcal{A}$ following the presented formal model have to meet the following constraints:

1) All task deadlines are less than or equal to their periods.

$$\forall \tau_i \in \mathcal{A},\ D_i <= T_i$$

2) The amount of tasks per application (N), the amount of components per task (M_i) and the amount of versions per component (M_{ij}) are positive, implementation-dependent, bounded integer constants.

3) Components of type *mandatory* and *unique version* consist of exactly one version:

$$\forall \tau_i \in \mathcal{A}, \forall \gamma_{ij} \in \Gamma_i \,/\, Y_{ij} \in \{\mathcal{M}, \mathcal{UV}\},\ M_{ij} = 1$$

$$\forall \gamma_j^d \in \mathcal{A}, \,/\, Y_j^d \in \{\mathcal{M}, \mathcal{UV}\},\ M_j^d = 1$$

4) The sum of the slack fraction of all optional components in each task cannot be greater than one:

$$\forall \tau_i \in \mathcal{A}, \quad \sum_{j=1}^{M_i} S_{ij} <= 1$$

5) Mandatory components of tasks, as well as mandatory dynamic components, must have bounded worst-case computation time:

$$\forall \tau_i \in \mathcal{A}, \forall \gamma_{ij} \in \Gamma_i \,/\, Y_{ij} \in \{\mathcal{M}\},\ \exists C \in \mathbb{N} - \{\infty\} \mid c_{ij1} < C$$

$$\forall \gamma_j^d \in \mathcal{A}, \,/\, Y_j^d \in \{\mathcal{M}\},\ \exists C \in \mathbb{N} - \{\infty\} \mid c_{j1}^d < C$$

Considering all the above, in this task model, the *theoretical* worst-case execution time[3] of each task i can be computed by applying the following formula:

$$\forall \tau_i \in \mathcal{A}, \quad C_i = \sum_{\forall \gamma_{ij} \in \Gamma_i \,/\, Y_{ij} \in \{\mathcal{M}\}} c_{ij1} \tag{4.1}$$

In Equation 4.1, the *wcet* of a task i is computed as the sum of the *wcet*s corresponding to the first (and only) version of all i's mandatory components.

[3]The term theoretical here means "without considering any overhead produced by the run-time system".

4.4 The Software Architecture

This section completes the general framework for building FRTS proposed in this thesis by adding a software organization (or *software architecture*) to the task model defined above. In the next two chapters, this basic framework is enhanced with two specific services related to synchronization and exception handling, respectively.

The software organization introduced here, inherited from the ARTIS architecture and then adapted to the new task model, proposes dividing the system in two different, hierarchically related *scheduling levels*. Each level schedules and executes a subset of the application components by utilizing a different scheduling paradigm. This organization is shown in Figure 4.2. In the *real-time level*, depicted in the lower part of the figure, scheduling entities are tasks and running entities are their mandatory components. In the *non-real-time level*, depicted in the upper part, scheduling entities are tasks' optional components, and running entities are their versions.

The real-time level applies a fixed-priority preemptive scheduling policy to tasks. By applying this policy, this level directly runs the mandatory components and releases the optional components, for them to be executed inside the non-real-time level. This policy, which is compatible with the feasibility test, permits guaranteeing the application timing constraints. Also, at the real-time level, the processor spare capacity is reclaimed (by using a sort of slack-stealing algorithm) in order to detect the intervals where the non-real-time level can be safely executed. In such intervals, the non-real-time level applies a utility-based policy to the ready optional components and executes the components' selected versions. This type of policy can take into account aspects such as quality characteristics of tasks, the dynamic situation of the environment and the amount of slack time available at each moment, effectively adapting the application behaviour at run time. The specific policy to be used by the non-real-time level is not imposed by the framework, allowing the designer to choose the best suited policy for each application.

The overall effect of this organization presents a twofold advantage: first, having a different scheduling approach at each level offers a high degree of flexibility at run time while preserving the required guarantee on tasks' timing constraints; and second, the *master/slave* relation between both levels ensures the correct execution of mandatory components independently from the scheduling policy utilized at the non-real-time level. The two scheduling levels are now presented in detail.

4.4.1 The Real-Time Level

In the software architecture, the real-time level contains the hard real-time part of the application. In particular, this is the level where application tasks are created and scheduled, and also where the tasks' mandatory components are executed. These actions are actually performed by the real-time level *first-level scheduler*, which is analogous to the regular run-time scheduler in traditional real-time systems. This section describes the main activities to be carried out by

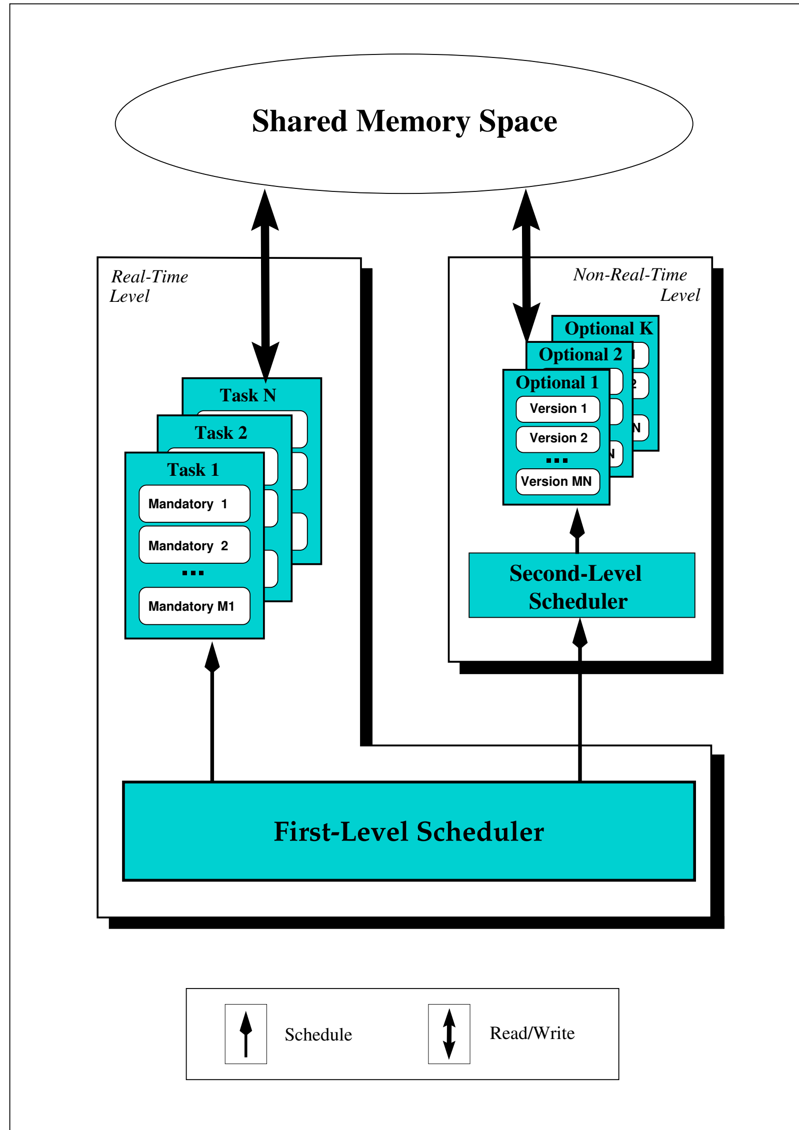


Figure 4.2: The software architecture.

the scheduler, emphasizing its particular version of the fixed priority preemptive scheduling policy.

Inside the real-time level, the first-level scheduler is in charge of four general activities:

a) **Task handling**. The first-level scheduler automatically releases each application task according to its period and its initial offset. Several real-time systems present a contrary approach, in which the application developer is in charge of explicitly releasing periodic tasks.

b) **Slack computation**. The scheduler dynamically keeps track of the slack time available at each priority level. This allows the system to calculate, whenever necessary, the slack available, which is used for executing the non-real-time level. In this abstract activity, the actual slack time algorithm to be used is not relevant, as long as it permits computing the available slack at any moment.

c) **Acceptance of dynamic mandatory components**. Each time a dynamic mandatory component is released, the scheduler computes an acceptance test, in order to determine if it can be safely accepted. Accepted components are executed in the real-time level, since they are mandatory.

d) **Fixed priority preemptive scheduling**. The scheduler applies this policy in order to determine which task and component to execute next. This policy is detailed in the next section.

The Scheduling Policy of the Real-Time Level

This section describes how the fixed priority preemptive scheduling policy can be effectively adapted to the proposed task model and software organization. As it is usual in many schedulers, and for the sake of clarity, the policy is described here in terms of a unique function, the *schedule* function, which is assumed to be invoked at each *scheduling point*[4]. Figure 4.3 presents a C-like pseudo-code of this schedule function. The current section utilizes the schedule function for illustrating the first-level scheduler policy, but it does not discuss the calculations related to the slack stealing algorithm. These calculations are detailed in the following section.

The key aspect considered by the real-time level scheduling policy is that, according to the task model, each application task is actually a strict *sequence* of components. In this sequence, each component can only be scheduled after having scheduled the previous one. As a result, only one component inside each ready task is actually runnable. This component is called the task's *running component*. Therefore, the first-level scheduler applies the following basic policy:

[4] A scheduling point is a time point at which the scheduler is called in order to handle an event that may make the running task change. The *schedule* function is supposed to be called at the end of the handling process, in order to determine the next task to be run.

```
void schedule (void) {
   /* Find the new running task and its running component */
   next_task         = highest_priority_ready_task(list_of_tasks);
   running_component = next_task->running_component;

   if (running_component->type == OPTIONAL) {
       /* Build the list of contiguous optional components */
       optional_list  = VOID;
       slack_fraction = 0.0;
       for (comp = running_component;
            comp != NULL && comp->type == OPTIONAL;
            comp = comp->next_component) {
           list_add(&optional_list, comp);
           slack_fraction += comp->slack_fraction;
       }
       /* Calculate the available slack */
       if (first_time_scheduled(running_component)) {
           running_component->available_slack =
                    calculate_slack(next_task, slack_fraction);
       } /* ...else, this value is assumed to be updated */

       if (running_component->available_slack > 0) {
           /* Calculate the interval of time up to the earliest
              release of a _higher priority_ task */
           time_to_release = next_release(list_of_tasks, next_task)
                             - get_time();

           /* Calculate the interference and the slack interval */
           interference = calculate_interference(next_task, get_time());
           slack_interval = MIN(time_to_release, available_slack);

           /* Send the message and run the non-real-time level */
           message->optional_list  = optional_list;
           message->deadline       = running_component->available_slack
                                     + interference;
           message->slack_interval = slack_interval;

           send_message(NON_REAL_TIME_LEVEL, message);
           execute(NON_REAL_TIME_LEVEL);
           return;
       } else {
          /* No slack => Advance to the next mandatory */
          next_task->running_component = next_mandatory(next_task);
       }
   }
   /* Directly execute the mandatory component */
   execute(next_task->running_component);
}
```

Figure 4.3: Schedule function of the first-level scheduler.

At each scheduling point, the running component of the highest priority runnable task is always selected for execution.

After selecting the next running task and its particular running component (see the first few lines of Figure 4.3), the first-level scheduler distinguishes between two main cases, depending on the type of the running component:

a) If the running component is mandatory, then the big `if` sentence in the schedule function is skipped and the component is directly executed.

b) Otherwise, if the component is optional, the scheduler has to check if there is enough available slack for the component. If so, the scheduler activates the component and runs the non-real-time level, in order to allow the component to be executed. In particular, three slack-related values are calculated by the scheduler:

   - The *slack available* for the component. Intuitively, this value is the component's portion of the slack available for the task the component belongs to. In other words, it is the component's slack fraction calculated over the current slack time available for the task the component belongs to.
   - The *slack interval*. This is defined as the portion of the previous value from the current time up to the next possible timer interrupt for the real-time level. This interrupt can be either the end of the available slack or the next periodic release of a higher priority task. In the latter case, the slack interval is less than the slack available, while in the former case both slack values match. The slack interval is thus the interval of time the non-real-time level will be run without any interrupt from the real-time level.
   - The component *deadline*. This is the last moment the component can be run in this activation. Intuitively, it corresponds to the end of the slack time available for the component, which may be later than the end of the slack interval calculated above[5].

   If the first value (the slack available) is greater than zero, then the two other values are also calculated and the scheduler sends a message to the non-real-time level. This message includes the optional component to be activated, its deadline and the duration of the slack interval. After sending the message, the scheduler runs the non-real-time level for an interval equivalent to the slack interval. While running this interval, the non-real-time level schedules the optional ready optional components according to its own scheduling policy. When the slack interval is exhausted, the non-real-time level is automatically preempted by the first-level scheduler, and the schedule function is invoked again in order to select the next running component.

[5] An optional component may be ready during more than one slack interval, as explained in the next section.

Otherwise, if the available slack for the optional component is zero, then it cannot be activated. In this situation, the scheduler selects the next mandatory component of the same task, if any, for execution.

As explained above, the non-real-time level is executed in the so-called *slack intervals*, which can be seen as intervals of time at which the non-real-time level runs without any interrupt from the real-time level. This interface between both levels allows the non-real-time level scheduling policy to be as autonomous as possible: the level is activated during well-defined intervals of time and, in each interval, it is informed of the length of the interval and which components are ready and when their deadlines expire. This autonomy is a fundamental issue for the non-real-time level, as explained below in Section 4.4.2.

Slack Scheduling

This section explains in detail the calculations related to the slack stealing algorithm incorporated to the first-level scheduler. The particular algorithm has been adapted from one of the strategies which García-Fornes discussed in [Gar96a], called the *strategy 3*. This strategy was shown to feature an appropriate performance for scheduling the task optional parts, providing a reasonable amount of slack time and allowing several optional parts to be ready simultaneously. This is considered a good feature, since it augments the degree of choice for the non-real-time level scheduling policy.

As explained above, each time the running component selected in the schedule function is optional, the component's slack available is calculated. The following present the list of actual calculations performed by the first-level scheduler, as shown in Figure 4.3:

1. First, a list of *contiguous optional components* is built. When several optional components are defined contiguously in the task sequence, all of them are activated at the same time. This is an optimization of the original policy stated above, which enforced scheduling only one component at a time. The optimization is based on two facts. First, the actual scheduling of optional components and their versions is carried out inside the non-real-time level, whose policy is supposed to produce better results when more components are ready simultaneously. Second, this minimizes the overhead related to the optional component scheduling at the real-time level, since the scheduler does not have to be invoked for activating each component in the list.

   The list of contiguous optional components headed by component γ_{ij} is here represented by γ^*_{ij}. This list can be formally defined as:

$$\gamma^*_{ij} = \begin{cases} \emptyset & Y_{ij} \in \{\mathcal{M}\} \\ \{\gamma_{ij}\} \cup \gamma^*_{ij+1} & Y_{ij} \notin \{\mathcal{M}\} \end{cases} \tag{4.2}$$

   On the other hand, since all the optional components in the list are to be activated at the same time, the slack time to be scheduled for them has to incorporate the sum of

their respective slack fractions. The *combined* slack fraction for the list of contiguous optional components headed by component γ_{ij} can be calculated as follows:

$$S^*_{ij} = \sum_{\forall \gamma_{ik} \in \gamma^*_{ij}} S_{ik} \tag{4.3}$$

Please note that, in this and the following equations, the asterisk means that the formula is defined for the list of optional components γ^*_{ij} rather than for component γ_{ij} alone.

2. In this step, the available slack for the list of optional components is calculated. This actually has two cases, depending on whether or not this is the first time that the list of optional components is being scheduled in the current release of its task.

   (a) For each release of a task i, the first time a list of optional components γ^*_{ij}, headed by component ij, is scheduled, their *available slack* is calculated. Intuitively, one may think that this value is calculated by multiplying the available slack for task i by the combined slack fraction S^*_{ij} calculated above. However, as the available slack for i is defined as a function of time and the combined slack fraction is a static value, another value of the slack fraction has to be calculated. This new value, called the *effective slack fraction*, takes into account the slack fraction of optional components of i already scheduled. The following presents the calculation of the effective slack fraction:

   $$ES^*_{ij} = \frac{S^*_{ij}}{1 - \sum_{1 \le k < j} S_{ik}} \tag{4.4}$$

   It can be easily proved that, if component ij is optional, the denominator of this fraction is never zero. The sum factor adds the slack fractions of optional components defined before component ij in the task sequence, which is a value bounded by 1, following property (4) in page 39. Since the sum factor excludes the slack fraction of component ij, this factor never reaches 1.

   Once this value is calculated, the available slack for the optional components in γ^*_{ij} can be calculated as follows:

   $$AS^*_{ij}(t) = \left(\min_{i \le k \le N} S^{max}_k(t) \right) \times ES^*_{ij} \tag{4.5}$$

   In this equation, term $S^{max}_k(t)$ represents the available slack for the priority level k at time t. At this level, the particular slack stealing algorithm to use for calculating this value is not relevant, as long as it may be invoked when required. Furthermore, the value is here assumed to be exact, meaning that if extra factors (such as blocking among tasks) affect the slack calculation, then these factors are assumed to have been taken into account.

(b) Otherwise, if the list γ^*_{ij} has already been scheduled in the current release of its task (that is, the list is now being selected again), then its available slack is not recomputed here. Such a situation happens when (an)other slack interval(s) have previously been scheduled for the list of optional components (see below) in their current activation. If this is the case, $AS^*_{ij}(t)$ is supposed to have been updated after the execution of every slack interval, without recomputing Equation 4.5. This update is done as follows: while the list of optional components γ^*_{ij} remains active, each time a slack interval is scheduled for *any* higher priority task, the value of the interval is subtracted from the components' available slack $AS^*_{ij}(t)$.

3. If there is some slack available then the current *slack interval* value is also calculated. The slack interval is defined as the portion of the available slack value, starting at the current time, up to the next possible time event to be handled by the real-time level. This event can be either the end of the available slack or the next periodic release of a higher priority task k, $x_k(t)$. Formally:

$$SI^*_{ij}(t) = \min\left(\min_{1\leq k<i}(x_k(t) - t),\ AS^*_{ij}(t)\right) \tag{4.6}$$

As explained above, this value indicates the length of the execution interval of the non-real-time level.

4. This fourth step calculates the *deadline* for the list of optional components. This value is required because the available slack for a list of components ($AS^*_{ij}(t)$) may be greater than the slack interval ($SI^*_{ij}(t)$) calculated in step 3. In such cases, the list of components may be active in several following slack intervals. In this situation, the non-real-time level must know when the components' activation has expired. Beyond this expiration time, the components which have not been executed have to be discarded until a future release of their task. The deadline for a list of optional components headed by component ij, $d^*_{ij}(t)$, can be calculated by adding the interference of higher priority tasks to the slack available for the list. Formally:

$$d^*_{ij}(t) = t + AS^*_{ij}(t) + \sum_{1\leq k<i} I_k(t, d_i(t)) \tag{4.7}$$

In this equation, $d_i(t)$ stands for the next absolute deadline of task i. $I_k(t_1, t_2)$ represents the exact interference due to task k between times t_1 and t_2, that is, task k's execution time between t_1 and t_2. The calculation of the interference can be performed by applying the equations calculating the *busy period* in the exact (optimal) version of the dynamic slack stealing algorithm (see [Dav93a] for details).

The method introduced above, based on the concepts of available slack and slack interval, effectively implements García-Fornes' *strategy 3*, with an extra advantage: when the slack scheduled for an optional component is exhausted, the scheduler does not have to recheck the

slack available at lower priority levels, since each active optional component (belonging to different tasks) has its own slack available value ($AS^*_{ij}(t)$) updated, as explained above. On the other hand, the strategy has been extended here to the new task model in which a task may contain several optional components, with each of which having a slack fraction. In fact, the actual slack scheduling policy has refined this by considering that a task actually contains *lists* of contiguous optional components. By following this philosophy, when the first component in each list is scheduled, the list's *combined* slack time is calculated and scheduled for all the components in the list, which are then activated together (see Figure 4.3). This scheme allows more optional components to be simultaneously ready inside the non-real-time level and produces less overhead to the real-time level, since it does not have to be invoked between the activation of consecutive optional components belonging to the same task.

An example

An example is now presented in order to illustrate the concepts of available slack, slack interval and optional component deadline. In the example, a certain execution trace of two tasks τ_1 and τ_2 is shown. The tasks' timing attributes, their internal structure and the run-time situation, which have been chosen for the sake of clarity, are shown in Figure 4.4. For the purposes of this example, the actual type of each optional component is not relevant, and hence it is not specified.

The trace is now discussed by describing what happens at each scheduling point:

- `t = 0`. τ_2 is released and its first, mandatory component γ_{21} is executed.
- `t = 2`. Component γ_{21} ends and the next running component γ_{22} is optional. At that moment, the scheduler builds the list of contiguous optional components (γ_{22} and γ_{23}) and makes the following calculations:

Available slack for τ_2	=	24
Combined slack fraction	=	0.55 + 0.2 = 0.75
Effective slack fraction	=	0.75 / 1.0 = 0.75
Available slack for components	=	24 * 0.75 = 18
Interference	=	4
Deadline of components	=	2 + 18 + 4 = 24
Slack interval	=	min((6-2), 18) = 4

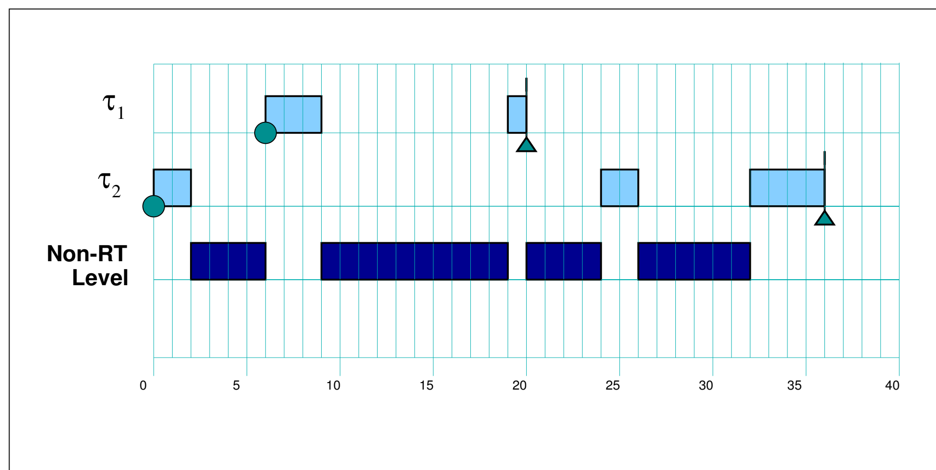
  By applying the slack time algorithm, the available slack for τ_2 at time 2 is 24 time units. Since the combined slack fraction for components γ_{22} and γ_{23} is 0.75, which also matches the effective slack fraction, then the available slack for these components is 18 time units. At time 2, the interference of higher priority tasks in the current release of τ_2 only includes one invocation of τ_1, whose total *wcet* is 4. The deadline for components γ_{22} and γ_{23} is the current time (2), plus their available slack (18), plus the interference (4). Finally, the slack interval is the minimum between the available slack for these

Task	Period	Offset	Deadline
τ_1	100	6	14
τ_2	150	0	36

(a) Timing attributes of tasks.

Task	Component	Type	Slack Fraction	WCET
τ_1	γ_{11}	M	–	3
	γ_{12}	O	0.4	–
	γ_{13}	O	0.6	–
	γ_{14}	M	–	1
τ_2	γ_{21}	M	–	2
	γ_{22}	O	0.55	–
	γ_{23}	O	0.20	–
	γ_{24}	M	–	2
	γ_{25}	O	0.25	–
	γ_{26}	M	–	4

(b) Internal structure of tasks.



(c) Execution trace.

Figure 4.4: Example of slack scheduling

components (18) and the time up to the next release of τ_1. This time is 4 time units, since τ_1 is released at t=6 and the current time is 2.

Therefore, components γ_{22} and γ_{23} are activated until t = 24, and the non-real-time level is executed for 4 time units.

- `t = 6`. Release of τ_1. Its first component γ_{11} is mandatory, and hence it is directly executed.

- `t = 9`. Component γ_{11} ends. As the next component is optional, the scheduler builds the list of optional components (γ_{12} and γ_{13}) and performs the following calculations:

Available slack for τ_1	=	10
Combined slack fraction	=	0.4 + 0.6 = 1.0
Effective slack fraction	=	0.1 / 1.0 = 1.0
Available slack for components	=	10 * 1.0 = 10
Interference	=	0
Deadline of components	=	9 + 10 + 0 = 19
Slack interval	=	min(∞, 10) = 10

The main difference regarding the calculations for τ_2 above is that τ_1 is the highest priority task, and therefore there is no interference from other tasks.

As a result, components γ_{12} and γ_{13} are activated until t = 19, and the non-real-time level is executed for 10 time units. This time, the available slack and the slack interval match, and thus the activated components only have one slack interval in which they may be executed.

- `t = 19`. The available slack expires. The scheduler still has component γ_{12} as the running component but, as there is no available slack for τ_1 the list of optional components is skipped and the following mandatory component (γ_{14}) is selected.

- `t = 20`. Component γ_{14} ends. As it is the last component of τ_1, this task also ends its current release. At that moment, the running component is again γ_{22} (heading the list of components $\gamma_{22}^* = \{\gamma_{22}, \gamma_{23}\}$). Since it is the second time these components are being scheduled, their available slack is not calculated again, but it is assumed to have been updated since it was calculated (in t=2). Therefore, the scheduler just checks if there is still slack available for them. Specifically, the following calculations are made:

Available slack for components	=	4
Interference	=	0
Slack interval	=	min(∞,4) = 4

Since this is the second time in the current release of τ_2 that components γ_{22} and γ_{23} are being selected, the available slack for them is not newly computed, but instead, has

been updated by the scheduler as the previous slack intervals have been scheduled. The remaining available slack for the components is now 4 time units, which is therefore the next slack interval scheduled for them.

- `t = 24`. The available slack expires. Since there is no more slack for components γ_{22} and γ_{23}, the scheduler advances until the next mandatory component of τ_2, γ_{24}, which is selected and executed.

- `t = 26`. Component γ_{24} ends, and the following component within τ_2, γ_{25}, is optional. The list of optional components built by the scheduler is only formed by this component. As this is the first time the component is scheduled, the complete set of calculations is done:

Available slack for τ_2	=	6
Combined slack fraction	=	0.25
Effective slack fraction	=	0.25 / 0.25 = 1.0
Available slack for components	=	6 * 1.0 = 6
Interference	=	0
Deadline of components	=	26 + 6 + 0 = 32
Slack interval	=	min(∞, 6) = 6

  As shown in the calculations, the remaining slack available for τ_2 is scheduled in a single slack interval for component γ_{25}. This case illustrates the purpose of the effective slack fraction: the combined slack fraction for γ_{25} is 0.25, but its available slack time is not 25% of the available slack for τ_2, since γ_{25} is the last optional component within the task. Thus, the effective slack fraction converts the off-line slack fraction provided by the designer into an on-line value, which may be directly multiplied by the slack available for the task in order to calculate the component's slack.

- `t = 32`. The available slack expires, and the last mandatory component γ_{26} belonging to τ_2 is executed.

- `t = 36`. Component γ_{26} ends, making τ_2 end its current release.

The presented example shows how the first-level scheduler applies the fixed priority preemptive scheduling policy in order to select which component of which task to execute next. The main issue here is how the optional components are scheduled by using a particular version of slack algorithm, which permits distributing the available slack among them. It also shows that a list of optional components belonging to a task may be active during several slack intervals. The optional components are then able to be executed in any of these intervals, according to the scheduling policy inside the non-real-time level. In the example, components γ_{12} and γ_{13}, and potentially γ_{22} and γ_{23}, are ready simultaneously in the slack interval between times 9 and 19. The word potentially means that, since γ_{22} and γ_{23} were also ready during a previous slack interval (between times 2 and 6), either of them, or both,

may have been already executed. Therefore, this method of slack scheduling produces a good degree of choice of optional components inside the non-real-time level.

4.4.2 The Non-Real-Time Level

In the proposed software architecture, the non-real-time level is where optional components and their versions are scheduled and where these versions are executed. As explained above, the non-real-time level is always subordinated to the real-time level in such a way that the first-level scheduler activates the task optional components according to the fixed priority policy and decides when to execute and when to preempt the non-real-time level according to its particular slack scheduling. As a result of this method of slack scheduling, the non-real-time level executes during the so-called *slack intervals*. This section describes the non-real-time level, and specially its scheduler, called *second-level scheduler*.

The Second-Level Scheduler

The second-level scheduler is the non-real-time level scheduler, which is in charge of scheduling the task optional components. Although the actual non-real-time level scheduling policy is not enforced by the framework, the general internal structure of the second-level scheduler has to meet the interface between this scheduler and the first-level scheduler. In particular, the framework proposes the second-level scheduler to be internally designed as an endless loop, shown in Figure 4.5, iterating once at each slack interval. The loop is structured in the following four basic steps:

1. At the beginning of each loop, the second-level scheduler retrieves the message from the first-level scheduler indicating the ready optional components (along with their *deadlines*) and the duration of the slack interval. Only new optional components must be added to the list, because the message may contain optional components which were already activated in a previous slack interval.

2. After processing the message, the second-level scheduler looks for *expired* optional components. These are components which were activated in a past slack interval but which have not been executed so far, and whose deadlines have expired. All these components have to be discarded until a future release of the tasks they belong to.

3. Once the scheduler knows which components are ready in the current slack interval, it applies its own scheduling policy in order to find out which components and versions to run in the current interval of slack. This is done inside a loop in which one component/version is chosen and executed per iteration. The loop ends when either the slack interval is about to finish or there is not any other ready component to execute.

4. Just before the slack interval runs out of time, the second-level scheduler voluntarily suspends itself by waiting again for the next message, in order to be in a known and safe state at the following interval.

```
void second_level_scheduler (void) {

   while(TRUE) {
       /* Self-suspension until next message */
       wait_for_message(&message);

       /* Process the message */
       now = get_time();
       end_of_interval = now + message->slack_interval;

       for (comp = list_get_head(message->optional_list);
            comp != VOID;
            comp = comp->next_component) {
           comp->deadline = message->deadline;
           list_add(&ready_comp_list, comp);
       }

       /* Check for expired components */
       for (comp = list_get_head(ready_comp_list);
            comp != NULL;
            comp = comp->next_component) {
           if(comp->deadline <= now) {
               list_remove(&ready_comp_list, comp);
           }
       }

       /* Schedule and execute the ready components */
       components_left = 1;
       while (components_left && now < end_of_interval) {
           /*
              The policy must select the component and the
              version, and _may_ change 'until_time'
           */
           until_time = end_of_interval;
           schedule(ready_comp_list, &comp, &version, &until_time);

           if (comp != VOID) {
               execute_until(comp, version, until_time);
               now = get_time();
           } else {
               components_left = FALSE;
           }
       }
   }
}
```

Figure 4.5: Main loop of the second-level scheduler.

Both the concept of slack interval and the internal structure of the second-level scheduler allow the two schedulers to work in a synchronized fashion, dynamically alternating the control of the processor between them. Thus, although the second-level scheduler is completely subordinated to the first-level scheduler, the former can actually schedule the optional components within the slack intervals just as if the real-time level did not exist, because it knows that it will not be preempted until the end of each interval. Furthermore, the knowledge about the slack intervals is a key requirement of the synchronization mechanisms introduced in Chapter 5. This knowledge also allows the second-level scheduler to safely share data structures with the first-level scheduler, as explained in Section 7.3.

The Non-Real-Time Level's Scheduling Policy

In general, the scheduling policy for the non-real-time level is intended to be some kind of utility-based or *best-effort* policy [Loc86]. However, as expressed above, the framework does not impose any restriction on the actual policy to be used by the second-level scheduler. This decision has been made in order to allow for the maximum flexibility in the framework, thus making the incorporation of several alternative second-level policies in the run-time system possible, and even allowing designers to incorporate their own policies. This is interesting in complex applications, in which scheduling decisions must take into account the current state of the environment and/or some quality aspects of tasks, which may vary from one application to another. This behaviour can thus compensate for the inability of the real-time level to consider these dynamic and quality aspects.

However, the framework does establish that each scheduling decision made by the second-level scheduler (inside the function `schedule` in Figure 4.5) has to at least include the following three decisions:

- The selection of the next *component* to be run, among all the ready optional components, according to the second-level scheduling policy.

- The actual *version* inside this component that must be executed. For SV components, this decision refers to how many versions will be executed, while for AV components, it refers to which version will be executed.

- The *time* up to which the version will be executed, which is bounded by the amount of slack time scheduled for the component. By adjusting this value, the second-level scheduler may decide to give the entire slack interval to the current version, or else to distribute the interval among the runnable components/versions.

By gathering all three decisions in a single function, the framework facilitates the inclusion of custom policies into the non-real-time level and minimizes the changes to be made in the run-time system.

4.5 The FRTL Run-Time System

Flexible Real-Time Linux (FRTL) is a run-time support system which has been specially designed and implemented in order to provide the designer with the framework proposed in this thesis. After having introduced the framework's task model and software architecture, this section now briefly describes the FRTL system. This basic description, which is centered on the functional aspects of this RTOS, is required in order to understand some of the issues discussed in both the next section (presenting the interface functions related to the task model) and the two following chapters (introducing specific services which complete the framework). Nevertheless, an exhaustive description of the FRTL system is presented later in Chapter 7.

FRTL is based on the most basic functionality of the RT-Linux v1 (version 1) system, presented in Section 3.3.2. In particular, it has made use of the following parts: 1) the concept of *rt-task* (or real-time thread running inside the Linux kernel address space) and the related context switching mechanism, 2) the *soft interrupts* mechanism, 3) the access to the hardware timer (in order to consult the current time and to program timer interrupts) and 4) the RT-FIFO facility.

The FRTL support system is divided into two levels, called the *real-time level* and the *Linux level*. These levels implement, respectively, the real-time and non-real-time levels proposed by the software architecture.

- At the real-time level, the original RT-Linux scheduler has been replaced by the first-level scheduler, which creates a modified rt-task for executing each application task. These new rt-tasks are internally designed in order to contain the task's list of components, rather than a single function, and to directly execute the mandatory components in the list.

- The non-real-time level is implemented inside Linux. In particular this level is implemented as a multi-threaded, user-level Linux process called the *optional server* process, running at the Linux highest priority. Within this process, the second-level scheduler creates one thread to run each optional component (or more precisely, each *version*) when necessary and uses the Linux timer and signal mechanisms in order to stop the thread when the version exhausts its given running time. Both schedulers are connected via RT-FIFOs, which provide the required message-passing and synchronization features described in the previous sections.

Applications to be run by the current version of FRTL have to be implemented in C language. Due to the two-level organization, the application code is normally placed in, at least, two different files: one containing the code of the mandatory components and another containing the code of the optional components' versions. The former file is compiled as a kernel module, which is loaded into the kernel address space when the application is started. The latter file is compiled as a regular user module and then linked along with the set of library functions which logically forms the second-level scheduler. Both sides of the application are

```
extern int rt_task_create(RT_TASK *task, RTIME period,
                          RTIME deadline, RTIME offset,
                          RTIME block_time, int priority, int id);

extern int rt_task_add_mandatory(RT_TASK task,
                                void (*func)(void *data),
                                void *data, RTIME wcet);

extern int rt_task_add_optional(RT_TASK task, COMPONENT comp_id,
                               rt_comp_type_t type,
                               float slack_fraction);

extern int rt_task_run_dynamic_mandatory(void (*func)(void *data),
                                        void *data, RTIME wcet,
                                        RTIME deadline);

extern int rt_app_start(void);

extern int rt_app_stop(void);
```

Figure 4.6: Set of interface functions at the real-time level.

run by means of a special *shell script*, that loads the kernel modules and executes the optional server process in the appropriate order. This shell script can also be used to stop the application, if necessary.

4.6 The Set of Interface Functions

This section presents the actual set of FRTL interface functions supporting the creation of tasks and components. Since the application is divided into the real-time and the Linux level, there are two different sets of functions: the creation of the application tasks and the release of mandatory dynamic components are provided by the first-level scheduler, while the creation of optional components and their versions, and the release of optional dynamic components are provided by the second-level scheduler. These two sets of functions are presented in Figure 4.6 and Figure 4.7 respectively.

Each application's real-time side is implemented inside a kernel module which includes invocations to the functions shown in Figure 4.6. For each application task to be created, the `rt_task_create` must be called (including the task's period, deadline, offset, worst-case blocking time and fixed priority). The four timing attributes are expressed in the `RTIME` data type, which is the RT-Linux original time type. In this function, a symbolic identifier (`id`) is also required but only for debugging purposes. After this call is made, a task identifier `task` is returned; this identifier is used afterwards in the following functions for referencing this particular task.

After creating a task, subsequent calls to the functions `rt_task_add_mandatory` and

```
extern int component_create(COMPONENT comp_id,
                            component_attr_t *comp_attr);

extern int component_run_dynamic_optional(COMPONENT comp_id,
                             component_attr_t *comp_attr);

extern void sched_init(void);

extern void app_start(void);
```

Figure 4.7: Set of interface functions at the non-real-time (Linux) level.

`rt_task_add_optional` add mandatory and optional components to the task, respectively. The sequence of invocations of these two functions establishes the actual arrangement of the components inside the task. For a mandatory component, three attributes are required: the function and its initial arguments to be run by the component and the function's worst-case execution time. For an optional component, its identifier, its type (UV, SV or AV) and its slack fraction have to be specified. Once all tasks have been created and all their components added, the application starts running by calling the `rt_app_start`, which activates both the real-time and non-real-time levels. A call to `rt_app_stop` stops running the application and then removes all tasks. Any required dynamic mandatory component can be released by calling the `rt_task_run_dynamic_mandatory` function, specifying the function to be run (along with its initial arguments), and its *wcet* and deadline. The component may be accepted or rejected, depending on the result of the on-line acceptance test applied by the first-level scheduler. This function returns 0 when the component is accepted and a negative value if the component is rejected.

At the non-real-time level, the optional server process has to first initialize the second-level scheduler by calling the `sched_init` function. After that, optional components are created by calling the `component_create` function, indicating the component's unique identifier and a structure containing all its attributes (its type, number of versions, the function to be executed by each version, etc.). The reason for having all of them inside a structure (called `component_attr_t`) rather than specifying them independently, is that it is likely that different application-dependent scheduling policies need different characteristics to be specified for optional components. If so, these new characteristics may be easily included in the structure, leaving the interface functions unchanged. The `app_start` function is invoked when all the optional components have been created, and it basically makes the second-level scheduler enter in its endless loop, described in Section 4.4.2. At run time, if any dynamic optional component is needed, it can be requested for execution by calling the `component_run_dynamic_optional` function, specifying the same parameters than in the creation function. If the application needs to define any extra parameter for these dynamic optional components (such as their deadline, for example) they can also be added to the `component_attr_t` structure.

4.7 Summary and Contributions

The contributions introduced in this chapter are directly related to the advantages that the presented framework provides, compared to other architectures supporting FRTS. In particular, both the proposed task model and software architecture incorporate features which are appropriate for building FRTS. These features are now summarized.

The framework's task model is based on the concept of *component*. This concept helps the designer to break the solution of a problem down into smaller pieces, and to combine them as necessary. In addition, designing software in components favors code reusability, which can be useful even inside the same application. The component concept is used in several aspects of the model:

- Tasks. Each application task is defined as a sequence of components. This allows the application developer to design each task as a sequence of smaller pieces, each one which is in charge of solving a part of the problem that the task is dealing with. The model allows each task component to be defined as either mandatory or optional. Mandatory components are guaranteed to execute, while optional components are given the maximum amount of execution time, in order to maximize their execution possibilities. Thus, the model provides a way of obtaining a guaranteed minimum-quality solution, which is computed by task mandatory components. The model also helps in achieving the best possible quality enhancement at run time, by means of executing as many optional components as possible. Furthermore, the model defines three different types of optional components which can be used depending on the application requirements.

- Dynamic components. The model also provides a way of running some actions occasionally (when the application decides to do so) without implementing such actions as a fixed part of a task. This is done by releasing dynamic components at run time. Dynamic components can be defined as either mandatory or optional. In particular, mandatory dynamic components actually behave as firm tasks: they are accepted or rejected upon arrival, depending on whether or not the system can guarantee their execution.

- Exception handlers. The model allows each task to be attached an extra component, to be called if the task commits a timing exception. This feature is further presented in Chapter 6.

Overall, this task model can be seen as a generalization of previous models for Imprecise Computations. This new model permits the developer to design a task by following any of these models, but it also allows the task to be given a more sophisticated structure. This structure includes several mandatory and optional components arranged without restrictions, different types of optional components, exception handlers, etc. Taking into account all the above, the task model may seem to be too complex for directly designing applications. However, it is worth noting here that the model's principal aim is not to be easy to use, but rather to provide all the features required by FRTS. In this context, a development tool or a special

compiler could help the developer in designing an application using this task model. Specifically, a graphical development tool called InSiDE [Jul00], which is currently being built, will guide designers in the implementation of such applications.

The framework's software architecture proposes separating the execution of mandatory and optional components into two different, hierarchically-related scheduling levels. The *real-time level* schedules tasks and runs their mandatory components, while the *non-real-time level* schedules task optional components and runs the components' versions. Each level follows a different scheduling policy. The real-time level applies a fixed priority preemptive policy; since this scheduling paradigm admits a feasibility analysis, it permits tasks to be guaranteed to meet strict timing constraints. This level is also in charge of extracting the maximum amount of slack time available, in order to offer this time to the non-real-time level. When the non-real-time level is invoked, it applies a sort of utility-based policy to schedule optional components. This policy, which is not imposed by the framework, is intended to incorporate specific knowledge of the problem being solved, quality aspects of tasks and the current situation of the environment into the scheduling of optional components. As a result, the non-real-time level can help the system to achieve a more intelligent and adaptive behaviour.

The advantages of such *combination* of scheduling paradigms can be summarized in three main points. First, the architecture defines a master/slave relationship between both system levels, in such a way that the scheduling actions inside the non-real-time level never affect the schedulability conditions of mandatory components. This property effectively makes the system more robust and easier to analyze. Second, the real-time level is liberated from the significant overhead involved in the utility-based policy applied inside the non-real-time level. And third, the architecture leaves the decision of which non-real-time level policy is best suited for the application to the application designer. Nevertheless, the framework does establish the internal structure of the non-real-time level scheduler and the interface between this and the real-time level scheduler; this provides a safe environment in which the designer is guided in adapting an existing policy or even in developing a new one.

Overall, the architecture presented makes it possible for each combined paradigm to retain its best quality. The real-time level guarantees hard deadlines by means of applying a simple, efficient scheduling policy. The non-real-time level achieves an adaptive and intelligent behaviour by means of a sophisticated policy which can be easily adapted to new requirements, if necessary.

5

Synchronization Facilities

One of the characteristics of the framework presented in the previous chapter is that communication among task components, either mandatory or optional, is accomplished via shared memory. When tasks share memory in a hard real-time system, a synchronization mechanism or protocol is needed in order to protect both the consistency of the data being accessed and the application tasks' hard deadlines. In the context of FRTS, the problem with traditional protocols is that they typically only address the case of hard tasks accessing the shared data. This chapter introduces and evaluates two synchronization protocols developed for the presented framework. The two protocols are actually special versions of the Priority Ceiling Protocol and the Ceiling Semaphore Protocol, which have been extended in order to allow both mandatory and optional components to share memory. The chapter discusses how the two protocols can be successfully integrated in both the real-time and non-real-time levels, emphasizing the interactions between them and the slack stealing algorithm utilized at the real-time level. In relation to these interactions, some weak points of the existing literature have been detected and then addressed. An exhaustive evaluation of the two adapted protocols is presented at the end of the chapter. The evaluation is made by comparing their respective theoretical properties as well as the actual overhead they produce to the FRTL system. The overall result is that the Ceiling Semaphore Protocol is far more appropriate than the Priority Ceiling Protocol in a real-time framework where hard and soft tasks share memory.

5.1 Introduction

When applications are internally designed as a set of cooperating tasks, as occurs in real-time systems, having some means of communication among tasks is often required. In such systems, the ability of tasks to share memory results in an appropriate communication mechanism, and, therefore, becomes a common requirement in applications of this kind. When tasks share memory in non-real-time systems (and even in some soft real-time systems), a synchronization mechanism (or *synchronization protocol*) is needed in order to protect the consistency of the data being accessed by tasks; that is, to avoid *race conditions* [Sil94]. In hard real-time systems, though, synchronization must *also* guarantee task timing constraints. In the fixed priority preemptive scheduling paradigm, a set of synchronization protocols has been developed in order to guarantee both data consistency and hard deadlines. This set of algorithms includes the Ceiling Semaphore Protocol (CSP) [Raj89], the Basic Priority Inheritance Protocol (PIP) [Sha90] and the Priority Ceiling Protocol (PCP) [Sha90]. All these protocols were originally designed for synchronizing *only* hard real-time tasks.

As stated in Section 2.6, a common characteristic in FRTS is to combine tasks with different criticality levels into the same system. However, typical research studies in FRTS do not address the issue of communication among tasks of different criticality, but define soft tasks (or even all types of tasks) to be independent. Exception of this are the *Open System* architecture [Den97], which proposes the Non-Preemptable Critical Section (NPS) protocol as a mechanism for safely sharing global data, and the HARTIK operating system [Lam97], in which the Stack Resource Protocol [Bak91] has been adapted in order to allow both hard and soft tasks to share memory.

The synchronization solutions proposed in this chapter have been developed by revising some of the well-established protocols in the literature and adapting them to the task model and software architecture presented in Chapter 4, specially to the slack stealing algorithm. Furthermore, in order to achieve predictability *and* efficiency in the run-time system, some characteristics about the actual FRTL system have also been taken into account. In particular, the following requirements and constraints have been considered:

1) The solution has to keep the key advantage of RT-Linux, which is the complete isolation between the real-time tasks and the Linux system, in the sense that no action taken in the Linux level can affect (i.e., cause a delay to) the RT-Linux level. This is perfectly achieved by the *soft interrupts* mechanism (described in Section 3.3.2). Therefore, a protocol like the NPS cannot be directly implemented in RT-Linux, since it is impossible to make a section of code within a Linux process non-preemptable.

2) Context switches *from* an optional component *to* a mandatory component for synchronization purposes should be avoided. The first reason is that, in order to produce this kind of context switch, the *optional server* process has to make a direct system call to the first-level scheduler, and there is no fully predictable mechanism for making that call.

The second reason is that this feature would greatly complicate the interface between the two schedulers, with no real gain in performance in most of the cases. Consider that an optional component γ_{ij}, belonging to task τ_i, is denied entry to one of its critical sections and because of this another (mandatory) component resumes execution. In this situation, component γ_{ij} will only be able to execute again if there is another (future) slack interval available for it *and* if the second-level scheduler chooses it for execution again. This is a combined situation with low probability. Therefore, it is not worth the complicated interface and the overhead involved in supporting this feature.

3) The synchronization solution has to be as efficient as possible in order to minimize the kernel overhead and to maximize the amount of available slack for the optional components. Thus, some simplifications has been made, such as the amount of blocking information available at run time: in complex real-time applications, with a large number of mutexes and possibly nested critical sections, it is not reasonable for the scheduler to keep track of the particular critical section each task could be executing (and the remaining execution time of each section), given only the lock and unlock requests made by tasks. This knowledge imposes a degree of overhead in the system that is not worth the accuracy it brings to the slack stealing algorithm (this is further explained in Section 5.3). For this reason, the only blocking information available at run-time is the worst-case blocking factor of each task, exactly the same value needed by the feasibility test (denoted by B_i in Section 4.3).

4) The solution has to be compatible with both the fixed priority preemptive scheduling followed at the real-time level and the scheduling policy which is provided at the non-real-time level. In other words, the synchronization mechanism has to be independent from the non-real-time level scheduling policy.

The restrictions and properties stated above as well as the analysis of the existing synchronization protocols have led to developing a variation of the CSP and PCP which could be appropriate for the task model, software architecture and implementation features presented in the previous chapter. Both protocols are presented and discussed in this chapter. The PIP was not considered for two reasons. First, compared with the PCP, the PIP provides less synchronization features, since it does not prevent deadlocks or chained blocking. And second, the PIP behaviour is poorer than the PCP behaviour when these protocols are combined with the slack time algorithm. In particular, it has been theoretically proven that the PIP produces higher worst-case blocking factors than the PCP (this is shown in [Sha90]), which leads to have less slack available at run time. Overall, the PCP presents all the advantages of the PIP, and exhibits better theoretical properties and run-time behaviour.

The rest of the chapter is organized as follows: Section 5.2 presents significant related work and some limitations which have been found in it. Sections 5.3 and 5.4 respectively present the incorporation of the PCP and CSP as valid synchronization mechanisms to the framework proposed in this thesis. Section 5.5 introduces the set of synchronization functions provided in the FRTL system to the application components. Section 5.6 evaluates the two

adapted protocols from a theoretical point of view, and also presents some empirical results, showing a significant difference in performance between them. Section 5.7 summarizes and discusses the contributions of this chapter.

5.2 Related work

5.2.1 Priority Ceiling Protocol

In the context of operating systems, the term *interference* describes a situation in which a runnable task has to wait to get the processor because another task is currently running. In fixed priority preemptive systems without synchronization, a task can only suffer interference from higher priority tasks. However, if tasks synchronize their access to shared resources, it can actually occur that a higher priority ready task i has to wait because a lower priority task j holds a resource which task i is requesting; this effectively makes task i suffer interference from the lower priority task. In such scenarios, generically called *priority inversion* situations, the higher priority task i is said to be *blocked* by (or to suffer blocking from)the lower priority task j. A valid synchronization protocol for a hard real-time system has to bound the maximum priority inversion time for each task i, hence allowing for the calculation of the task's worst-case blocking factor (or B_i). This factor can then be introduced in the formula testing the task feasibility.

The work in [Sha90] describes the PCP as a valid synchronization protocol for uniprocessor hard real-time systems using fixed priority preemptive scheduling and synchronizing tasks by means of binary semaphores or *mutexes*. Hereafter, the term semaphore will always refer to binary semaphore and thus both terms (mutex and semaphore) will be used indistinctly.

The PCP is presented below. Although the following are not the original terms by which the PCP was described in [Sha90], they are nearer to a valid PCP's implementation, and probably describe its behaviour more clearly:

1) The PCP sometimes makes a task execute at a priority level different from its original priority level. Therefore, for each task, the protocol distinguishes between its *base* priority (that is, its original fixed priority) and its *current* priority, which can vary as the task executes. Each time a task is periodically released, its current priority is made equal to its base priority. The run-time scheduler always chooses the ready task with the highest *current* priority for execution.

2) A *priority ceiling* (or simply ceiling) is defined for each semaphore. This ceiling, calculated off-line, is defined as the base priority of the highest priority task which uses the semaphore.

3) When a task i requests to *lock* a mutex S, the system checks the *system ceiling*, defined as the ceiling of the highest priority semaphore locked by any *other* task. This semaphore is denoted by S^*. If the current priority of i is strictly higher than the ceiling of S^*, then task i is allowed to lock S and it enters the critical section.

Otherwise, if i's priority is not higher than S^* ceiling, task i is then *blocked* by the lower priority task j holding the lock of S^*; that is, i is blocked in S^*. Then, the current priority of j is raised up to the priority of i (i.e. j *inherits* the priority of i), and hence j resumes execution.

4) When a task i requests to *unlock* a mutex S, the highest priority task j blocked in S, if any, is awakened, and S is unlocked. Task j is then allowed to lock the semaphore it requested before, which may be S or another one. Besides, the current priority of i is reevaluated, being set to the current priority of the highest priority task still blocked by i. If there is no task blocked by i, then i retrieves its base priority.

It is demonstrated in [Sha90] that, by following the behaviour above, chained blocking[1] and deadlock[2] situations are naturally avoided. Furthermore, the worst-case blocking time (B_i) of each task τ_i is confined to the longest critical section within a lower priority task whose guarding mutex has a ceiling which is higher than or equal to i (that is, the priority of τ_i).

$$B_i = \max_{i<j\leq N}(z_{jk} \,/\, ceiling(s_{jk}) \leq i) \tag{5.1}$$

This is formally expressed in Equation 5.1, where z_{kj} represents the cost of the k-th critical section in task j and s_{kj} is the guarding semaphore of that critical section. In this equation the identifier of each task corresponds to its priority, with a lower number representing a higher priority.

5.2.2 Ceiling Semaphore Protocol

The Ceiling Semaphore Protocol, also known as the Highest Locker protocol, or the Ceiling Locking protocol in the Ada community, is the simplest of the priority inheritance protocols. Its behaviour can be summarized in the following terms:

1) As explained for the PCP, each task in the CSP has a fixed base priority and a dynamic current priority. At run time, each time the task is released, its current priority is made equal to its base priority. The scheduler always chooses the ready task with the highest current priority for execution.

2) The ceiling of each semaphore is calculated off-line. This ceiling is defined as the base priority of the highest priority task using the semaphore. The CSP requires this ceiling calculation to be revised recursively when there are nested critical sections. In particular, when a critical section is defined inside another critical section, the ceiling

[1] Chained blocking occurs when a task i, in a given release, has to wait for *several* semaphores, each one held by a different lower priority task. This produces potentially large blocking times to tasks.

[2] A deadlock is defined for a set of tasks as a run-time situation in which each task in the set is waiting for a resource which is held by another task in the set. When a deadlock occurs, the entire task set is blocked without any possibility of evolution.

of the semaphore guarding the inner section is recalculated as the maximum between its ceiling and the ceiling of the semaphore guarding the outer section. If a semaphore ceiling is changed because of this rule, the rest of semaphore ceilings also have to be reevaluated, which may produce other ceilings to change, and so on.

3) When a task i requests to *lock* a semaphore S, the current priority of the task is immediately raised to (or *inherits*) the ceiling of this semaphore.

4) When a task i requests to *unlock* a semaphore S, the current priority of the task is set to the priority it had when it requested to lock S.

The computation of each task's worst-case blocking factor (B_i) under the CSP can be performed in the same terms as under the PCP (expressed in Equation 5.1). Furthermore, the theoretical blocking properties of both protocols are equivalent (the CSP also avoids chain blocking and deadlock situations).

However, the CSP exhibits some particular run-time properties which are worth noting:

- Under the CSP, whenever a task requests to lock a semaphore, the lock is *always* granted. Therefore, a locking request cannot produce a suspension of the calling task or a context switch. Another implication is that the queue of each semaphore always remains empty. In fact, what the protocol does is to dynamically adjust task priorities to achieve synchronization, rather than actually using semaphores in the traditional sense.

- A task i can only be *blocked* in one situation: when it is released while a lower base-priority task j is running at a priority higher than i's priority, because it has locked a semaphore. In this situation, the blocking of task i remains until task j unlocks the semaphore(s) that it locked previously and recovers a previous priority which is lower than i's priority. Once this potential blocking is over, task i cannot be blocked again during its current release. This property, which can be referred to as *early blocking*, is shown later in this chapter to be a a very interesting feature.

- The CSP lock and unlock operations are more efficient (i.e., they produce less overhead) than the corresponding PCP operations. In particular, the PCP lock operation has a cost of $O(n)$, with n being the number of mutexes, because it must search for the highest priority locked mutex; the PCP unlock operation also has a cost of $O(n)$ because it has to recompute the system ceiling, searching again for the highest priority locked mutex. Both lock and unlock operations have a cost of $O(1)$ under the CSP.

The first and second properties clearly show that the CSP usually produces a lower number of context switches than the PCP, having the same run-time situation. Overall, the CSP is a much simpler and more efficient algorithm than the PCP, having the same theoretical properties and producing the same worst-case blocking time for the application tasks.

5.2.3 Slack Stealing Algorithm and the Priority Ceiling Protocol

In [Dav93a], it is shown how a system using the Dynamic Slack Stealing (DSS) algorithm to execute soft tasks can also use PCP mutexes to synchronize hard and soft tasks. This work also states that exactly the same approach holds if the CSP is used instead of the PCP. This section presents a brief summary of that work and describes some problems detected in it.

The DSS algorithm computes the slack time in a per-priority basis, with each task being assigned a different priority. The slack time available at time t at each priority level i is denoted by $S_i^{max}(t)$. This value is calculated, over a certain interval of time, by subtracting the interference due to higher priority tasks from that interval. However, as described in Section 5.2.1, when tasks share resources, a task can also suffer interference from a lower priority task (which is called blocking in this case). Protocols such as the PCP bound that blocking by calculating the worst-case blocking factor, or B_i for each task. Therefore, since interference of tasks are worsened by blocking, the slack stealing algorithm must also subtract the blocking factor from the slack available, in order for all tasks to finish by their deadlines.

However, always subtracting B_i from the slack available at the i priority level is a pessimistic approach, since the blocking situations of tasks do change at run time, not always being at their worst case. For this reason, the *current blocking factor*, denoted by $b_i(t)$ is introduced. This factor is bounded by B_i, but it can be dynamically updated by applying the following rules:

1) Each time task i ends its current release the factor $b_i(t)$ is set to B_i.

2) When task i is released again, this value can be reduced. First, the system ceiling (the ceiling of S^*, according to the definition of the previous section) is checked. If the priority of task i is strictly higher than the current system ceiling, then $b_i(t)$ is set to zero. Otherwise, the task j holding S^* is checked. If the base priority of task j is higher than the priority of task i, then $b_i(t)$ is set to zero. Otherwise, $b_i(t)$ is set to $z_j(t)$, which is the remaining execution time of the critical section currently being executed by task j.

3) Each interval of time (from t^1 to t^2) that a task j is blocking another task i, (and hence, task j is running at a raised priority), the blocking factor $b_i(t)$ can be reduced in: $b_i(t^2) = b_i(t^1) - (t^2 - t^1)$.

When blocking among hard tasks is present, factor $b_i(t)$ has to be introduced in the slack stealing main equation which calculates the slack time available at time t for running soft tasks ($S^{max}(t)$). This is presented in Equation 5.2. In this equation, k represents the highest priority ready task and $lp(k)$ is the set of tasks with priority lower than or equal to k.

$$S^{max}(t) = \min_{\forall i \in lp(k)} (S_i^{max}(t) - b_i(t)) \tag{5.2}$$

In systems where PCP mutexes are only used by hard tasks, the DSS algorithm must include the modifications presented above, but the PCP itself is not changed. However, if soft

tasks are also allowed to use mutexes, the PCP must take into account the slack time available, in order to prevent the execution of a soft task critical section from jeopardizing the timing constraints of any hard task. It is stated in [Dav93a] that when a soft task tries to lock a mutex with ceiling j, the lock can only be granted if the task has enough slack time to execute its critical section in preference over all hard tasks with priority j or less. Equation 5.3 shows this test, with c being the cost of the critical section in the soft task.

$$\min_{\forall i \in lp(j)} (S_i^{max}(t) - b_i(t)) > c \tag{5.3}$$

5.2.4 Problems Detected in this Work

The condition stated in Equation 5.3 has a significant problem: it does not consider the case in which the mutex has already been locked. Under the PCP, when a hard task requests a lock, the test used to grant or deny the request simply checks the priority of the task against the system ceiling. The protocol itself ensures that this test is enough to avoid granting the lock to a mutex previously locked by another task. However, in Equation 5.3 neither the system ceiling nor the priority of the soft task are considered. In fact, the priority of the soft task is not relevant here because the DSS makes the soft task execute at the priority level of the highest priority ready task, which can be any task depending on the moment at which the soft task arrives to the system. Thus, it can actually occur that when a soft task requests a mutex lock it verifies Equation 5.3 but the mutex has already been locked by a (lower priority) hard task. The proposed test is therefore not sufficient to safely grant the lock request, because although it does guarantee the timing constraints of tasks, it does not ensure the consistency of shared data. One possible solution to this problem is to include the current mutex's internal *status* (i.e. whether it is locked or not) in the condition used to grant lock requests made by soft tasks. Another solution would be to set the ceilings of the semaphores that may be used by soft tasks to the highest priority, which on the other hand would increase the blocking time of hard tasks.

It is also worth noting that in the presented work by Davis, nothing is stated about what happens to the soft task if it is *not* allowed to lock the mutex, which is not a trivial problem. When the system denies a mutex lock to a soft task, it has two possibilities: the first one is to abort (or kill) the soft task. The second one is to preempt and suspend the task and insert it into the mutex queue (or in another, special queue), until a future moment at which the request can be granted. This second possibility obviously leads to much more complicated interactions between the DSS and the PCP, which would have to be carefully evaluated. Furthermore, the second alternative is not valid for this architecture, since it violates the second restriction expressed in Section 5.1: it would provoke a context switch from the soft task which has been denied the mutex lock to the hard task holding the lock of that mutex.

5.3 Integration of the Priority Ceiling Protocol

Since the software architecture proposed in the previous chapter establishes two different levels of computation, PCP mutexes have to be offered at each application level, by means of the first-level scheduler and the second-level scheduler. Nevertheless, the mechanism must be coherently designed as a single synchronization facility for all components.

5.3.1 The PCP at the Real-Time Level

Taking into account that at the real-time level, scheduling entities are tasks and running entities are their mandatory components, the first level scheduler includes this slightly modified version of the PCP:

1) A base priority and a current priority is defined for each application task. Whenever a task is periodically released, its current priority is made equal to its base priority. The scheduler always chooses the running component belonging to the ready task with the highest current priority for execution. On the other hand, the definition of both priorities for each task is naturally extended to all the components of the task.

2) The ceiling of each semaphore, calculated off-line, is defined as the base priority of the highest priority task using the semaphore. In this case, a task is said to use a semaphore when any of its *mandatory* components uses it.

3) When a mandatory running component γ_{ik} belonging to a task i requests to *lock* a mutex S, the system checks the ceiling of the highest ceiling locked semaphore, S^*. If the current priority of i is strictly higher than the ceiling of S^*, γ_{ik} is allowed to lock S and it enters the critical section.

   Otherwise, if i priority is not higher than S^* ceiling, then γ_{ik} (and hence the whole task i) is *blocked* by the mandatory running component γ_{jl} (belonging to lower priority task j) which holds the lock of S^*. In this latter case, the current priority of j is raised up to the priority of i and hence γ_{jl} resumes execution, while i remains blocked in S^*.

4) When a mandatory running component γ_{ik} belonging to a task i requests to *unlock* a mutex S, the highest priority task j blocked in S, if any, is awakened, and S is unlocked. Then, the running component γ_{jl} of task j is allowed to lock the semaphore it requested before, which can be S or another one. Also, the i current priority is reevaluated, being set to the current priority of the highest priority task still blocked by i, if any, or else to i base priority, if there is no such task.

The worst-case blocking time (B_i) can be computed by using Equation 5.1, constraining the critical sections to be examined to only those belonging to *mandatory* components of tasks.

5.3.2 The PCP and the Slack-Stealing Algorithm

According to the theory presented in Section 5.2.3, the slack algorithm used by the first-level scheduler must introduce the computation of the current blocking factor ($b_i(t)$) at each priority level i. However, this computation cannot be performed exactly as it was introduced since, as stated in Section 5.1, the scheduler does not know the cost of each critical section (it only knows the worst-case blocking factor B_i for each task). As a result, the second rule presented above for calculating each task $b_i(t)$ factor is reformulated in the following terms:

> *"When task i is released again, if its priority is strictly higher than the system ceiling, then $b_i(t)$ is set to zero. Otherwise, it is not changed".*

Compared to the original rule, the two cases in which task i is not strictly higher than the system ceiling have been removed here, for different reasons. The first case (when the task j holding S^* has a higher base priority than i) is not considered because the first-level scheduler never releases a task while a higher priority task is running; thus, this situation cannot occur. The second case (when j has a priority level lower than i) has been removed because the first-level scheduler does not keep track of factor $z_j(t)$ for each task j. As a result, $z_j(t)$ is here approximated by B_i. This adds pessimism to the computation of the available slack, leading to a potentially reduced slack-time value for each task. Nevertheless, this pessimism can be bounded: for each task i, the difference between the available-slack value presented in Section 5.2.3 and the one here is limited by $\max_{\forall j \in lp(i)}(B_j)$.

Therefore, the first-level scheduler dynamically keeps track of these *approximated* blocking factors and introduces them in the formula calculating the available slack for optional components. This formula (previously presented in Equation 4.5, page 46) is thus reformulated as follows:

$$AS^*_{ij}(t) = \left(\min_{i \leq k \leq N} (S^{max}_k(t) - b_k(t)) \right) \times ES^*_{ij} \tag{5.4}$$

5.3.3 The PCP at the Non-Real-Time Level

Access to mutexes at the non-real-time level is provided to optional components by the second-level scheduler. The algorithm used by the scheduler must consider the theory and limitations shown in Section 5.2.3, as well as the constraints in Section 5.1, in order to implement a valid synchronization mechanism in this system.

In particular, the PCP *lock* operation for optional components is set to feature the following behaviour:

1) As occurs for mandatory components, when an optional component wants to enter one of its critical sections, it must first obtain the lock of the semaphore guarding that section.

2) As stated in Equation 5.3, the second-level scheduler needs to know the duration of that critical section. Optional components are hence forced to specify both the semaphore they want to lock S and the duration c of the critical section they want to enter at each lock request operation.

3) When an optional component actually makes a lock request, $lock(S, c)$, the second-level scheduler applies the following test:

   "if the requested mutex S is not *locked and the duration c of the critical section is less than or equal to the remaining time of the current slack interval, then the lock is granted; otherwise, the lock is denied"*.

   This test can be performed very efficiently (with a cost of $O(1)$).

4) When a lock request is granted, the optional component is allowed to enter its critical section, but the state of the semaphore itself is *not* changed. This is because since the critical section will be finished by the end of the current slack interval, any mandatory component that subsequently requests to lock that semaphore will find it unlocked anyway.

5) When a lock request is denied the optional component is killed.

The PCP *unlock* operation for optional components is void, because the state of the semaphore is not changed in the lock operation, as explained in the rule 4 above.

5.4 Integration of the Ceiling Semaphore Protocol

The last section has presented a valid PCP-based protocol for synchronizing mandatory and optional components in this architecture, with the term *valid* meaning that hard tasks are not jeopardized and shared resources are consistently accessed. However, that protocol is somewhat restrictive: an optional component might be denied the lock (and therefore killed) even in the case of having enough slack time to atomically finish its critical section, due to the fact that another lower priority mandatory component may hold the lock on the corresponding semaphore. The CSP-based protocol presented in this section reduces the probability of this situation to a very rare case.

The CSP has been introduced in the system in a similar fashion as explained above for the PCP. In fact, the FRTL run-time system supports both protocols, and the actual protocol to be used by a particular application is set at the initialization stage.

5.4.1 The CSP at the Real-Time Level

The first-level scheduler implements the regular CSP lock and unlock operations, in order to offer them to the mandatory parts. The protocol's behaviour has been changed a little, in order

to take into consideration that mandatory components instead of tasks run at the real-time level:

1) In the CSP, each task has a fixed base and a dynamic current priority. Each time the task is released, its current priority is made equal to its base priority. The scheduler always chooses the running component belonging to the ready task with the highest current priority for execution.

2) The ceiling of each semaphore, calculated off-line, is defined as the base priority of the highest priority task using the semaphore. In this case, a task is said to use a semaphore when *any* of its components, either mandatory or optional, uses it.

3) When the running component γ_{ij} belonging to task i requests to lock a semaphore S, the current priority of task i is immediately raised to the ceiling of this semaphore.

4) When the running component γ_{ij} of a task i requests to unlock a semaphore S, the current priority of the task is set to the priority it had when it requested to lock S.

The worst-case blocking time (B_i) can be computed here by using Equation 5.1, but considering only the critical sections defined inside the *mandatory* components of tasks.

5.4.2 The CSP and the Slack-Stealing Algorithm

The slack-stealing algorithm is adapted here in the same terms as was explained for the PCP, with the only difference being the computation of the dynamic blocking factors ($b_i(t)$). In this case, the first and third original rules also hold, and the second one is reformulated as follows:

> *"On each release of a task i, when it starts running for the first time after being released, $b_i(t)$ is set to 0. This value will be valid until the task ends its current release".*

This computation of $b_i(t)$ is also approximate for the same reason as for the PCP: B_i is used as an approximation of $z_j(t)$, when task j blocks task i. The difference here is that, due to the early blocking, whenever a task i starts running, its blocking factor can be safely set to zero. Then, when Equation 5.2 is used to know the slack time available for running an optional component, the computed slack at its priority level will be exact. Therefore, task i can only receive a reduced available-slack value if lower priority tasks are being blocked at that moment. Under the CSP the worst-case reduction in the slack value is $\max_{\forall j \in lp(i) \sim i}(B_j)$.

It is worth noting that, on one hand, the reduction above is potentially smaller than its equivalent under the PCP, since B_i is *not* included. On the other hand, the probability for a task to receive a pessimistic available-slack value is also smaller here, because this only occurs when lower priority tasks are being blocked when a task has to execute one of its optional components.

The first-level scheduler dynamically keeps track of these blocking factors. Whenever an optional component needs its available slack to be calculated, the same equation defined for the PCP (Equation 5.4) is used to do so. However, the amount of slack time here is potentially greater than it is for the PCP, since the computation of the dynamic blocking factors for the CSP is more accurate (less pessimistic) as shown above.

5.4.3 The CSP at the Non-Real-Time Level

The behaviour of the CSP lock and unlock operations provided by the second-level scheduler to the optional components is analogous to the behaviour explained above for the PCP at the non-real-time level. The difference in this case is that, due to the early blocking, the probability of an optional component finding a mutex which is already locked is reduced here to a unique and very rare case, explained below. The test applied by the second-level scheduler on a CSP lock request made by an optional component can be expressed in the following terms:

> *"if the requested mutex S is* not *locked and the duration c of the critical section is less than or equal to the remaining time of the current slack interval, then the lock is granted; otherwise, the lock is denied"*.

Here, again, when the lock is denied to an optional component, the component is killed. However, this is less behaviour restrictive *in practice* than the one explained for the PCP, since the semaphore will be unlocked at almost every lock request. Therefore, in practical terms, the lock will be granted if the optional component has enough slack time to atomically finish the critical section.

The CSP unlock operation at this level is void (as explained in the PCP) since the second-level scheduler does not change the mutex status at a successful lock request made by an optional component.

The CSP anomaly

The case in which the semaphore is already locked when an optional component tries to lock it is now discussed. In the original terms of the CSP, a task always finds mutexes unlocked, due to the *early blocking* property discussed above. This property applies perfectly to the real-time level in the framework's CSP version, since this level strictly follows a fixed priority preemptive scheduling. Thus, no mandatory component would ever find a CSP mutex locked. However, this is not true at the non-real-time level. Due to the way slack intervals are scheduled for optional components, an *anomaly* can occur in the CSP. This anomaly is presented in Figure 5.1.

The anomaly is based on the fact that optional components can be scheduled during several (consecutive) slack intervals. When this occurs, it is probable that optional components belonging to several tasks are ready simultaneously, leading to a situation in which an optional component is selected for execution inside a slack interval which was scheduled for optional

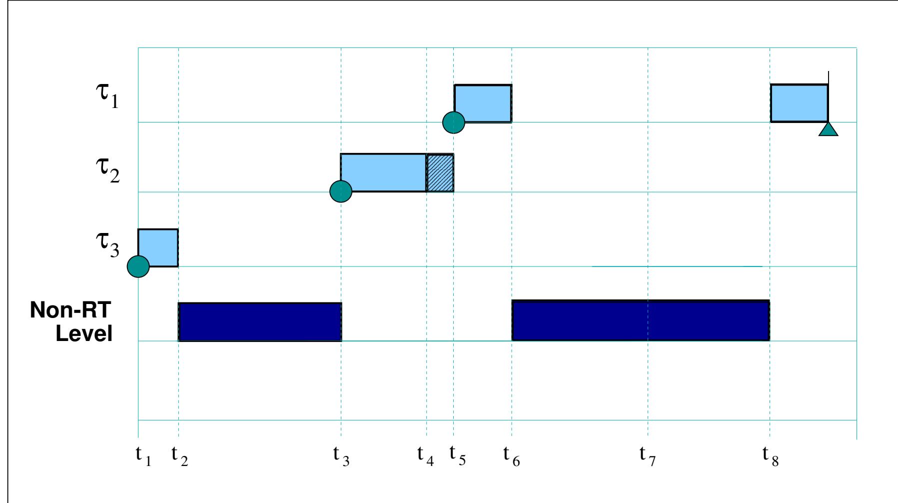


Figure 5.1: CSP anomaly at the non-real-time level.

components belonging to *another* task. Consider the case in the figure, on which three tasks, τ_1, τ_2 and τ_3, are executed. Task τ_3 starts running first (at t=t_1), and its first component γ_{31} is mandatory. At t=t_2, this task starts running a set of optional components (γ^*_{32}) during an initial slack interval $[t_2, t_3]$. At t=t_3, task τ_2 is released. The first component of this task γ_{21} is mandatory and, at t=t_4 it requests to lock a semaphore S, shared with task τ_3. As the ceiling of S is 2 (task τ_1 does not use it), the τ_2 keeps running at priority 2. At t=t_5, while γ_{21} is executing the critical section guarded by S (shaded box in the figure), task τ_1 is released and starts running its first mandatory component γ_{11}. At t=t_6, τ_1 schedules a slack interval for its optional component(s) γ^*_{12} up to t=t_8, when the slack available for γ^*_{12} expires. Precisely at this slack interval is where the anomaly *may* occur: imagine a time t_7 inside the slack interval $[t_6, t_8]$ in which an optional component belonging to γ^*_{32} requests to lock S and finds the mutex locked.

Thus, the general conditions for the anomaly to occur are:

- A list of optional components γ^*_{Lk} belonging to a lower priority task τ_L are released and scheduled during one or more slack intervals in which not all of the components are executed.
- While the list γ^*_{Lk} remains active, a mandatory component γ_{Mj} belonging to a medium priority task τ_M runs and locks a semaphore S, which is shared with some components in task τ_L.
- While component γ_{Mj} is running the critical section guarded by S, a higher priority task τ_H *which does not use* S is released.
- A slack interval is scheduled for an optional component belonging to τ_H. During this

slack interval, a component belonging to γ^*_{Lk}, which has not yet been executed, is selected for running.

- This component belonging to γ^*_{Lk} requests to lock S.

The long list of conditions gives an idea of the low probability of this situation happening, specially taking into account that critical sections inside mandatory components can be reasonably considered to be short in time. Thus, there is not a great likelihood of the medium priority task being preempted by the release of the higher priority task in the middle of such a critical section.

In any case, this anomaly could be prevented if every mutex in the system were assigned a ceiling of 1. In many applications, this solution obviously leads to a potentially greater amount of blocking time for every application task. The designer thus has to balance between the schedulability conditions for mandatory components versus the run-time scheduling conditions for optional components.

5.5 Interface Functions

The PCP and CSP algorithms have been introduced to the FRTL system by following the indications explained in sections 5.3 and 5.4, respectively. Figure 5.2 presents the actual set of synchronization functions available to application components within the FRTL system. These functions allow all the application components to synchronize their access to shared memory by locking and unlocking PCP or CSP mutexes. This set (like the set of task-related functions in Section 4.6) is divided into two subsets: in Figure 5.2, the functions with the `rt` prefix are provided to mandatory components by the first-level scheduler, while the rest of them are provided to optional components by the second-level scheduler.

```
int  rt_mutex_set_pcp_mode(void);
int  rt_mutex_set_csp_mode(void);

int  rt_mutex_create(RT_MUTEX mutex, int ceiling);
int  rt_mutex_destroy(RT_MUTEX mutex);
int  rt_mutex_lock(RT_MUTEX mutex);
int  rt_mutex_unlock(RT_MUTEX mutex);

int  mutex_lock(RT_MUTEX mutex, systime_t time);
int  mutex_unlock(RT_MUTEX mutex);
```

Figure 5.2: Set of synchronization functions.

The protocol to be used by a particular application is decided inside the real-time level, at the initialization stage, by means of calling one of these two functions: `rt_mutex_set_pcp_mode`

or `rt_mutex_set_csp_mode`. This design decision has been made in order to allow the rest of the interface functions to be independent from the protocol used. It is therefore possible to decide which protocol to use just by calling the appropriate initialization function, without any change in the rest of functions (basically, lock and unlock) invoked within components. Obviously, it is not possible to change the protocol while the application is running.

Mutexes are also created at initialization time (and at the real-time level) by invoking the `rt_mutex_create`. Once a mutex has been created, it can be accessed at run-time from both levels. In order to permit this access in a straightforward manner, the identifier of each mutex is specified (not returned) in the `rt_mutex_create` function. That is, mutex identifiers are decided at design time and thus, the particular mutex to be used by each lock or unlock operation (inside the code of mandatory and optional components) can be directly determined. The function `rt_mutex_destroy` is used to delete a mutex, at the end of the application execution.

At run time, mutexes are accessed by means of the two sets of lock and unlock operations shown in the figure, with each set available for one type of components. Functions `rt_mutex_lock` and `rt_mutex_unlock` provide the traditional PCP or CSP primitives to mandatory components. At the non-real-time level, the `mutex_lock` function needs to include both the identifier `mutex` of the mutex to be locked and the duration `time` of the critical section the optional component wants to execute. This, as explained in the previous section, is required because both values are needed by the second-level scheduler in the two protocols. The `mutex_unlock` function for both algorithms is void, because the lock operation at the non-real-time level never changes the internal status of the mutex (see Section 5.3.3). This function is nevertheless provided for completeness.

5.6 Evaluation of the Two Protocols

This section reviews first the theoretical properties of the special PCP and CSP versions introduced in this chapter, specially discussing their similarities and differences. Then, it shows some experimental run-time results that basically reveal significant differences in performance between them.

5.6.1 High-Level Evaluation

The two synchronization solutions presented above maintain the requirements imposed in Section 5.1, so, in this sense, both are valid solutions for this architecture. In particular, the *isolation* requirement still holds because, although both the real-time and Linux subsystems do share memory and synchronize their access to that memory, no action executed in the Linux side can ever cause a delay to the real-time side.

Both algorithms may be considered to be rather simplistic at the non-real-time level, due to design decisions such as killing an optional component when it is denied a mutex lock. Such simplistic behaviour is justified if one takes into account two aspects. First, the framework

design has tried to minimize the interactions between the first-level and second-level scheduler, in order to give the second-level scheduler policy a great deal of autonomy. And second, the framework intends to provide a basic synchronization mechanism which can be valid to any scheduling policy at the non-real-time level. A sophisticated scheduling policy could, for example, use the information of which semaphores are locked in order not to run an optional component using one of these semaphores.

From the designer point of view these solutions have two main advantages. First, both PCP and CSP are well known and widely used in the real-time community. And second, the way mutexes are offered makes their use fairly simple at the design and implementation stages of the application. Both aspects are now presented.

At design time, the necessary number of mutexes is determined in the usual way (for PCP and CSP), depending on the pieces of data to be shared and the different tasks accessing them. This must be done by considering that a task is using a piece of data whenever *any* of its components accesses it. On the other hand, calculating each mutex ceiling is also easily carried out, just by checking which components (for the CSP) or which mandatory components (for the PCP) use each mutex and the priority of the tasks the components belong to.

At implementation time, the use of mutexes is also quite simple: since the system designer has to identify each mutex at design time and this identifier is specified (*not* returned) when the `rt_mutex_create` function is called, then this mutex can be easily used for implementing every lock call and unlock call on both sides of the application.

Thus, the two protocols have been proven to be valid for the framework presented in the previous chapter and, for this reason, both are supported by the FRTL run-time system (so that the application designer can choose between them). However, from a theoretical point of view, the CSP has proven to be far more appropriate than the PCP. In general, the CSP is a simpler algorithm featuring less overhead and the same blocking properties, which is in itself reason enough to be considered better than the PCP. Furthermore, the CSP has proven to fit better the particular requirements imposed by this framework, in two main respects. First, given the same (limited) run-time information about blocking, the CSP provides more accurate dynamic blocking factors, which produces more slack time available for optional components. And second, the condition to grant or deny the lock to a mutex at the non-real-time level is less restrictive (in practice) in the CSP; this leads to a higher probability of success in locking the application mutexes by optional components. Both benefits are mainly due to the CSP's *early blocking*, a feature shown to be especially appropriate in a system where hard and soft tasks must share memory. As a curiosity, it is interesting to note that Lamastra et al. [Lam97] reached a similar conclusion in their operating system, HARTIK, which also mixes hard and soft tasks but uses a different scheduling paradigm (Earliest Deadline First) and a different synchronization protocol (Stack Resource Protocol).

LOCK					
Protocol	**# mutex**	**Min.**	**Max.**	**Avg.**	**Var.**
PCP	4	6	15	9.7050	1.7389
	8	6	18	9.6122	2.6009
	16	6	22	10.7260	6.9184
CSP	4	5	10	8.4673	0.3323
	8	5	11	8.2139	0.3699
	16	5	10	8.2210	0.3637

UNLOCK					
Protocol	**# mutex**	**Min.**	**Max.**	**Avg.**	**Var.**
PCP	4	6	13	9.1353	0.5643
	8	6	13	8.9780	0.4852
	16	6	14	9.5054	0.7727
CSP	4	5	11	8.3006	0.3919
	8	5	11	8.0882	0.3768
	16	5	11	8.0687	0.3567

Table 5.1: Measured values of the cost of lock and unlock operations.

5.6.2 Run-Time Evaluation

This section introduces some experimental results for the two protocols implemented in the FRTL system. Please note that the purpose of this section is only to compare the two protocols. A complete description of the measurement mechanisms is introduced later in Section 7.4 and exhaustive measurements of all the sources of kernel overhead are presented in Section 8.5.

The experiments were performed by running several sample applications on a PC with an AMD K6-2 333 Mhz. processor, with the FRTL system running above RedHat Linux 5.2 (kernel 2.0.36). Data were collected at run time and stored in a piece of the shared memory by the first-level scheduler, a mechanism used for measuring the scheduler overhead and for debugging purposes (this mechanism is described in Chapter 7). The amount of physical memory available constrained the collected data to about 90,000 events per execution, which normally corresponds to approximately one minute of execution time. The set of sample applications was implemented with from 2 to 8 tasks and from 4 to 16 mutexes. Task periods were chosen, within 20 and 200 milliseconds, in such a way that the *hyperperiod* was less than 60 seconds, which ensures that the results about the available slack time for each task are consistent. All the measurements presented here are expressed in *ticks*, which is the regular time unit in RT-Linux, corresponding to 0.838 microseconds.

The measured costs of the lock and unlock operations for applications using 4, 8 and 16 mutexes are summarized in Table 5.1, which shows the maximum, minimum, average and variance values. The results show how the cost of both lock and unlock operations are greater (in average and in worst case) for the PCP than for the CSP, and also how the PCP lock operation becomes clearly costlier as the number of mutexes increases.

Another interesting feature which was tested was the impact caused by each protocol on the slack time computation. That is, which protocol produced a lower reduction in the slack time available. It must be noted that there is not one *best* application in which to test this feature, since each task's timing characteristics, the number of mutexes, the actual usage of these mutexes (the moments at which lock and unlock operations are called, the length of the critical sections, etc.) affect the results. However, the experiments showed that the CSP generally leaves more slack time available to tasks than the PCP does. The actual detected gain in slack time, though significant, is small in absolute terms because the CSP better performance is due to better accuracy in the computation of $b_i(t)$, which normally is a small number compared with the regular intervals of slack time. Finally, another property detected was that the percentage of slack time available to higher priority tasks was normally greater under the CSP. This is also due to the better accuracy of the $b_i(t)$ factor, which produces more slack time available at all priority levels. The actual increase is greater in higher priority levels because of the behaviour of the slack algorithm itself: the amount of slack time not available to (or not used by) higher priority tasks can be reclaimed by lower priority tasks.

5.7 Summary and Contributions

This chapter presents a variation of two well-known protocols (the PCP and the CSP) which have been proven to be valid synchronization protocols for mandatory and optional components, in the context of the proposed framework and run-time system. The design of the synchronization solutions has pointed out some limitations in the existing literature (related to the combination of the PCP and the slack stealing algorithm), which have been addressed.

The second main issue discussed in this chapter is to exhaustively compare the two protocols, in order to determine which one better fits the requirements of the framework. The evaluation of the protocols has stated that their different run-time behaviour leads to a significant difference in performance between them, which can be summarized in two major points:

1) The CSP is a much simpler and more efficient algorithm. Both its lock and unlock operations have a cost of $O(1)$, while the corresponding PCP's operations have a cost of $O(N)$, with N being the number of application mutexes. The CSP clearly produces less overhead at run time, also taking into account that one of its theoretical properties is that it produces fewer context switches than the PCP.

2) The CSP has an interesting property that might be called *early blocking*. This refers to the fact that a task can only be blocked from the moment it is released to the moment it starts execution, and also that a lock request made by a task is *always* granted. This early blocking feature has a twofold advantage: first, it leads to smaller $b_i(t)$ factors, because when a task starts execution its factor can be safely set to zero until the end of its release; and having smaller $b_i(t)$ factors leads to more available slack.

   The second advantage is that this feature produces a less restrictive lock test because the protocol ensures that each mutex will be always unlocked when a task tries to lock

it. In this framework, this general property is maintained at the real-time level, and, therefore, every mandatory component always finds mutexes unlocked. At the non-real-time level, this property holds in the regular case, but there is one exception: the way optional components are scheduled in this framework produces an anomaly in the CSP at the non-real-time level. In this anomaly, an optional component may find a mutex locked when it attempts to lock it. However, this chapter has characterized the anomaly as a very rare and improbable situation. As a result, in the regular case, the only reason by which an optional component is denied a mutex lock is because it does not have enough slack time to complete the critical section.

Overall, although the PCP and CSP present similar theoretical properties (they both prevent chain blocking and deadlock situations) and both originate the same worst-case blocking factors, the CSP is clearly a much more appropriate protocol for this architecture. The CSP algorithm exhibits more efficient behaviour, with fewer context switches and less overhead at each lock and unlock operation. Furthermore, the CSP algorithm also produces better scheduling possibilities for the optional components, providing both more available slack time (specially for higher priority tasks) and a higher probability of successfully locking application mutexes.

6

Timing Exception Support

As explained at the end of Chapter 2, one of the key requirements of FRTS is to have fault tolerance capabilities. In real-time systems, one of the key issues related to fault tolerance is to make the system able to deal with timing exceptions produced by tasks. A *timing exception* or fault occurs when an application task fails to meet its timing requirements (that is, its deadline) at run time. Hard real-time systems avoid timing exceptions by design and then verify that design by means of a feasibility analysis. Both the design and the analysis are based on the system following a specific, well-defined run-time behaviour. Among the rules defining this behaviour, the most sensitive is the one ensuring that each application task never utilizes more processor time than its worst-case execution time. Because of this, a situation in which a task executes beyond its *wcet* can also be considered a timing exception in itself. This chapter introduces specific mechanisms dealing with timing exceptions produced at run time. Although both *wcet* and deadline exceptions are considered, the proposal here is that by only controlling *wcet* exceptions (that is, by never allowing an application task to execute for more time than its *wcet*) it can be guaranteed that no task will miss its deadline. Overall, the timing exception support mechanisms proposed in this chapter have three main goals: first, the mechanisms have to be completely integrated in the framework presented in previous chapters in order to take advantage of all the framework flexibility. Second, the mechanisms have to be efficiently designed and implemented in the FRTL run-time system. And third, the overhead related to these mechanisms has to be introduced into the system complete feasibility test. The present chapter describes the design of these mechanisms in detail and introduces their related FRTL interface functions. The mechanism implementation details and their impact on

the feasibility test are presented, respectively, in Chapter 7 and Chapter 8.

6.1 Introduction

Hard real-time systems require a strict guarantee on the fulfillment of their timing requirements, which in practice means proving that the deadlines of their tasks are always met at run time. In the context of fixed priority preemptive systems, this guarantee is obtained by means of a feasibility test that mathematically proves the ability of each task to meet its deadline under worst-case load conditions. This test is based on a set of hypotheses (or constraints) which are related to the task model and to the run-time behaviour of the system. Although several of these constraints have been progressively removed or relaxed, as the scheduling theory has evolved (see Section 2.4), some of them still hold. The following list summarizes the constraints usually imposed by the feasibility test:

1) The set of hard tasks is fixed and known.

2) All hard tasks are either periodic or sporadic (the sporadic having a known minimum inter-arrival rate).

3) Each task is assigned a fixed priority.

4) At run-time, tasks are scheduled according to their fixed priorities, in a preemptive manner. That is, the system always executes the runnable task with the highest priority.

5) A hard task cannot voluntarily relinquish the processor nor delay itself once it is selected for execution.

6) The maximum requirement of processor time is bounded and known *a priori* for each hard task. This time is named the task's worst-case execution time (or *wcet*). The *wcet* value which is introduced to the feasibility test for each task cannot be exceeded at run time. This also applies to the run-time support system: the system maximum overhead must also be known in order to be introduced to the test.

The former four constraints are straightforward, because they are enforced by either the task model ((1) to (3)) or the run-time scheduler ((4)). Conversely, the two latter constraints depend on how the designer has implemented the application tasks, and, therefore, they represent the most sensitive aspect of the feasibility analysis. Constraint (5) may be easily met, by avoiding the arbitrary use of delay sentences[1] and the access to hardware devices in such a way that the accessing task is temporarily suspended until the device completes the request.

The last constraint is the most difficult to achieve, since it relies on the designer to be able to accurately measure the *wcet* of each task. The existing techniques for calculating task *wcet*s

[1] In many real-time systems, such as applications developed in Ada, delay-like sentences are used to suspend the execution of a task at the end of each release, until the beginning of the following one. Clearly, this use of a delay sentence does not violate the constraint (5) above.

can be divided into two broad categories: analytical and empirical. Analytical techniques analyze the binary code of each task in order to calculate its execution time. Unfortunately, this is a very complex problem, where existing methods cannot be applied to all possible hardware platforms; furthermore, some advanced features of new generation processors (such as caching or pipelining) produce significant variations in the processing speed, making the problem still harder to solve. Even the trivial approach of disabling the processor cache when calculating execution times does not ensure the worst case since, as shown in [Lun99], in certain circumstances, a cache miss may result in a shorter execution time than a cache hit. The empirical approach is based on executing each task thousands of times on the particular hardware the final application will run on and measuring each execution time. Then, the task *wcet* is set as the longest measured time plus a certain safety factor. This approach, which is the most frequently used in the industry, is not completely safe since tasks cannot normally be measured under every possible data entry or run-time condition.

The issue of accurately calculating task *wcet*s is thus a serious problem in real-time systems. This is even more difficult in FRTS, since these systems may have to deal with complex problems and unpredictable environments. In such conditions, application tasks will normally be complex (i.e., their binary code will be difficult to analyze) and run-time conditions may be quite different from the testbed conditions in which task *wcet*s can be measured. As a result, since a framework for building FRTS cannot be completely sure of the accuracy of *wcet* estimates, the system reliability would be improved if specific mechanisms dealing with run-time *timing exceptions* were introduced. In a real-time system the obvious timing exception which has to be avoided is the *missed deadline*. However, as presented above, the main reason for which a task can miss its deadline is that this task or another one exceeds its *wcet*. Thus, the *wcet* of each task can also be considered a timing requirement, with its violation being considered a timing exception.

Technically speaking, a run-time exception is defined as a system event associated with some error or violation (e.g., a division-by-zero operation, an illegal memory access, etc.) executed by an application task. Traditionally, operating systems *detect* these exceptions (with some help from the hardware) and define a treatment or *handling*[2] for each of them. This chapter presents specific mechanisms for detecting and handling *wcet* and deadline timing exceptions in the context of the framework for FRTS proposed in this thesis.

The rest of the chapter is organized as follows: Section 6.2 reviews some significant previous work on this issue. Section 6.3 discusses the requirements and design principles imposed over the developed support mechanisms. This support is then described in detail within Sections 6.4 and 6.5. Section 6.6 presents the set of FRTL interface functions related to timing exceptions. Finally, Section 6.7 summarizes the ideas introduced within the chapter and evaluates its contributions.

[2]The handling mechanism typically includes a notification of the exception to the faulty task (which may then execute particular handling actions), or even the abortion of the task, if the error is unrecoverable.

6.2 Previous Work

The most relevant work on timing exception handling for hard real-time systems is probably the set of mechanisms developed by Stewart and Khosla [Ste96], which have been implemented within the Chimera II operating system [Ste92]. The set of run-time mechanisms includes the detection and handling of timing exceptions (both *wcet* and deadline exceptions), and an automatic task profiling facility. These mechanisms are now described.

The detection of run-time timing exceptions is performed differently depending on the type of exception, but both are based on the fact that the kernel is *tick driven*[3]. To detect deadline exceptions, a kernel virtual timer is dynamically set to the earliest deadline; if that programmed time is reached by the current system time, then the scheduler is notified about the task(s) that have missed their deadlines. To detect *wcet* exceptions, a software counter is defined for each task. Whenever a task is released, its counter is set to the task's *wcet*; at each clock tick, the running task's counter is decremented; if the counter of the running task ever reaches zero, then the scheduler is notified.

When the scheduler is notified about a timing exception, it automatically invokes a task-specific, user-defined *timing failure handler* (TFH). Handlers are thus developed by the application designer and associated to the application tasks. A TFH can include all kinds of recovery actions, such as aborting the failing task, sending messages to other tasks, activating an alarm, etc. TFHs have two special features: first, a TFH may be defined to be executed at a different priority than the priority of the task it belongs to; and second, invocation of handlers may be temporarily disabled when, for example, the running task is executing a critical section.

The task monitoring is performed by means of the *automatic task profiling* (ATP) software mechanism, which collects dynamic timing information about tasks at each context switch. For each task, the collected data include the number of releases, the execution time for the current release, the number of timing exceptions, etc. This information is immediately made available to soft tasks, for further processing, via shared memory.

Overall, these timing exception and profiling mechanisms are actually simple, low-level, policy-independent mechanisms, which have been designed from the viewpoint of the RTOS, rather than by considering any particular task model or scheduling policy. The positive effect of designing such mechanisms as a fundamental part of the kernel is that they are accurate and can be efficiently implemented, therefore producing very little overhead. However, the support that the mechanisms provide to the designer is poor, including only the possibility of calling a function (the TFH) in the context of the failing task. No other specific services (such as killing, restarting or resuming the task, or other tasks), are provided; if needed, all these actions have to be entirely implemented by the application programmer. On the other hand, as the proposed mechanisms are independent from any particular scheduling scheme, the work by Stewart and Khosla does not address any kind of feasibility analysis.

[3] In a tick driven kernel, the system time is measured in fixed-length intervals or *ticks*, periodically signaled by means of a hardware timer interrupt.

6.3 Design Principles

The timing exception support mechanisms introduced here have been designed in order to achieve two main goals. First, the mechanisms have to be fully integrated within the framework proposed in this thesis; this means that they must be flexible, consistent with the task model, and predictable in order to be introduced into the feasibility test. And second, the mechanisms must be efficiently designed and implemented in the FRTL run-time system, in order to produce a reasonably low overhead.

Two different cases have been considered in the design of the exception mechanisms. This classification depends on the expected timing behaviour of the run-time system or kernel. Please note that, although the classification has been made in the context of the FRTL system, it can be applied to most real-time kernels:

a) The first case occurs when the kernel overhead can be assumed to be accurately measured; that is, when the worst-case overhead is perfectly calculated and introduced into the feasibility test. In this case, any application which is guaranteed by the feasibility test maintains this guarantee at run-time as long as no application task consumes more execution time than its *wcet*.

b) The second case occurs when the assumption above cannot be made, that is, when the kernel produces more overhead than expected. Because of this excessive overhead, tasks can lose their deadlines even if no application task violates its *wcet* restriction,

In the first case, the run-time system can ensure meeting the deadlines of feasible tasks only by detecting *wcet* timing exceptions. This is a direct consequence of the schedulability test: if the run-time system never allows a task to execute beyond its *wcet* then the task cannot jeopardize any task's deadline. As a result, the timing exception support proposes including both a detection mechanism for *wcet* faults and a flexible handling mechanism. The handling mechanism includes the automatic and immediate termination of the faulty task and, optionally, the execution of a user-defined handler, which is intended to contain application-specific actions that have to be executed after the exception. This support for detecting and handling *wcet* exceptions is detailed in Section 6.4.

In the second case, in which the kernel itself may exceed its expected overhead, the support for *wcet* exceptions proposed above is not sufficient to ensure that task deadlines are not violated. In this case, an explicit mechanism for detecting missed deadlines has to be introduced. Obviously, this can only occur if the kernel has not been sufficiently tested and thus its overhead measurements are not accurate. Such a situation happens, for example, while the kernel is being ported to a different hardware or when the kernel is not working properly. For this reason, the timing exception support proposes an optional detection mechanism for missed deadlines, which can be activated if the application decides to do so. This optional mechanism should actually be considered as a tool for the FRTL development and tuning up phases, rather than a mechanism for the application developer. This support is described in Section 6.5.

6.4 Support for *wcet* Exceptions

The framework support dealing with task *wcet* exceptions is actually formed by two separate activities: *detection* and *handling* of the exceptions. Both activities are provided at the system real-time level, because this is the level affected by timing exceptions. In particular, the fault detection as well as some common handling actions are directly established by the framework. These fixed handling actions basically enforce the immediate abortion of the current release of the faulty task. In order to compensate for this fixed behaviour, the framework also proposes providing the designer with the ability of implementing a custom exception handler for each task, which is invoked when that task produces a *wcet* exception. Following the framework philosophy, the handling mechanism allows the designer to define handlers as either mandatory or optional *components*, which can be attached to the application tasks. Furthermore, the importance (or urgency) of a particular task's handler is considered here to depend on the task's characteristics, rather than on its priority; for this reason, the priority of a handler can be different from the priority of its associated task.

The framework offers a highly flexible exception handling support to the designer, which is fully integrated with the task model presented in Section 4.2. The following describes the support for defining application handlers:

- A different handler may be implemented for (and attached to) each application task. However, this is only an option: different tasks may be attached to the same handler, and a task may not have any handler.
- Each handler can be defined as either mandatory or optional. *Mandatory* handlers are considered as hard tasks for scheduling purposes, while *optional* handlers are executed in slack time. Please note that being mandatory or optional is only a *scheduling* type, rather than an execution type: despite their type, handlers are always executed at the real-time level. This permits them to execute in the context of the faulty task[4]. If a handler is defined as mandatory, its *wcet* has to be specified, in order to be considered as interference at the appropriate priority level.
- In either of these two types, each handler is given a specific, fixed priority, which may be different from the priority of its related task. At run time, having a ready handler and a ready task with the same priority, the first-level scheduler chooses the handler first. This provides the designer with the ability to execute a handler at the highest possible priority in the system (ensuring that it may be executed immediately and without interference), without having to explicitly reserve the highest priority level for such handlers. This semantic has to be taken into account when assigning priorities to tasks and handlers.

The rest of the section first details the detection and handling mechanisms, and then extends the framework formal task model presented in Section 4.3 in order to include formal definition of the user-defined *wcet* handlers.

[4]A *wcet* exception can only be produced while an application task is running a mandatory component, at the real-time level.

6.4.1 Detection of *wcet* Exceptions

A *wcet* timing exception occurs when a hard task consumes more execution time than the value expressed as its *wcet* in the feasibility test. Since hard tasks in this framework are composed of mandatory and optional components, this definition has to be somewhat refined.

In the task model introduced in Section 4.2, there are actually two different entities with a known *wcet*: mandatory components and tasks. On the one hand, the designer is forced to provide the *wcet* of each mandatory component; on the other hand, each task's *total wcet* is automatically calculated by adding up the *wcet*s of its mandatory components. Thus, a *wcet* exception can be detected at either the mandatory component level or the task level. These two alternatives are now discussed.

In the first approach, a *wcet* exception is detected for a task as soon as one of its mandatory components exceeds its *wcet*. This approach is safe, in the sense that the system detects the fault as soon as the minimum-size entity with a known *wcet* violates its computation restriction. However, this alternative is not optimal because the feasibility test guarantees the computation time at the task level, meaning that if a given task does not consume more execution time than its *total wcet*, then the schedulability of lower priority tasks is not jeopardized. In other words, if some mandatory components inside task i consume more than their *wcet*s, but this is compensated by other i's mandatory components consuming less than their *wcet*s (in such a way that the task as a whole does not exceed its total *wcet*), then the system is safe. Thus, this second approach (in which *wcet* exceptions are detected at the task level) is less pessimistic.

For this reason, the framework proposes considering a *wcet* exception only when a task consumes more than its total *wcet*. Nevertheless, mandatory components exceeding their maximum computation times have to be detected (even if not producing an actual exception) in order to be able to correct their *wcet* estimations. Considering all this, the framework mechanism for detecting *wcet* exceptions is stated by means of the following rules:

1) In absence of previous timing errors, the system is said to be in *no-fault mode* for each application task i. In this mode, when a mandatory component γ_{ij} consumes less execution time than its *wcet*, c_{ij1}, the remaining time is considered as *gain time*. The concept of gain time (introduced within the context of slack stealing algorithms (see [Dav93a])) is defined as the portion of a hard task's guaranteed *wcet* which the task does not consume in a particular release. When the task ends each release, any gain time available for the task can be reclaimed as slack time for this task's priority level and all lower priority levels. The framework here proposes to reclaim the available gain time at the end of each mandatory component, rather than at the end of the task release.

2) A situation at which a mandatory component γ_{ij} consumes more than its *wcet*, but task i does not exceed its *total wcet*, is considered as a *local fault*. Local faults do not produce an interrupt and they are not considered as *wcet* exceptions. After component γ_{ij} has produced a local fault, the system enters into the so-called *compensation mode* for task i and stores the fault as part of the *profiling information* that the system records

at run time. The profiling mechanism is a part of a general-purpose debugging mechanism defined for the FRTL system, by which the run-time information produced at the real-time level is stored in a piece of main memory; this information can be retrieved afterwards by a Linux process in order to be appropriately presented to the designer. This mechanism is described in Section 7.4.

In this particular case, the profiling information stored for the local fault includes the moment at which the fault has occurred, the identifier of the faulty component and the actual execution time consumed by the component. By later analyzing this information, the designer can detect which mandatory components are not keeping their computation constraints, the moments at which the faults occurred and how much execution time these components actually consumed.

3) While the system is in compensation mode for a task i, the gain time produced by its subsequent mandatory components, if any, is used for *compensating* the previous execution excess, rather than being reclaimed as slack time. If the excess can be completely compensated, the system returns to the no-fault mode for i, expressed by rule (1).

   The system knows when local faults have been compensated by keeping track of the *execution left* for each ready task i. This value is set to the i's total *wcet* when task i is released and it is decremented as mandatory components of i are executed. Thus, the system knows that any previous fault has been compensated when, at the end of a mandatory component γ_{ij}, the execution left of task i matches with the sum of the *wcet*s of i's mandatory components following component γ_{ij}.

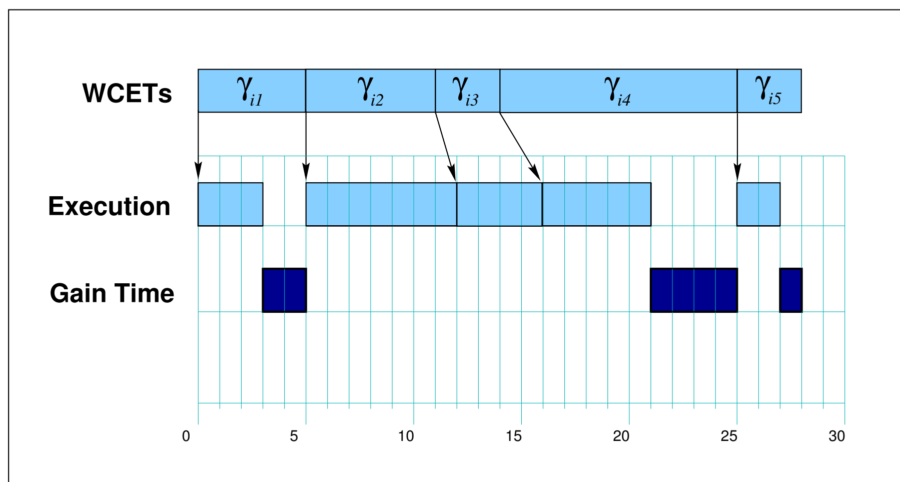
4) If, while running any of its mandatory components, a task exhausts its total *wcet*, then the task is *immediately* interrupted by the system and a *wcet* exception is raised. At this moment, the handling mechanism described below in Section 6.4.2 is invoked.

Figure 6.1 presents an example of this mechanism, illustrating some of the situations that may occur during the execution of the mandatory components inside a task i. Section (a) in the figure presents a table showing the characteristics of the task components, including the *wcet* of each mandatory component forming the task. Section (b) shows a possible logical execution trace. Please note that this figure is not an actual execution chronogram, but a representation of the actual execution time consumed by mandatory components and the available gain time at the end of each component. In this execution diagram, the upper part depicts the mandatory components as contiguous boxes, with their widths being their respective *wcet*s. The lower part of the diagram shows the actual execution time consumed by each component. Arrows between both parts point out the difference between the expected (logical) starting time of each component and the real time. For the sake of clarity, the example task does not contain any optional component.

When task i is released, its *execution time left* is made equal to the task's total wcet, 28 time units. Then, the execution of each component is as follows:

Component	Type	Wcet	Slack Fraction
γ_{i1}	M	5	–
γ_{i2}	M	6	–
γ_{i3}	M	3	–
γ_{i4}	M	11	–
γ_{i5}	M	3	–

(a) Internal structure of the example task i.



(b) A possible execution of task i, illustrating the actual execution time consumed by each mandatory component and the gain time produced.

Figure 6.1: Example of the compensation mechanism.

- Component γ_{i1}. This component has a *wcet* of 5 time units, but it consumes only 3 of them. Thus, at the end of its execution, there is a gain time of 2 time units (this time is added to the slack time available at all priority levels equal to or lower than i). The execution time left of task i at this point is set to 23 time units.

- Component γ_{i2}. The second component executes for 7 time units, that is, one time unit beyond its *wcet*. Since the task has not exceeded its total *wcet*, this situation is detected by the system at the end of γ_{i2}. The system stores the local fault as profiling information and enters in compensation mode for the task. The execution time left of task i is set to 16 time units (that is, one time unit lower than the theoretical remaining *wcet* of the task).

- Component γ_{i3}. The third component also executes for one time unit more than its *wcet*, which is 3 units, producing another local fault for the task. As for the previous component, this local fault does not produce an actual *wcet* exception since the task's *wcet* has not been violated (the task's execution time left has not reached zero). Nevertheless, the previous excess of one time unit produced by component γ_{i2} has been incremented in another time unit due to this fault. As a result, the execution time left of task i is set to 12 time units (that is, two time units lower than the theoretical remaining *wcet* of the task).

- Component γ_{i4}. This component consumes 5 out of the 11 time units of its *wcet*. When the component ends, the task's execution time left is then set to 3 time units, which matches the remaining *wcet* for the task. Thus, the component's unused execution time is utilized for compensating the excess due to the previous two local faults (2 time units). Furthermore, after the compensation, there is still some gain time (4 time units), which can be turned into slack time. As a result, after the execution of this component, the two local faults produced previously have been corrected and the system turns again into the no-fault mode for task i.

- Component γ_{i5}. The fifth component then executes in no-fault mode, and its gain time (1 unit) is normally reclaimed as slack time. Note that, if this component had tried to execute for more than 3 time units, the execution time left for task i would have reached 0, and then the system would have immediately detected it and raised a *wcet* exception.

Overall, by following the presented behaviour, the system is able to compensate (whenever possible) for the *local* faults produced by mandatory components inside the task. Therefore, the proposed mechanism effectively delays raising a *wcet* exception up to the last possible moment (i.e., when the total *wcet* of the task is exceeded). This maximizes the amount of mandatory components that can be executed and increases the probability of solving the local faults by means of the compensation mechanism.

At this level, there are actually two alternative compensation mechanisms and it is interesting to note why the framework has selected the one presented above. The two possible mechanisms are based on:

a) Reclaiming each task's gain time at the end of the task's release. This alternative accumulates the gain time being produced at the end of each mandatory component and only turns this time into slack time when *all* the task's mandatory components have been executed.

b) Reclaiming gain time at the end of each mandatory component. This is the alternative adopted by the framework, in which gain time is reclaimed as soon as each mandatory component finishes.

At first glance, alternative (a) seems more reasonable, since it reserves the gain time each task may be producing in order to compensate *in advance* for a potential local fault. In other words, here gain time is turned into a *time budget* for a potential later excess. Furthermore, all the remaining gain time can be reclaimed as slack time at the end of the task's last mandatory component. However, this alternative prevents the optional components of each task (at least these defined between mandatory components) to get any extra slack due to its own task's gain time. As a result, alternative (a) optimizes the case where faults do occur, but worsens the scheduling possibilities of optional components during the normal execution of the system. Conversely, the selected alternative (b) offers the best gain-time performance in absence of faults (which is obviously the regular case), and a compensation mechanism *after* a local fault has actually been produced.

6.4.2 Handling of *wcet* Exceptions

According to the detection mechanism introduced in the previous section, when a *wcet* exception is raised at run time, the handling mechanism now presented is invoked. The handling mechanism is based on the framework's premise that no task can be allowed to run beyond its *wcet*. Therefore, the first action taken by the system when a task produces a *wcet* exception is to terminate this task. The termination process releases any resource (e.g, locked mutexes) allocated for the faulty task and then forces the task to immediately finish its current release. Furthermore, the task's context is reset to its initial status. Thus, at its next periodical release, the task will not start running at the point it was interrupted because of the exception; it will start running its first component, just as if the fault has not occurred. If there is not a handler associated to the faulty task, then the handling mechanism ends with this termination process; otherwise, the task's handler is activated and executed. This option is now presented.

As introduced at the beginning of this section, handlers are defined as extra components (either mandatory or optional) which are attached to the application tasks. In absence of a *wcet* exception, these 'components' are always omitted. On the contrary, when a task for which a handler is defined produces an exception, this handler is activated. The handler will then be scheduled according to its type, as follows:

a) Mandatory handlers. Once released, they become ready entities to be scheduled at the real-time level. For scheduling purposes, a mandatory handler is considered as a ready task, at the priority defined for the handler, containing a single mandatory component.

This allows the first-level scheduler to schedule it away from any particular task, and to select it in advance of a task with the same priority. The handler *wcet*, provided by the application designer, is required in order to calculate each task's worst-case interference. This interference is needed for both the off-line feasibility test and the run-time slack time computation.

b) Optional handlers. These components, when released, are also considered as ready tasks at the real-time level. In particular, the first-level scheduler considers an optional handler as a ready task, at the particular priority defined for the handler, which wants to execute an optional component with a slack fraction of 1.0. Following the scheduling policy introduced in Section 4.4.1, the handler has to wait until it is the highest priority ready task *and* a slack interval can be scheduled for it. When both conditions hold, the system executes the handler in the slack interval, but at the real-time level. If the slack interval is exhausted before the handler has finished, then the handler has to wait until a future moment at which the two mentioned conditions are simultaneously satisfied again.

There is an important issue about handler scheduling that needs further discussion here: to define what happens if a mandatory handler exceeds its *wcet* at run time. This is a real possibility, although obviously it should never be produced, and hence it is explicitly addressed by the framework. When a mandatory handler is executed, the mechanism of detecting task *wcet* exceptions is used in order to control if the handler consumes its maximum computation time. The issue here is to establish which actions to take if the handler actually produces a *wcet* exception. The framework's premise of never allowing a task to exceed its *wcet* implies terminating the handler, but it may be dangerous for the application to just kill running handlers without executing any correcting action. Besides, it is not reasonable for the designer to define handlers to deal with handler's exceptions, because these new handlers would require more handlers, and so on. The solution adopted by the framework is that, if a mandatory handler commits a *wcet* exception, the system interrupts it, and continues its execution in slack time until the handler ends. In other words, from the moment at which the handler fails up to the end of its current release, the system schedules it as an optional handler.

6.4.3 Introducing User-Defined Handlers into the Task Model

This section revises the framework formal task model, formerly presented in Section 4.3, in order to add the user-defined handlers to the task definition.

In the formal task model previously presented, each application task τ_i is defined in the following terms:

$$\tau_i = (T_i,\, D_i,\, O_i,\, C_i,\, B_i,\, H_i,\, M_i,\, \Gamma_i)$$

In this tuple, the factor H_i, which represents the exception handler that the designer may create for task i, is now discussed. The handler H_i is formally defined by:

$$H_i = (Y_i^h,\, C_i^h,\, P_i^h)$$

Within this tuple, the super-index h indicates that the attributes refer to a handler rather than to a task. These attributes are now described:

- Y_i^h is the handler's type, which can be *void* (if no handler is defined for task τ_i), *mandatory* or *optional*:

$$Y_i^h \in \{\mathcal{V}, \mathcal{M}, \mathcal{O}\}$$

- C_i^h is the handler's *wcet*, which must be known and bounded if the handler is defined as *mandatory*:

$$\forall \tau_i \in \mathcal{A},\, H_i = (Y_i^h, C_i^h, P_i^h),\, Y_i^h \in \{\mathcal{M}\} \implies \exists\, C \in \mathbb{N} - \{\infty\} \mid C_i^h < C$$

- P_i^h is a positive integer number, greater than zero, expressing the fixed priority of the handler. This priority has to be explicitly specified in the model because it may be different from the priority of the task for which the handler is created (i).

It is explained above that if a mandatory handler is associated with an application task i, then the handler's *wcet* value, C_i^h, has to be explicitly considered in the feasibility test. However, this value cannot be be directly incorporated to the formula calculating task i's *wcet*, C_i (Equation 4.1, page 39), because both handler and task may be defined with different priorities. Therefore, in the general case, mandatory handler *wcet*s have to be explicitly incorporated to the test's equation. This is presented in Chapter 8.

6.5 Support for Deadline Exceptions

This section presents the framework mechanisms for detecting and handling deadline exceptions. According to the classification discussed in Section 6.3, an explicit support for deadline exceptions is not required in systems meeting three conditions:

a) the system has been guaranteed by means of a complete feasibility test, including both the application tasks and the kernel overhead.

b) The application tasks are never allowed to exceed their *wcet*s at run time.

c) The kernel never produces more overhead than expected.

Both the proposed framework and its corresponding FRTL system are designed in order to fulfill these requirements and thus, tasks are completely guaranteed to meet their deadlines *by design*.

However, in practice, there are situations where it is convenient to count on an extra degree of control and safety at run time: for example, this mechanism was useful while the FRTL was under development; even now, it is useful if FRTL is running on a new hardware platform. In situations like these, the measurements of the kernel overhead factors may not be accurate, potentially jeopardizing task deadlines. The deadline detection mechanism actually allowed the detection of problems within the kernel more easily.

The support for *detecting* deadline exceptions was incorporated to the FRTL system in the early stages of its development, mainly for debugging the kernel itself. This mechanism basically includes a dynamic tracking of the *active deadlines*. An active deadline is the moment at which the next absolute deadline of a ready tasks will expire. The tracking is done as follows: when a task is released, its deadline becomes active and it is set as the current time plus the task's deadline. Then, if the task ends its release before the system time reaches the task's active deadline, then this deadline is no longer considered active until the task is released again; otherwise, if the system time reaches the task's active deadline, then a *deadline exception* is produced for the task. This exception is signaled by an interrupt which is explicitly programmed by the run-time system.

When a deadline exception is produced, the *handling* mechanism which is then invoked is different from the *wcet* handling introduced above, since deadline faults are not provoked by any application task but by the kernel itself, and these faults have to be corrected off line. The handling mechanism includes the following: when the exception is raised, the system writes the event, as well as some specific information about the fault, as profiling information. Then, it stops the application. After the application has been stopped, the profiling information about the fault can be recovered in order to determine the causes of the fault.

The deadline exception support is thus intended to be used by the system developer rather than by the application developer, but it is nevertheless available in the FRTL system as an extra security mechanism. However, since the mechanism itself suffers from significant overhead[5], it is offered as a run-time option. The application may decide to use it or not, by means of an initialization setting. This option is disabled (by default), unless the application explicitly enables it.

6.6 Interface Functions

Figure 6.2 contains the profiles of the FRTL interface functions related to the exception support presented in this chapter. These three functions are available only at the application real-time level and may only be called at the initialization stage. The `rt_task_associate_handler` function is called to create a handler for a specific task, which has to be previously created. The function arguments include the task for which the handler is defined and the name and initial arguments of the function that the handler will run. The initial data to be passed to a handler function are useful when different tasks share the same function as a handler, because

[5] The detection mechanism involves the dynamic maintenance of an ordered list of active deadlines, inserting and removing these deadlines as tasks start and end their releases.

```
extern int rt_task_associate_handler(RT_TASK task,
                                     void   (*func)(void *data),
                                     void   *data,
                                     int    scheduling_type,
                                     RTIME  wcet,
                                     int    priority);

extern int rt_deadline_exception_on(void);

extern int rt_deadline_exception_off(void);
```

Figure 6.2: Set of exception-handling interface functions.

the data may be used to identify the faulty task in the handler's common code. The rest of the arguments are the handler's scheduling type, its *wcet* and its priority. As explained above in this chapter, the scheduling type may be either mandatory or optional. If the handler's type is mandatory, the *wcet* argument is required; otherwise, it is ignored. Finally, the priority parameter is optional: a value of zero within this argument means that the handler inherits the priority of the task for which it is defined.

The last two functions in the figure are used respectively to activate and deactivate the deadline exception mechanism inside the first-level scheduler.

6.7 Summary and Contributions

The present chapter has introduced explicit detection and handling mechanisms for timing exceptions into the framework proposed in this thesis, as a way of giving the framework fault tolerance capabilities. It has first been justified that, if the run-time kernel can be assumed to be predictable and its worst-case overhead is known, task deadlines cannot be missed as long as tasks are never allowed to exceed their *wcet*s. Taking this as a basic premise, the chapter has presented a sophisticated support for detecting and handling *wcet* exceptions at run time.

The detection of *wcet* exceptions has been fully integrated with both the framework task model and the slack time algorithm used by the first-level scheduler. In this mechanism, the gain time of tasks is utilized for compensating (whenever possible) the local *wcet* faults committed by mandatory components inside the task. The presented compensation mechanism permits delaying the moment at which the exception is raised as much as possible, effectively maximizing both the amount of components that can be executed and the probability of compensating the local faults.

The support for handling *wcet* exceptions presents a well-defined behaviour, in which the faulty task is immediately aborted and a user-defined handler, if defined, is activated. This support has taken advantage of all the flexibility present in the framework task model, in such a way that the designer can implement task handlers and associate them to the application tasks in the form of extra mandatory or optional *components*. The definition of handlers is

quite sophisticated. It includes several alternatives, such as associating a different handler to each application task, making several tasks share the same handler, and defining tasks without any handler. Besides, each handler is defined to be executed at a particular priority, which can be different from the priority of the task the handler belongs to.

Finally, detection of deadline exceptions is also provided but only as an option, since in this context a deadline exception can only occur due to excessive overhead produced by the FRTL kernel. When the appropriate measurements of the kernel overhead are introduced within both the feasibility test and the run-time system, this detection mechanism can be safely disabled.

7

Design and Implementation of the FRTL Run-Time System

This chapter presents the run-time support system (or kernel) implementing the framework proposed in this thesis in detail. This system is called *Flexible Real-Time Linux* (FRTL). As its name suggests, FRTL is an RTOS developed by enhancing the original capabilities of RT-Linux v1 in order to achieve a sophisticated support which is appropriate for building FRTS. The FRTL run-time system (or simply FRTL) explicitly offers the framework features discussed in previous chapters: the task model based on mandatory and optional components, the two-level software architecture and the synchronization and exception handling facilities. This chapter explains how these features have been added to RT-Linux. The key design principle observed throughout the development of FRTL has been to maintain the RTOS as predictable and efficient as the original RT-Linux system. Predictability is necessary in order to be able to characterize the RTOS timing behaviour, which can be then introduced in the feasibility test. Clearly, a real-time application cannot be predictable if the RTOS itself is not predictable. Efficiency (that is, low overhead) has to be provided as well, since the response speed demanded by hard real-time applications usually requires exploiting the system processor up to its maximum speed. In this sense, the RTOS would not be useful in practice if a significant part of the processor time were 'wasted' by the kernel. The design and implementation issues discussed in this chapter have made FRTL feature both qualities. This is demonstrated by means of the complete feasibility test and actual overhead measurements, respectively, which are both presented in Chapter 8. Besides the run-time system itself, FRTL also includes a set of system

debugging and profiling tools. These tools are also described at the end of the chapter.

7.1 Overall Description of FRTL

This section introduces an overall description of FRTL, emphasizing how the framework functionality has been implemented. The internal design and the implementation details of each level are then exhaustively discussed in the two following sections.

The general design of FRTL has been developed by combining the functional requirements of the framework software architecture (described in Section 4.4) and the original RT-Linux philosophy. The resultant FRTL system is thus decoupled in two levels, the *real-time level* and the *Linux level*. These levels assimilate the framework real-time and non-real-time levels, respectively. This general scheme is shown in Figure 7.1. In the figure, square boxes in light grey represent the different parts of the FRTL support system: in the real-time level, these parts are the first-level scheduler and the frames defining the application tasks; in the non-real-time level, the FRTL parts are the second-level scheduler and the frames defining the application optional components. Inside both the application tasks and the application optional components, rounded white boxes represent the actual application code. This code corresponds to the task mandatory components and the optional component versions, respectively. Ovals, such as the one located at the top of the figure or inside the Linux kernel box represent memory areas. Finally, white boxes depict parts of Linux which have not been changed.

7.1.1 FRTL's Real-Time Level

According to the framework proposal, the FRTL real-time level has to include the first-level scheduler and the real-time side of the application (that is, the definition of tasks and the code of their mandatory components and exception handlers). This level has been implemented by modifying the original support provided by RT-Linux v1. This support is formed by a few simple, well-designed, efficient abstractions and services providing hard real-time capabilities. These capabilities include the *soft interrupts* mechanism, the *rt-task* abstraction (or thread periodically running an application function, inside the Linux kernel address space), a real-time scheduler dispatching rt-tasks according to their priorities, and the RT-FIFO facility (a communication channel between rt-tasks and regular Linux processes). All these features have been described in Section 3.3.2.

Chapters 4 to 6 have introduced the functionality which is expected from the first-level scheduler, including both specific services to be provided to the application and certain internal activities which have to be performed. The following subsections summarize these services and activities, explaining their implementation in FRTL.

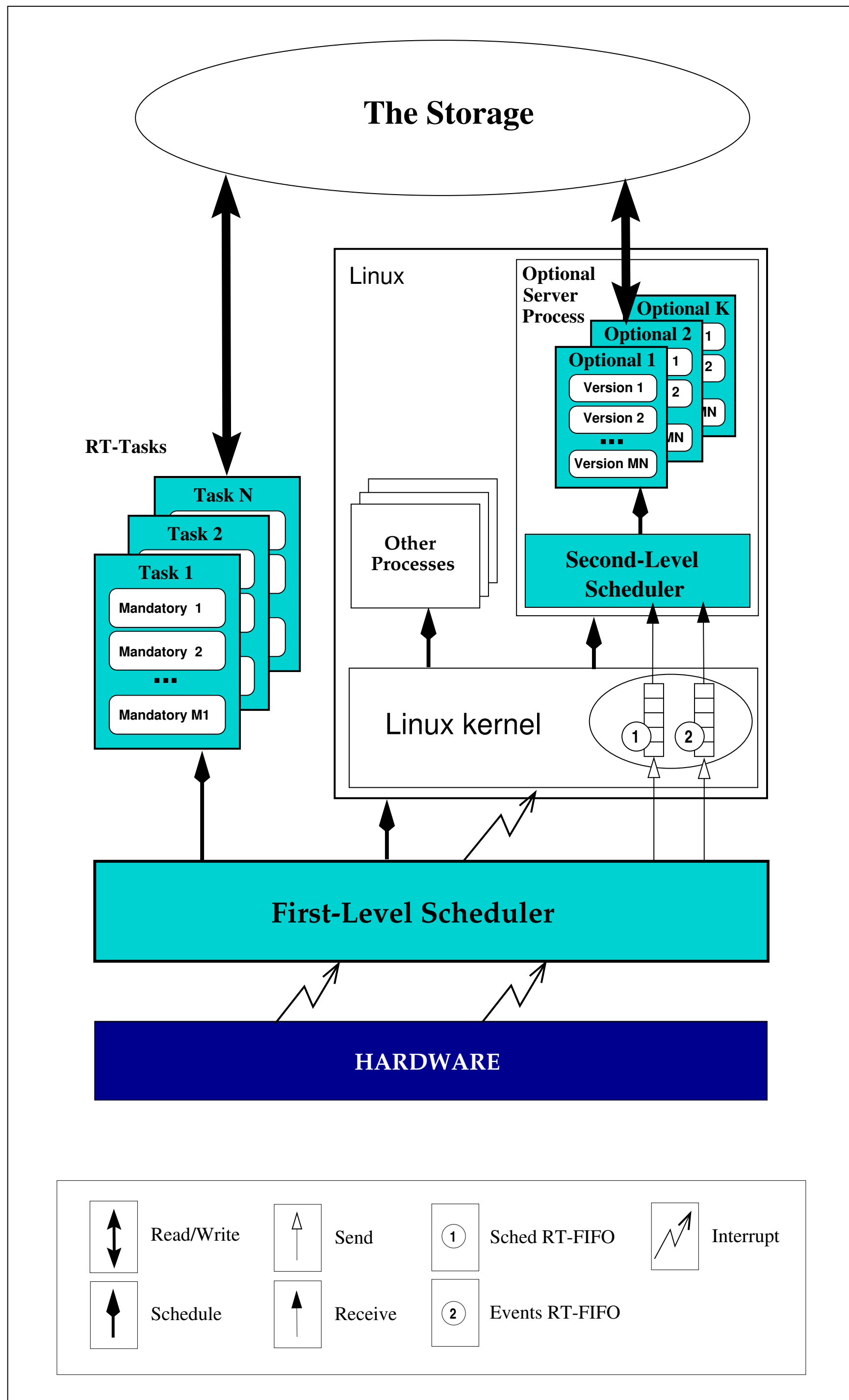


Figure 7.1: The Flexible Real-Time Linux system.

Task handling

The framework's task model defines each application task as a sequence of components. At the real-time level, only one component inside each ready task is actually able for execution (the task's *running component*). Thus, only one execution thread is actually needed for implementing each application task. For this reason, the first-level scheduler creates one rt-task run each application task and releases this rt-task according to the task's period. Each time the rt-task is released, it executes the functions implementing the mandatory components belonging to that particular task in order.

To do this, each rt-task is made to run a generic, private 'wrapper' function[1] which internally executes the components of a given task i. This function is is structured in two nested loops. The first outer loop is an endless loop which iterates once each time the task is periodically released. In each iteration, it completely traverses i's list of components by means of an inner loop. The rt-task suspends itself at the end of this inner loop, when every component has been visited. Then, the scheduler will release the rt-task again at i's following period. The inner loop visits one component γ_{ij} per iteration. This component can be either mandatory or optional (in the optional case, the rt-task actually considers the list of contiguous optional components γ^*_{ij}). If component γ_{ij} is mandatory, the rt-task invokes γ_{ij}'s function, that is, it directly executes the component. Otherwise, if γ_{ij} is optional, a message is sent to the second-level scheduler (by means of a dedicated RT-FIFO, called *events fifo*) for each optional component in γ^*_{ij}. These messages include the identifiers of the components and their deadlines, so that they can be activated inside the non-real-time level. Then, the scheduler checks whether or not a slack interval can be scheduled for the non-real-time level.

Slack computation

The FRTL first-level scheduler uses the system slack time in order to determine the execution intervals of the non-real-time level, according to the calculations presented in Section 4.4.1. These calculations are based on the available slack time at each priority level, which the scheduler has to dynamically update. This slack update may be performed by using a standard slack stealing algorithm. Among the different algorithms proposed in the literature, the scheduler utilizes the Dynamic Approximate Slack Stealing algorithm (DASS) [Dav93b]. The DASS algorithm has been selected, after a performing a comparative study presented in [Gar97b], because it features an appropriate balance between its performance (i.e., the amount of slack time it is able to *steal*) and the overhead it produces.

Release of dynamic mandatory components

The FRTL first-level scheduler includes the algorithm of accepting firm tasks introduced in [Dav95] as the test for accepting mandatory dynamic components when they are released.

[1] A *wrapper* is a function used to enclose the invocation of other function(s). This is used in situations where the invocation involves other specific actions that must be hidden from the function(s) being called.

The test actually implemented has adapted the original algorithm in order to use the DASS algorithm instead of the original *exact* slack stealing algorithm (DSS). The current implementation only allows a dynamic mandatory component to be released by another mandatory component (either dynamic or defined inside a task). That is, mandatory dynamic components cannot be invoked from the non-real-time side of the application.

Scheduling policy

The first-level scheduler implements the scheduling policy described in Section 4.4.1. This policy is rewritten here in terms of the implemented run-time system.

- If the running component γ_{ij} of the highest priority runnable task i is mandatory, the scheduler directly puts i's *rt-task* to execution since, as explained above, mandatory components are executed by rt-tasks.

- Otherwise, if γ_{ij} is optional, the scheduler checks whether there is slack available for γ^*_{ij}. If so, the scheduler calculates the next slack interval and sends a message to the second-level scheduler (by means of another dedicated RT-FIFO called the *sched fifo*), indicating the length of the interval. Then, the scheduler runs the *Linux task*[2] during this interval. The FRTL non-real-time level, which is actually implemented as a user level process inside Linux, can be subsequently executed, as explained below.

  However, if slack is not available for γ^*_{ij}, then the scheduler executes i's *rt-task*. Inside this rt-task, the optional components belonging to γ^*_{ij} are skipped and the next mandatory component of i, if any, is then executed.

Please note that the abstract message queue, which according to Section 4.4.1 is used for communicating the two schedulers, is actually decoupled in two RT-FIFOs, the *events fifo* and the *sched fifo*. The former is used by rt-tasks to inform the second-level scheduler about the optional components to activate and their respective deadlines. The *sched fifo* is used each time the non-real-time level is executed during a slack interval, to both inform the second-level scheduler about the length of the interval and to achieve the synchronization between the two schedulers imposed by the software architecture. This is further detailed below.

Synchronization mechanisms

As presented in Chapter 5, the framework proposes two alternative synchronization mechanisms in order to serialize the access of the application components to their shared data. The mechanisms provide mutexes following adapted versions of PCP and CSP algorithms, respectively. Since mandatory and optional components are executed at different running levels, access to mutexes has to be provided at each level by means of the corresponding scheduler. However, this has to be done in a coordinated manner. Thus, both schedulers have to be able to access the *same* information about the status of each mutex; because of this, the internal

[2]Please recall that in RT-Linux, the entire Linux is run by means of a special rt-task, usually called the Linux task.

information about mutexes is stored in a shared memory area which is accessible from both system levels. This area is called *the storage* in Figure 7.1. A portion of the storage is reserved in advance for the schedulers' private use, including the mutex support and the debug mechanism described at the end of this chapter, in Section 7.4.

Timing exception mechanisms

The timing exception support defined by the framework includes the detection and handling of *wcet* faults and (optionally) the detection of deadline faults at the real-time level. Thus, these actions are performed by the FRTL first-level scheduler. In particular, the scheduler detects *wcet* exceptions by keeping track of the remaining execution time of each runnable task. Deadline exceptions are detected by keeping track of the active deadlines, that is, the next absolute deadline of all ready tasks. At each scheduling point, the first-level scheduler programs a timer interrupt for the nearest potential exception (that is, the nearest active deadline or the remaining execution time of the task which is about to execute) and then raises the exception if this interrupt is actually produced.

According to the exception support described in Chapter 6, when a *wcet* exception is produced, the faulty task is terminated and the task's handler, if any, becomes runnable. Therefore, the task's handler requires an execution thread which is different than the task's thread. For this reason, the first-level scheduler creates a rt-task for each *wcet* handler and makes it run another generic wrapper function. This wrapper is internally structured in an endless loop, which internally invokes the handler's function and then suspends itself until the scheduler releases the handler again. As a result, each time the handler rt-task executed, it runs the handler once. On the other hand, the framework requirement of scheduling tasks and threads independently also forces the FRTL first-level scheduler to create a different thread for executing each one.

7.1.2 FRTL's Linux Level

As shown in Figure 7.1, the framework non-real-time level is implemented as a regular user-level Linux process, called *optional server*. In this process, the application code corresponding to the optional component versions is linked along with a set of library functions implementing the second-level scheduler. The *optional server* process is executed at the Linux real-time class highest priority, which ensures that each time the first-level scheduler executes Linux, the optional server process always takes control in preference to any other runnable user process.

The optional server process is a multi-threaded process, with one thread running the second-level scheduler. The scheduler then creates a different thread for each optional component version which has to be run. This allows each version to be independently selected and executed, according to the scheduler policy. The internal design of this process ensures that each time the process is executed by Linux, only the scheduler thread is actually able to run. The threads corresponding to all the versions are stopped in such a way that they can only be executed by the second-level scheduler. The second-level scheduler runs an endless loop, with

one iteration per slack interval. The following describes the implementation of the sequence of actions executed by the scheduler in each iteration:

1. At the beginning of the loop, the scheduler is suspended by having previously read the *sched fifo* in a blocking manner. The entire process is thus stopped until the message from the first-level scheduler arrives. When this occurs, the second-level scheduler wakes up and gets the length of the slack interval during which it will execute from the message.

2. The scheduler then retrieves the messages in the *events fifo*. These messages indicate the optional components to be activated and their deadlines. The fact of having two message queues instead of one allows for a more efficient and flexible implementation of the second-level scheduler. Also, this solution is more efficient because if an optional component is ready during more than one slack interval, its activation is reported only once to the scheduler (in its first slack interval); hence, the time spent checking which *new* components have to be activated is reduced. Having two queues is also more flexible, because the *events fifo* can also be used by the application components in order to inform the second-level scheduler about other dynamic information which is significant for its particular scheduling policy (e.g. changes in the environment). Thus, the *sched fifo* is only used for synchronizing both schedulers, while the *events fifo* is only used for communicating all the *events* required for the non-real-time level scheduling policy to the second-level scheduler.

3. At this point, the scheduler has to remove all the expired optional components (those whose deadlines have been reached) from the list of active components.

4. The rest of the slack interval is used by the scheduler for running the versions belonging to the ready optional components. In order to do so, it enters into an inner loop, in which it iterates while the end of the slack interval has not been reached and there are optional components still to schedule. In each iteration, it selects a particular version of a ready optional component, and then runs this version for a certain time, which is always less than or equal to the remaining slack interval. The three decisions taken by the scheduler in each iteration (component, version, and time to run) are the output parameters of a *single* function inside the second-level scheduler code. Thus, the non-real-time level scheduling policy can be changed, if necessary, by simply reprogramming this function.

5. The scheduler exits the inner loop when either the slack interval is almost consumed or the scheduler determines that there is nothing else to execute. At this moment, before Linux is actually preempted by the real-time level, the second-level scheduler suspends itself by reading the *sched fifo*. This suspends the entire optional server process until the following slack interval, when this sequence will start again. The time remaining (i.e., the time from the suspension of the second-level scheduler to the actual moment at which Linux is preempted by the real-time level) is then used by other Linux processes.

In the development of the FRTL non-real-time level, two alternative design approaches were evaluated. The first one is to consider that the entire Linux is the non-real-time level, while the second one is to implement this level inside a user-level process. It is now explained why the second alternative was finally chosen. In the first approach, the second-level scheduler corresponds to the Linux scheduler, and the optional component versions could be implemented as either different processes or different kernel threads running in the same process. Compared to the second alternative, this approach eliminates one step in the non-real-time level execution, making both FRTL schedulers be more tightly coupled. Although this is a good feature, the first alternative design was rejected for two reasons. First, as the Linux kernel is in constant development (new versions appear very often), the code of the second-level scheduler would require constant reviews in order to keep it compatible with these new versions. And second, the addition of new scheduling policies to the non-real-time level would be extremely more complex for the application developer. As a result, the decision of building the non-real-time level at the Linux user level was taken for both compatibility and scalability reasons. Furthermore, the specific design and implementation of the non-real-time level inside the *optional server* process has achieved a reasonable degree of responsiveness at the user level, which is considered to be sufficient for the non-real-time level.

7.1.3 The Storage

The shared memory area by which mandatory and optional components communicate is called *the storage* in the FRTL system, as depicted in Figure 7.1. Access to this shared memory is provided by the so-called *Storage Manager*. This is an abstract entity which provides a structured way of accessing the memory (that otherwise would be a raw set of bytes) to the application. As happens for other parts of FRTL, the functionality provided by the Storage Manager has been decoupled and implemented inside both the first-level and the second-level schedulers. The Manager's functionality includes two facilities:

a) A special mechanism which allows both system levels to physically share a memory region. This mechanism has been developed by the RT-Linux community[3]. Part of this shared memory is reserved to be used by the first-level scheduler in order to implement the debug and profiling mechanism described at the end of this chapter. The rest of the shared memory is the area actually called *the storage*, which the Storage Manager makes available for allocating shared memory objects.

b) A predictable dynamic memory allocator of objects in the storage. This allocator allows the two schedulers and the application components to allocate memory objects in (and to deallocate them from) the storage in a *bounded* time. Once allocated, objects are transparently accessed from both levels of the system. A part of the storage is reserved for the schedulers' private use, which includes the implementation of mutexes:

[3]The actual mechanism of memory sharing implemented in RT-Linux v1 has an upper bound on the amount of memory that can be shared, despite of the amount of physical memory available on the computer. This limit is exactly of 4 megabytes minus one byte.

the structure storing the internal status of each mutex created by the application is allocated in the storage by the first-level scheduler; then this structure can be accessed by either scheduler when needed. The rest of the storage is then made available for the application components in order to allocate (and use) their own memory objects at run time.

The Storage Manager is not further detailed here because it has not been developed by the author of this thesis. A complete description can be found in [Esp98].

7.2 The Real-Time Level

This section describes the FRTL real-time level in detail, specially discussing the design and implementation of th first-level scheduler. The section first discusses the main design principles which form the basis of both the scheduler internal structure and its run-time behaviour. By following these principles, the FRTL real-time level has been implemented as a predictable, efficient and scalable real-time kernel. Second, the section presents some relevant implementation details of this level.

7.2.1 Interrupt-Driven Kernel

The first design decision that influences the timing behaviour of the real-time level is that it is *interrupt driven*, as opposed to being *tick driven*. This means that, rather than producing an interrupt at each clock tick, the hardware timer is programmed to interrupt only at certain time points. These particular points are significant for the first-level scheduler because they signal *time events*, such as the release of a periodic task, the end of the scheduled slack time or a *wcet* exception.

The interrupt-driven scheme is more efficient than the tick-driven approach, because it produces less overhead: interrupts are produced only at instants which have relevancy for the system. However, this design results in a more complex implementation, since the use of the hardware timer has to be multiplexed among all the different time events required by the kernel. This multiplexing can be done by both maintaining a list of future time events and executing the following actions each time an interrupt is produced:

1. Consulting the hardware timer in order to set the current system time.
2. Checking the list of time events in order to find out the particular time event (or events) for which the current interrupt was programmed. This may cause the execution of further code dealing with this particular interrupt.
3. Removing the time event(s) corresponding to the current timer interrupt from the list and programming the timer to interrupt at the following (nearest) time event.

This scheme is not only relevant for the real-time level internal design but also for the system schedulability test, as shown in Chapter 8.

7.2.2 Common Entry and Exit Points

As every operating system, the first-level scheduler is an *event driven* system, meaning that its code is always executed as a result of a *system event*. A system event is typically either a hardware interrupt or a system call invoked by the application. In response to such an event, the system respectively executes the handler associated with the interrupt (usually called Interrupt Service Routine (ISR)) or the function associated with the system call.

Some real-time scheduling policies require that the functions implementing interrupt handlers and system calls see an updated snapshot of the system, specially of the dynamic timing information (such as the current system time). Therefore, this timing information must be updated *before* these functions are executed. In particular, dynamic slack stealing algorithms need such an update, which must include not only the current system time but also the recalculation of the slack time at each priority level.

In order to meet this requirement, the first-level scheduler has been designed to have a homogeneous scheme for handling every system event. In this scheme, the scheduler code dealing with each particular system event is structured in the following three sequential steps:

1. *System Update*. This update is performed by calling a function that updates both the system time and the dynamic timing attributes of each application task (such as its slack time, execution left, blocking factor, etc.). This function is also in charge of releasing tasks according to their periods and of storing profiling information about the execution of tasks.

2. *Specific code*. In this step, the code dealing with the particular event produced is invoked. Typically, the scheduling conditions for the running task or other tasks are modified by this code. For example, the specific code for a PCP mutex lock system call may suspend the calling (running) task.

3. *Rescheduling*. After an interrupt or system call is produced, the scheduler is normally required to choose the task (and component) to be run next. This is done by calling a rescheduling function. If the selected new task is the same one that it was running before the event was produced, then the call to the rescheduling function simply ends, making that task resume execution. Otherwise, an explicit context switch is produced to the new running task. In any case (before either ending or producing a context switch) this function has to reprogram the hardware timer to interrupt at the next time event.

Thus, the *update* and *rescheduling* functions may be considered, respectively, as the common *entry* and *exit* points of all the events processed by the first-level scheduler. The design decision of having common entry and exit points has been adopted not only for the reasons expressed above, but also because it actually simplifies the system timing behaviour and the addition of new facilities. This effectively makes the FRTL system easier to analyze as well as more scalable.

7.2.3 The Scheduler's Layered Structure

Many general-purpose operating systems are conceptually structured in levels or *layers*, with each layer being supported by the functionality provided by the previous one(s). Some of those systems are even implemented in perfectly isolated parts, related only by some means of communication mechanism (such as the Minix operating system [Tan97]). This layered internal design, either conceptual or actually implemented, permits building up the system in a progressive fashion, reasoning about it at different abstraction levels and detecting (and correcting) errors more easily, as shown by Dijkstra in [Dij68].

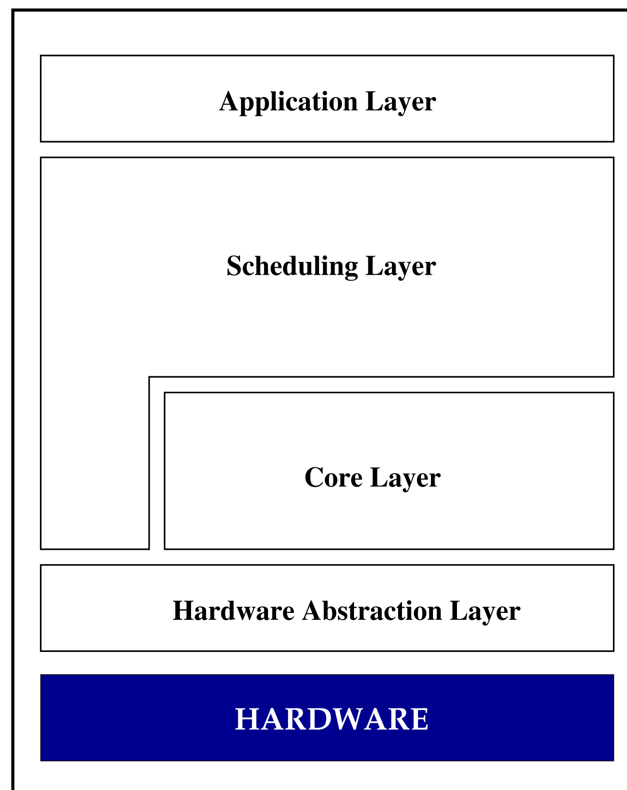


Figure 7.2: The internal layered design of the FRTL's real-time level.

This design paradigm may also be applied to hard real-time kernels, even though they are usually simpler than typical general-purpose operating systems, in order to be small and to provide a predictable behaviour. As a result, the FRTL real-time level has been designed to be structured in the following four layers, as depicted in Figure 7.2:

a) **Hardware Abstraction Layer**. The purpose of this layer is to hide all particular details of the hardware architecture. It contains the hardware-dependent code, normally written

in assembly language, which basically includes the lowest level of the hardware interrupt handling and the rt-task context switch code. Practically all of this layer's code is hardcoded into the Linux kernel and most of it has not been changed from the original RT-Linux v1 version.

b) Core Layer. This layer provides the upper levels with the functionality which is independent from the specific scheduling policy and task model that the kernel offers to the application. This functionality includes the *task context* abstraction and some basic actions which are needed regardless of the system scheduling policy.

The task context abstraction hides the actual low-level elements forming an rt-task context (the stack, the Floating Point Unit (FPU) registers, the hardware interrupt mask, etc.) by providing functions for creating, destroying and executing a rt-task context. A specific function for creating the context of the Linux rt-task (i.e. the task that *executes* Linux) is also provided, since this rt-task has some special characteristics that differentiate it from the application regular rt-tasks.

According to this layered design, the two common steps which are executed when processing a system event (that is, *system update* and *rescheduling*, as presented in Section 7.2.2) have also been decoupled into two functions, one at the Core layer and another at the Scheduling layer. The system update step is decoupled in the `rt_update_system` and the `rt_update` functions, while the rescheduling step is decoupled in the `rt_dispatch` and the `rt_schedule` functions.

This layer provides the `rt_update_system` and `rt_dispatch` functions, which execute the policy-independent actions of the related steps. The `rt_update_system` is the first function to be called at each event being processed by the first-level scheduler, since it updates the system time (clearly, this is required by any further update). The `rt_dispatch` is the last function to be called at the end of processing the event, after the scheduler has decided the next task to execute and the moment corresponding to the next time event. Taking these two parameters, the `rt_dispatch` function programs the hardware timer to interrupt at that moment and then restores the context of the selected rt-task[4].

According to this design, the `rt_update_system` and `rt_dispatch` functions are actually the first and last function being invoked within the first-level scheduler at *every* system event. This actually permits the implementation of two interesting built-in mechanisms. The first one is the automatic disabling of hardware interrupts within the scheduler code: when an event is produced, the `rt_update_system` saves the *interrupt state* of the running rt-task in its context and then disables the hardware interrupts; in its turn, the `rt_dispatch` function restores the interrupt state of the rt-task selected

[4]If the selected rt-task was the same one that was running before processing the system event, then its context is not actually restored. In this case, the `rt_dispatch` function finishes, returning to the invoking function, which also ends, and so on. When the all the pending functions inside the scheduler code have finished, the rt-task automatically *resumes* execution.

to run, just before either returning or context switching to this rt-task's context. As a result, the kernel code is always executed with the hardware interrupts disabled without requiring any action from the upper layers. The second mechanism implemented by means of these two functions is a policy-independent mechanism for measuring the kernel overhead. This mechanism is further explained in Section 7.4.

c) Scheduling Layer. This layer supports the particular task model and scheduling policy provided by the first-level scheduler and implements the actual services available to the application. In particular, the scheduling policy is implemented within the `rt_udpate` and `rt_schedule` functions. These functions include the policy-dependent code of the update and rescheduling steps executed at each system event. Figure 7.3 illustrates the internal structure of a *generic* interrupt handler or system call (represented in the figure as the `rt_system_event()` function), illustrating the actual functions involved. The figure also indicates the sequence of function calls performed each time a system event produces the execution of this generic `rt_system_event()` function.

The `rt_update` function first calls the Core layer's `rt_system_update` function to update the system time and then performs the specific update required by the first-level scheduler policy (it recomputes the slack time at each required priority level and releases periodic tasks according to their periods). After processing the specific event produced, the `rt_schedule` function is called. This function decides which rt-task has to be executed next and which is the next relevant time event for the real-time level; then it calls `rt_dispatch` indicating both data.

d) Application Layer. This layer contains the real-time level of the application. It only uses the services provided by the previous layer, which altogether form the set of system calls available. These services can be divided into three groups: the initialization functions, the system calls and the cleanup functions. This classification is the topic of the following section.

The proposed layered design, besides the advantages mentioned at the beginning of the section, provides a rational structure by which the kernel can be enhanced with a reasonably low number of changes required. The separation between the Hardware Abstraction and the Core layers permits the system to be ported to other hardware architectures by simply re-implementing the Hardware Abstraction layer. The separation between the Core and the Scheduling layers allows the scheduling policy and task model to be easily changed by simply changing the Scheduling layer. Furthermore, new facilities can be easily added to the system, in the Scheduling layer, without having to deal with several kernel low-level details. Overall, this design has facilitated the addition of functionality to FRTL in a progressive and consistent manner.

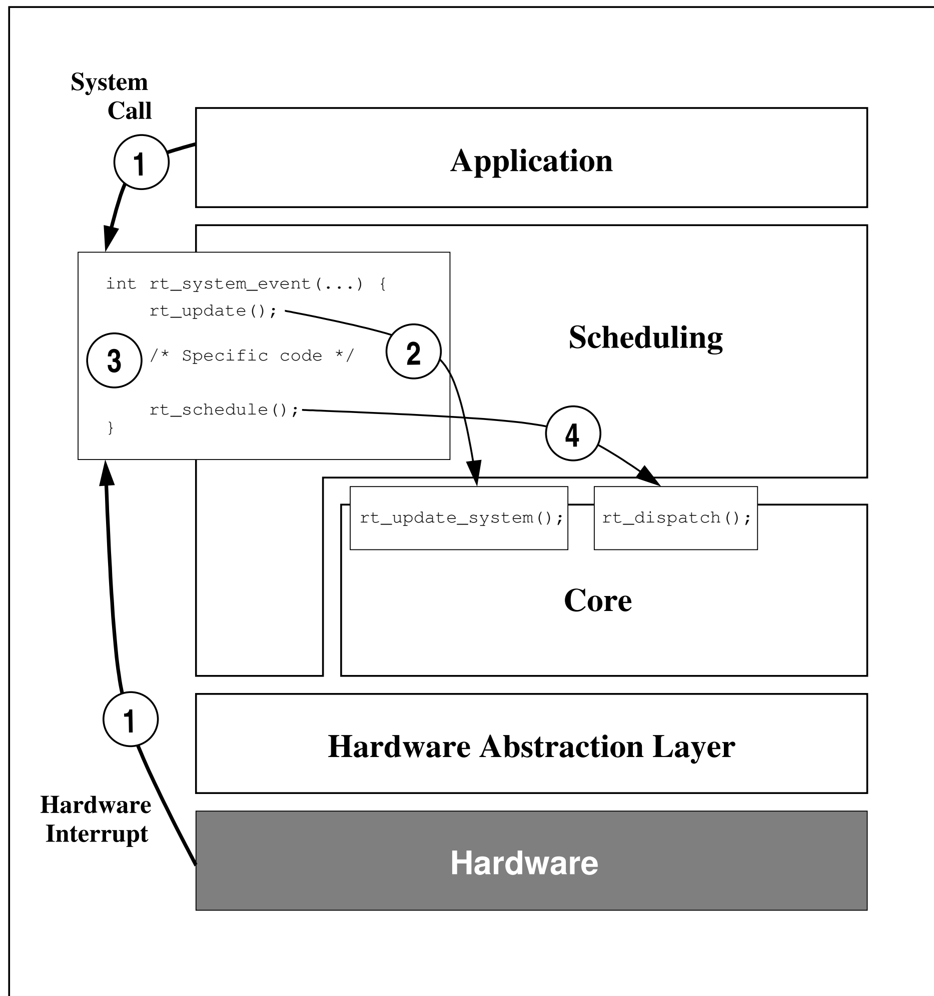


Figure 7.3: Sequence of calls when processing a system event: (1) The system event is produced when either the application invokes a system call or the hardware timer interrupts the processor; each possible event produces the execution of a scheduler function which is here generically called `rt_system_event`. (2) `rt_update` first calls `rt_system_update` and then executes its own update. (3) The particular event produced is processed inside `rt_system_event`. (4) After processing the event, `rt_scheduler` is invoked; this function selects both the next rt-task to run and the next instant for a time event, and, finally, it invokes `rt_dispatch` to run that rt-task and to program the hardware timer to interrupt at the time event instant.

7.2.4 Taxonomy of System Calls

In many RTOSs, and also in the Realtime Extensions of the POSIX standard, each interface function offered to the real-time application can be called at any time during the application execution. For example, there is no restriction for creating new threads while the application is running. This presents a problem since the application cannot be sure in advance that all the resources it sequentially requests are available. Following the example, if an application invokes a system call for creating a new thread and the creation function fails, then the application may completely crash while actually running. Clearly, this is not acceptable in hard real-time systems.

Because of this, the set of interface functions provided to the Application Layer in FRTL has been divided into three groups, depending on the moment at which they are allowed to be called:

a) **Initialization functions**. Functions in this group can only be called *before* the application starts running. These include the functions for creating the application tasks, components, and all other resources required (mutexes, exception handlers), the functions for configuring the first-level scheduler (in aspects such as the mutex policy or the deadline exception mechanism), and the function to start running the application (called `rt_app_start`). This function presents two purposes. The first purpose is calculating some values required by the time the application starts running but that cannot be calculated after *all* tasks have been created (such as the initial slack time at each priority level). The second purpose is to stop accepting calls to initialization functions. This fact ensures that all the resources needed by the application are available before it actually starts running, or else it is not started.

b) **System calls**. The set of system calls groups all the services the first-level scheduler offers to the application real-time level, while it is running. In the current FRTL support, this group is formed by the following functions: `rt_mutex_lock`, `rt_mutex_unlock` and `rt_task_run_dynamic_mandatory`.

c) **Cleanup functions**. Since hard real-time applications are normally used for controlling some environment, they are supposed to work continuously, without any stop. However, there are circumstances where the application must be switched off (e.g., during the application test phase or during some software or hardware fixing). Stopping such an application involves to carefully deallocate all the resources being used, which can be a non-trivial process. For this reason, the FRTL system offers a special function to safely stopping the application (`rt_app_stop`). This function is in charge for correctly deallocating all the resources and for properly stopping all hardware devices, thus ensuring that the system remains in a safe state until the moment to start the application again. This has been very useful during the testing phase of FRTL.

7.2.5 Implementation Details

This section presents the implementation issues that influence the timing behaviour of the first-level scheduler. The discussion presented here is referenced afterwards in the sections analyzing this timing behaviour, in Chapter 8.

Disabling of Hardware Interrupts

Consider a situation in which an operating system is running a system call on behalf of the running task and a hardware interrupt arrives, making the system execute the corresponding service routine. In situations like this, more than one thread may be executing the operating system code. Thus, sensitive sections of the system code, such as access to global variables, have to be protected from race conditions. The disabling of hardware interrupts is one of the most widely used mechanism for obtaining mutual exclusion in the code of an operating system. This mechanism has the negative effect of delaying the notification of a hardware interrupt, if the interrupt is received while the scheduler has disabled interrupts. The disabling may be done locally, in order to protect only certain sections of code, or globally, that is, affecting the entire operating system code. Systems using local disabling are more efficient, but have to consider the possibility of multiple execution threads in the sections of code in which interrupts are not disabled, which normally leads to more complex design and coding. The global disabling generally produces longer delays in the processing of interrupts, but allows for a clearer implementation of the operating system. Obviously, this issue has to be carefully considered when designing an RTOS.

The FRTL first-level scheduler uses a global disabling of interrupts as a means of internal synchronization. In particular, as stated in Section 7.2.3, interrupts are automatically disabled at the beginning of `rt_update_system` and enabled again at the end of `rt_dispatch`, just before returning (or context switching) to the running task. This decision is based on the fact that the scheduler code is relatively small and has been designed and coded very efficiently. Therefore, disabling interrupts only in special sections of code is not worthwhile, since this would lead to a small gain in performance but to a great increase in code complexity.

Implementation of the Application Tasks

From the application's point of view, tasks are actually a sort of *frame* containing optional and mandatory components which are periodically executed. In fact, the application designer writes the code of components and then defines tasks by adding their corresponding components. Therefore, an important issue in the FRTL first-level scheduler is how application tasks are internally implemented.

In the FRTL real-time level, each application task is executed by an rt-task that is made to execute a private function called `rt_task_runner`. A simplified version of this function is presented in Figure 7.4. This generic function receives a single argument: a pointer to the private structure containing all the information about the particular application task that the

```
void rt_task_runner(rt_task_t *task) {
      rt_component_t  *comp;
      while (1)
              comp = first_component(task);
              while (comp != NULL) {
                      task->running_component = comp;
                      if (is_mandatory(comp)) {
                               (comp->func)(comp->data);
                               rt_gain_time_update(task,comp);
                               comp = comp->next;
                       } else {
                               rt_run_optional(taks,comp);
                               comp = next_mandatory(comp);
                       }
              }
              rt_task_end();
       }
 }
```

Figure 7.4: The `rt_task_runner` function.

function has to execute. From that structure, the `rt_task_runner` function extracts the information about the specific components belonging to the task.

The `rt_task_runner` contains two nested loops. The outer loop is endless, performing one iteration each time the task is periodically activated. In its turn, the inner loop is in charge of completely traversing the task list of components, starting at the first component and calling the `rt_task_end` function after the last one. Each iteration of the inner loop examines one component. This component is set as the task's *running component* (this attribute is later checked in the `rt_schedule` function). If the running component is mandatory, its function is directly called; after that, the `rt_gain_time_update` function is invoked. Otherwise, if the component is optional, the `rt_run_optional` function is invoked; after this call is finished, the rest of contiguous optional components are skipped until the task's next mandatory component, if any, is found.

An important issue at this level is that these three functions invoked within the `rt_task_runner` function (`rt_gain_time_update`, `rt_run_optional` and `rt_task_end`) are actually *implicit system calls* for the first-level scheduler. The term implicit here means that these invocations are hidden to the application programmer. Nevertheless, these three functions are proper system calls, meaning that they are internally designed by following the three-step scheme explained above in Section 7.2.2 and executed with the interrupts disabled. The purpose of each of these functions is now briefly reviewed.

The `rt_gain_time_update` function is used to keep track of the task's *gain time*. As explained in Section 6.4.1, the first-level scheduler uses the gain time corresponding to mandatory components for two alternative purposes: if a task has committed one or more local faults, then the gain time produced by each of the task's mandatory components is used to compen-

sate for the fault(s). Otherwise, this gain time is reclaimed as slack available for the task's priority level and all lower priority levels. These two alternative actions are performed inside this system call.

When the current running component γ_{ij} is optional, the `rt_run_optional` function is invoked. This system call first builds the list of contiguous optional components γ^*_{ij}, with the current component leading this list. Then, it checks whether or not there is slack time available for γ^*_{ij}. If there is some slack available, the `rt_run_optional` function sends a message to the second-level scheduler for each optional component in the list. In each message, the component's identifier and its deadline are included. These messages are sent through the *events fifo*, as explained in Section 7.1. At the end of this system call, the `rt_schedule` function is called. In this case, the function will select component γ_{ij} for execution and, after calculating the current slack interval, it will execute Linux during the interval. Please note that, since all the contiguous optional components in γ^*_{ij} are activated at once, there is no need to return to the rt-task (hence to the real-time level) in order to activate and schedule each one. For this reason, when control returns to the rt-task code, it has to advance not just to the next component, but specifically to the next mandatory one.

Finally, the `rt_task_end` is invoked when there are no more components to be run in the task. In this situation, the current release of the task is finished and the task must be suspended until its next periodic release. Then, this system call calculates the slack time at the task's priority level and changes the task's status (from *ready* to *delayed*). This necessarily makes another task be selected to run when the `rt_schedule` function is called at the end of this system call.

Handling of Timer Interrupts

The first-level scheduler has direct access to the hardware interrupts, meaning that it can capture any of these interrupts in preference to the Linux kernel. To do so, it has to install a handler for a particular interrupt. This handler is automatically invoked when this interrupt is produced.

The only hardware interrupt used by the first-level scheduler is the timer interrupt, which is captured by the `rt_timer_handler` function. As explained in Section 7.2.1, the scheduler has to multiplex this timer in order to produce the different types of *time events* required by the real-time level. Specifically, the first-level scheduler distinguishes three types of time events:

a) The release of an application task.

b) The end of a slack interval.

c) A timing exception (either a *wcet* or a deadline exception) caused by a task.

The multiplexing of the hardware timer is done by means of two actions, executed by the first-level scheduler:

```
void rt_timer_handler(void) {
        rt_task_t *task;

        /* (1) System Update */
        rt_update();

        /* (2) Wcet and Deadline exception checking */
        if (running_task != LINUX_TASK
            && running_task->execution_left == 0) {
                rt_raise_exception(running_task, WCET);
        }
        if (deadline_exception_support_activated) {
                task = list_get_head(list_of_active_deadlines);
                now  = get_time();
                while (task != NULL && task->next_deadline <= now) {
                       rt_raise_exception(task, DEADLINE);
                       list_remove_head(&list_of_active_deadlines);
                       task = list_get_head(list_of_active_deadlines);
                }
        }

        /* (3) Rescheduling */
        rt_schedule();
}
```

Figure 7.5: The `rt_timer_handler` function.

- At the end of executing each system event (inside the `rt_schedule` function), the first-level scheduler examines the time events (corresponding to any of the three types) to be produced in the future and programs the timer hardware to interrupt at the nearest of them.

- When a timer interrupt is produced, the timer handler of the first-level scheduler (called `rt_timer_handler`) is automatically invoked. In this function, the scheduler examines the event(s) corresponding to the current system time. Please note that several time events may be produced at the same time (for example, the release of two tasks and the end of a slack interval may happen at the same time). A specification in pseudo-code of this function is presented in Figure 7.5. This function only deals with time events corresponding to timing exceptions explicitly, because the other two types of events (a task release and the end of a slack interval) are processed inside the `rt_update` and `rt_schedule` functions, respectively.

Implementation of Timing Exception Support

As introduced in Chapter 6, the first-level scheduler includes a specific mechanism for detecting *wcet* and deadline timing exceptions at run time. This mechanism is actually implemented

by using the two-action technique for multiplexing the hardware timer, described in the previous section.

Within the first action (programming the timer) two issues related to the exception detection need further discussion:

- A potential *wcet* fault only has to be considered in the programming of the nearest interrupt if the scheduler has selected a mandatory component for execution. Obviously, if the non-real-time level has been chosen, a *wcet* fault cannot occur.

- The nearest deadline exception has to be considered only if executing a mandatory component, but for a different reason: the dynamic slack time calculation itself ensures that any slack interval scheduled for the non-real-time level will be finished *before* any task can miss its deadline. In order to determine the next potential deadline exception, the first-level scheduler maintains an *ordered* list of pending active deadlines. This list is updated each time a task is released and each time a task ends its current release.

As presented above in Figure 7.5, each time a timing fault is detected inside the `rt_timer_handler` function, the `rt_raise_exception` function is invoked, indicating both the faulty task and type of exception produced. The support for detecting deadline exceptions is activated and deactivated by means of a global variable, whose value can be set by calling the `rt_deadline_exception_on` function. Inside the `rt_raise_exception`, the actual handling mechanism defined for each type of exception is different: a *wcet* exception produces an automatic reset of the faulty task's context, the release of all its occupied resources (e.g., locked mutexes) and the activation of the task's user-defined exception handler, if any. A deadline exception makes the application stop by invoking the `rt_app_stop` function.

7.3 The Linux Level

As described above in Section 7.1, the FRTL non-real-time level is implemented as a regular user-level Linux process, called the *optional server* process. This section presents the main design and implementation issues related to this process. These issues are basically aimed towards two major goals: first, to synchronize the second-level scheduler with the first-level scheduler in such a way that both schedulers agree on how to dynamically share the processor. And second, to achieve the highest possible reliability and predictability at the Linux user level, without any modification of the Linux kernel.

7.3.1 Internal Design of the Second-Level Scheduler

The framework software architecture proposes that the first-level and second-level schedulers dynamically share the processor according to three general rules:

1) The first-level scheduler determines when to execute and when to preempt the non-real-time level. This is performed by calculating the *slack intervals* or intervals of execution of the non-real-time level.

2) After programming a slack interval for the non-real-time level, the first-level scheduler is committed not to interrupting this level in the middle the interval. This property allows the second-level scheduler to execute without interruption within such intervals. The advantage of this approach is that the second-level scheduler can schedule the application optional components being almost unaware of the existence of the real-time level. In fact, this scheduler does not need to know anything about *tasks* or *mandatory components*, at least in theory[5].

3) The second-level scheduler is committed to finishing each slack interval in a known state *before* being actually preempted by the first-level scheduler.

While the first and second rules are enforced by the first-level scheduler, the third rule is entirely under the responsibility of the second-level scheduler. The scheduler is supposed to be suspended by synchronously reading the *sched fifo* at the beginning of each slack interval. In order to achieve such behaviour, the scheduler is provided with a special design, by which each of the phases inside the scheduler's endless main loop (see Section 7.1.2) is actually a *time-bounded phase*. The term *time-bounded* means that each phase has a pre-computed *wcet*. At run time, there is a *checkpoint* before entering each phase. At that checkpoint, the phase's *wcet* is compared with the remaining slack interval *before* entering the phase. If the phase cannot be completely executed in the remaining slack interval, then that phase is not even started and the scheduler immediately suspends itself at the *sched fifo* until the next interval.

This design consisting of time-bounded phases also has to be applied to the final phase, in which the second-level scheduler selects and executes the ready optional component versions. This is done by means of the so-called *timed execution*. This is a method by which the scheduler executes a particular version for a *certain* interval of time, which is always smaller than the remaining slack interval. The utility of this method is primarily to permit the scheduler to finish this final phase before being preempted. However, the timed execution method is actually a general scheme that may be used for implementing very sophisticated scheduling policies. For example, some RTAI policies discussed in Section 2.2 are able to build a calendar or plan of task execution, in which each task is given a limited time, so that a task quality value is maximized. Such policies could use the timed execution to give each task the exact time stated in the calendar.

Overall, by following this design, the second-level scheduler can always suspend itself on time, at the beginning of its loop, before Linux is preempted by the real-time level. This has only one exception: the scheduler needs some initial time at each slack interval in order to read the message indicating the slack interval, to consult the current time and to check if

[5] The actual second-level scheduling policy may require information about the application tasks and their structure. If so, this information can be added to the data associated to each optional component when this component is created (see Section 4.6).

its first phase can be executed. In other words, the scheduler cannot suspend itself before reaching its first checkpoint. This case is therefore solved by the first-level scheduler: if the slack interval to be programmed for the non-real-time level is not long enough to 1) allow the optional server process to start running, and 2) allow the second-level scheduler to achieve its first checkpoint, then the first-level scheduler does not send the message which wakes up the second-level scheduler. In this case, although Linux is executed during this (small) interval, the optional server process is not awakened.

7.3.2 Synchronization Between the Two Schedulers

Since the first-level and second-level schedulers work in a cooperative manner, they need to share information. This sharing has to be carefully designed, since it has to be done in such a way that data consistency is guaranteed and the particular run-time behaviour of each scheduler is maintained. In FRTL, the schedulers actually exchange information by means of two mechanisms, each of which has a different synchronization method:

a) Message passing. The first-level scheduler uses two RT-FIFOes (the *sched fifo* and the *events fifo*) for sending messages to the second-level scheduler. In this case, synchronization is achieved by the RT-FIFO mechanism itself: from the real-time side, RT-FIFOes are wait-free structures, and thus accessing them never causes delays to the first-level scheduler. From the Linux side, the Linux kernel manages them like other special files (such as regular *fifos* or *pipes*), internally protecting the consistency of the data stored inside them. Reading an RT-FIFO from a Linux process may be done both synchronously and asynchronously, and the associated delay for the process cannot be strictly bounded, but it is usually small.

b) Sharing memory. The two schedulers share common memory variables, such as the structures storing the information about the application mutexes[6]. Synchronization in this case is obtained by using the concept of slack interval. As explained in the last section, the slack interval concept ensures an uninterrupted execution interval of the optional server process at the Linux level. This provides the second-level scheduler with the ability to safely share memory variables with the first-level scheduler without using an explicit synchronization mechanism between them, just by ensuring that the access is completed in the current slack interval.

7.3.3 Implementation Details

The support provided by the second-level scheduler inside the optional server process has been implemented as efficiently as possible, having special care of achieving the scheduler's expected run-time behaviour. This section discusses the most relevant implementation details,

[6]The particular case of sharing the mutex information needs only synchronization for reading access since, according to the PCP and CSP algorithms (see Section 5.3.3 and 5.4.3), the mutex status is not actually changed inside the lock or unlock functions at the non-real-time level.

including the mechanism used for the multi-threading support and the set of Linux system calls invoked by the scheduler.

Multi-Thread Support

In the early stages of the optional server process development, the support for running multiple threads inside the process was implemented by using the *pthreads* (POSIX threads) library available in Linux. The POSIX threads support creates threads at the *kernel level*, that is, each thread is an known entity in the Linux kernel and is scheduled by the Linux schedule. This kind of support is appropriate when Linux scheduling policy is compatible with the requirements of the multi-threaded application. However, when the multi-threaded application requires its own scheduler (for special policies and custom synchronization mechanisms) and needs to create, destroy or restart threads frequently, then the POSIX thread support becomes inefficient and tricky to use. In particular, the following problems were detected:

- Context switches among *pthreads* are not explicitly provided, nor are functions for suspending and waking up threads. This complicates the design of a multi-threaded process which requires that only *one* thread be running at a time. An explicit context switch can only be implemented by using a complex mechanism involving special synchronization primitives (mutexes *and* variable conditions), and fiddling with thread priorities in order to ensure a proper sequence in the suspension or awakening of the threads during the switch. Explicit *cancellation points* must also be included within the thread's code in order to allow the thread to be asynchronously killed. This is required in FRTL if, for example, a version runs out of time in its *timed execution*.
- Inside such a multi-threaded process, a special *pthread* must be created to run the scheduler code. That thread is in charge of creating the other threads and performing the explicit context switches required to execute them. All this requires invoking several Linux system calls, hence adding a great deal of overhead and unpredictability to the scheduler.
- Tests showed how the system call to create a new *pthread* (`pthread_create`) takes about 20 to 50 milliseconds on a AMD 333 Mhz K6-2 computer with low load (this cost gets much worse if the system is loaded with many processes). This is an intolerable overhead for the FRTL system, since that time may easily be the same order as many slack intervals in a typical application.

These problems have led to a completely different implementation of the thread support within the optional server process. This support has been implemented by using the traditional Unix `setjmp` and `longjmp` facilities[7], which respectively allow saving the context of the running process at one execution point and restoring it later when needed. By means of these

[7]The optional server process actually uses a modern version of these facilities, called `sigsetjmp` and `siglongjmp`, which add information about the blocked signals to the context to be saved or restored.

two functions, it has been possible to implement a user-level thread facility, that is, a mechanism that achieves a multi-threaded behaviour inside a process which is still considered as single-thread by the Linux kernel.

In the support implemented here, a thread is created for every version of each optional component at the initialization stage of the second-level scheduler. After creating a thread, its initial context is stored in a special structure associated to the corresponding version. At run time, when a version has to be executed, the process saves the context of the second-level scheduler at that point and loads the version's initial context (with a cost of a few microseconds), effectively producing a context switch to that version. When the version finishes (or alternatively when the timer produces an interrupt indicating the end of the version's scheduled time) the scheduler saved context is restored, then producing another context switch to the point at which the scheduler began executing the version. If that particular version must be executed again, the scheduler simply has to restore its initial context again.

Overall, although this mechanism has to be very carefully implemented, it has interesting advantages over the *pthread* support in the particular case of the optional server process, First, its overhead is practically nil, since the `setjmp` and `longjmp` are *not* proper system calls (they are library functions). Second, from the Linux scheduler point of view, the optional server process is a single-threaded process (running at the highest priority all the time), and hence the Linux internal handling of threads does not affect the optional process behaviour. This solution is thus more straightforward, efficient and predictable. The only drawback of this mechanism is that it is not possible to *suspend* a version in order to be *continued* afterwards. This implies that any version which cannot be completely executed in a particular slack interval has to be killed; this version has to be started again at the next slack interval if its component is still ready and the scheduler decides to execute the version again.

Set of Linux System Calls Invoked by the Scheduler

The second-level scheduler uses the least possible number of Linux system calls, in order to minimize the unpredictable effect of the Linux kernel over the optional server process. The list of utilized system calls include the following: `gettimeofday`, in order to know the current system time; `setitimer`, in order to program (and to stop) the Linux timer which signals the end of the version's scheduled execution time; and `sigaction`, in order to set the function to execute when the signal produced by the timer (`SIGALRM`) is delivered to the process.

Finally, it is worth noting that, although the system calls actually invoked from the optional server process are the minimum possible, and although that process runs at the Linux highest priority, it may still suffer interference from the Linux kernel, specially due to the execution of Linux interrupt handlers. For this reason, the actual coding of the second-level scheduler has introduced some *security intervals*. These intervals are subtracted from the available slack at each slack interval. The security intervals are placed to compensate for the Linux interference due to hardware interrupt handling and for potential delays in the timer interrupt mechanism

and the SIGALRM signal delivery. Conservative decisions of this type inside the second-level scheduler permits increasing the reliability of the non-real-time level, while leaving the Linux kernel code unchanged.

7.4 System Tools

Debugging an application normally involves keeping track of the application's steps in order to find out exactly where the error is produced in the code. This tracking can be done by simply printing data to the screen as the application executes or by using more sophisticated debug programs (such as the gdb tool available in Linux). Such tools allow the designer to put breakpoints in the application's source code, to execute the application step by step, to consult the values of application's internal variables while running this application, etc. However, all this cannot be used for for debugging the FRTL real-time level, since it runs inside the Linux kernel address space, in privileged mode; in these conditions, standard debug programs are not available. Furthermore, storing information to a disk file is also not possible (since it produces unbounded latencies), and even the existing facility for printing data to the console (called printk) is rather limited.

These difficulties can also affect the system monitoring or *profiling*. This activity involves the collection of run-time performance data about the application tasks and the run-time system and its later analysis. The information derived from this analysis is very important in a real-time system, since it allows the designer to find out if the system is running as expected (i.e., tasks are meeting their deadlines, interrupts are being produced at the right rate, the kernel is not producing an excessive overhead, etc.). If the profiling is done by software methods[8], the limitations stated in the previous paragraph also apply here, since the information stored at the real-time level cannot be stored in disk or appropriately presented on the screen.

In order to overcome these limitations, the FRTL system incorporates an integrated debug and profiling mechanism which is implemented in shared memory. This mechanism is decoupled in two parts: an on-line collection of data and an off-line analysis of these data. These parts are now detailed.

7.4.1 Collection of Run-Time Data

Each time the first-level scheduler executes in response to a hardware timer interrupt or a system call being invoked, part of its code is devoted to collecting run-time information. The collected information is stored in main memory, in an area which can be later read by the Linux process(es) performing the analysis of this information (see below). The collection mechanism has been designed in order to produce the minimum overhead possible. This is

[8]Profiling can be done by hardware, software or hybrid methods. Hardware monitoring is performed by attaching special measurement devices to the computer's processor, buses, etc. This type of profiling is very accurate but difficult to use. Software monitoring is done by means of a piece of software, which can be either a special task or a part of the run-time system. Hybrid monitoring is a combination of hardware and software methods.

achieved by preventing the scheduler from performing complex statistical calculations. The run-time information is *directly* extracted from scheduler's variables, and, therefore, the cost of this mechanism is actually the cost of storing a few bytes in main memory, each time the scheduler executes. All the required calculations are performed off line by the analysis programs.

The run-time information recorded by the scheduler may be classified in the following groups:

a) Static information. Since tasks and their components are created at the real-time level, the tools analyzing the run-time data need to know about the number of application tasks and their structure (the number and particular arrangement of components inside tasks). This is called *static* information. It is collected once by the scheduler, after all tasks have been created and just before starting to run the application.

b) Trace information. In a general sense, the trace information of a process allows for a later reproduction of the execution steps of this process. In a real-time application, the trace information normally includes events such as the moments at which tasks are released, task deadlines, execution intervals of each task, etc. In general, this is the information normally required for depicting a *chronogram* of the application's execution.

The first-level scheduler collects trace information about the execution of both the application tasks/components and its own, by means of recording a well-defined set of significant events. Table 7.1 presents the actual types of events that the first-level scheduler keeps track of. Although the scheduler records all the events belonging to each type, the overhead involved is actually very low. All the kernel-related events, plus the task-release event are collected by the following simple method. Each time the scheduler executes, it consults the system time in two instants: at the beginning of its code, in the `rt_update_system` function (t_1), and at the end of its code, in the `rt_dispatch` function (t_2). Then, interval $[t_1, t_2]$ is actually the cost of processing the event and interval from t_2 in one event to t_1 in the following event is the execution interval of the task which was selected in t_1.

c) Profiling information. Most of the profiling and statistical information about the application and kernel execution is computed by the off-line programs described below, by means of analyzing the *trace* information. However, some of this information is collected by the first-level scheduler, since it can provide it directly from its variables without any extra calculation. This class of information includes the following types: the total execution time consumed by a task mandatory component, the amount of slack time used for running a task optional component, and the response time of a task at each release.

d) Debug information. This basically includes strings and values of variables. Its primary purpose is to debug the first-level scheduler code. This class of information is not actu-

ally analyzed by the tools described below; it is just stored to a particular file for being read by the kernel programmer.

In order to facilitate the ulterior analysis of the run-time data, every piece of information collected by the first-level scheduler is actually stored in a common structure allocated in shared memory. In this sense, the memory area used by the profiling mechanism can be seen as a linear array of such structures. This structure is basically formed by a *time stamping* (i.e., the recording of the time at which the information is collected), a descriptor indicating the type of information, and the particular data associated to this type. Related to this, the amount of information recorded by the first-level scheduler depends on the actual size of the shared memory area reserved for this mechanism. Since the retrieval of the profiling data is not done in parallel with the application execution but rather once the application has been stopped, the size of shared memory devoted to this mechanism puts a limit on the amount of information that can be collected each time the application is executed. In the current version of FRTL, there is a limit of four megabytes in the amount of memory that can be shared between the system levels. The profiling mechanism normally uses two out of these four, leaving the other two to be used as "the storage" for the application (see Section 7.1.3). These two megabytes permit storing about 90,000 events per execution.

7.4.2 Off-line Analysis of the Collected Data

The profiling and debug information recorded by the first-level scheduler at run time is processed after the application has been stopped by two special FRTL tools. The first tool is called `wtrace`. This tool is responsible for both reading the shared memory area containing the profiling and debug information and generating three files with the retrieved data:

a) *quivi file*. The `wtrace` generates this file by reading the trace data stored in shared memory, preprocessing these data, and writing them in a special syntax. This is the syntax expected by the `quivi` program, a graphical display tool for visualizing execution chronograms. In the *quivi file*, all the events related to the kernel are considered as execution intervals for a special *kernel task* which has the highest priority. The task-related events are associated to their corresponding tasks with their corresponding priorities. Thus, the quivi chronogram depicts the execution of the FRTL first-level scheduler, the application mandatory components and the non-real-time level.

b) *debug file*. The items of information corresponding to debug events are stored in this file in text format. Thus, this file can be directly read by the programmer.

c) *raw file*. All the trace and profiling information found in shared memory is written to this file. The purpose of this file is to save all the useful information stored in memory for further processing, which is carried out by the `atrace` analysis tool.

By processing the *raw file*, the `atrace` tool can analyze several execution aspects of the system. This program is basically in charge of calculating statistical values of the trace and

TASK EVENTS	
Type	**Description**
Activation	Release of an application task. This event includes the moment at which the task is released and the deadline relative to this release time.
Execution	Interval of execution of a task. This interval is defined by a starting and an ending time.
Local *wcet* exception	This is a local *wcet* exception produced by a mandatory component inside a task. This event is detected inside the `rt_gain_time_update` function.
Global *wcet* exception	This signals a task completely consuming all its *wcet*.
Deadline exception	When the deadline exception support is activated, this event is reported when a task misses one of its deadlines.

KERNEL EVENTS	
Type	**Description**
Timer interrupt	Cost involved in processing one timer interrupt, corresponding to either a "task release" or an "end-of-slack" time event.
Exception interrupt	Cost involved in processing one timer interrupt corresponding to a *wcet* or a deadline exception.
Gain time	Cost of executing a single invocation of the `rt_gain_time_update` implicit system call.
Run optional	Idem for the `rt_run_optional` implicit system call.
End task	Idem for the `rt_task_end` implicit system call.
End handler	Idem for the `rt_handler_end` implicit system call.
Mutex lock	Idem for the `rt_mutex_lock` system call.
Mutex unlock	Idem for the `rt_mutex_unlock` system call.
Execute dynamic	Idem for the `rt_task_run_dynamic_mandatory` system call.

Table 7.1: Trace information collected by the first-level scheduler. This table classifies the trace data into two groups: information collected for each task, and information collected for the events being executed by the scheduler.

profiling information collected by the first-level scheduler. In particular, `atrace` computes the number of instances, and the average, minimum, maximum and variance values of the following data: *each* kernel event defined in Table 7.1, the actual execution consumption of each mandatory component of each task and the actual slack time assigned to each optional component of each task. This information, which is presented to the screen in a tabular form, is very useful to the application developer, since it clearly shows aspects such as mandatory components exceeding their *wcet*s, tasks exceeding their deadlines, the maximum overhead involved in processing each kernel event, etc. In fact, the measurements of the kernel overhead presented in Chapter 8 have been obtained by means of the information provided by this tool.

7.5 Summary and Contributions

The purpose of this chapter has been to show that the framework's theoretical proposals can be achieved in a real system. The proposed run-time system, called FRTL, has taken advantage of the RT-Linux basic functionality in order to feature sophisticated capabilities for building flexible real-time applications. Furthermore, the careful design and implementation of FRTL have succeeded in producing an RTOS which is as predictable and efficient as the original RT-Linux.

The FRTL system is actually formed by a two-level run-time system and a set of off-line analysis tools. The run-time system *real-time level* has been implemented by modifying the RT-Linux real-time support. In this level, the implemented *first-level scheduler* offers the task model and real-time services proposed by the framework. This includes the definition of tasks, dynamic mandatory components and exception handlers, the specific versions of the fixed priority preemptive policy and the slack stealing algorithm, access to PCP and CSP mutexes, the detection and handling of timing exceptions, etc. Besides presenting how these features have been implemented, this chapter has detailed the internal design of the real-time level. This design is based on two paradigms: a layered structure and a homogeneous approach for dealing with every run-time system event. The real-time level has been structured in a set of layers, each of which is based on the functionality provided by the previous layer(s). This presents several advantages, such as the incremental building of the system, the isolation of errors, the more direct porting to distinct hardware platforms, etc. The homogeneous handling of system events proposes designing every hardware interrupt handler or system call provided by the first-level scheduler in a common three-step scheme (system update, particular actions dealing with the event produced and rescheduling). This allows the timing behaviour of the scheduler to be more homogeneous (which effectively facilitates its timing analysis) and also favors scalability, since new features can be added more straightforwardly

The FRTL *non-real-time level* has been implemented inside a user-level Linux process, called *optional server*. The optional server process is a multi-threaded process running at the Linux highest priority, which ensures that every time the first-level scheduler runs Linux, this process gets the CPU in preference over any other user process. Inside the optional server process, the *second-level scheduler* provides the application optional components and their

versions with a utility-based scheduling policy and access to shared memory through the use of mutexes. This scheduler has been internally designed in order to meet two requirements. The first requirement is to perfectly synchronize its execution with the execution of the real-time level; this can be expressed (in practical terms) as getting the second-level scheduler to be in a specific execution state at the end of every slack interval, just before being preempted by the real-time level (this ensures the scheduler will start running from this particular point, at the beginning of the next slack interval). This requirement is met by structuring the scheduler's main loop in a set of time-bounded phases, in such a way that the scheduler voluntarily interrupts its execution *before* starting a phase which may not be able to finish before being preempted. The second requirement is to make minimum use of Linux facilities (i.e., system calls), in order to limit the unbounded overhead produced by the Linux kernel and, specially, to circumvent the possibility of the optional server process being suspended by Linux.

Overall, the design and implementation of the non-real-time level inside a Linux process has achieved three main goals: first, this level features all the requirements and services imposed by the framework, including the synchronization with the first-level scheduler (which allows for the dynamic sharing of the processor between both system levels). Second, the non-real-time level has achieved an appropriate degree of predictability and reliability while leaving the Linux kernel code unchanged. This permits a direct porting of the FRTL non-real-time level to future versions of Linux. And third, the implementation of the second-level scheduler at the user level facilitates changing the scheduler policy, if required by the application designer.

Finally, the set of FRTL *system tools* are actually a part of the its integrated debug and profiling mechanism. This mechanism uses a portion of the shared memory area for storing debug and profiling data at run time (this information is collected by the first-level scheduler). After stopping the application, the system tools retrieve this information from memory and write it in a set of files, for further processing. This processing includes the display of the application execution chronogram, the writing of the debug information in a text format and the statistical analysis of several timing attributes of both the application tasks and the kernel. By means of this analysis, it is possible to calculate the minimum, maximum, average and variance of every overhead source of the FRTL real-time level. The analysis also includes information about task and component *wcet* exceptions, task response times, slack time consumption, etc.

8

Complete Feasibility Analysis

Any run-time support system or kernel intending to be useful in building hard real-time applications must be provided with information about its internal overhead. This overhead has to be taken into account when testing the application, since it can affect the timing behaviour and the schedulability conditions of tasks. The typical approach of many commercial RTOSs is to provide time measurements of different system parameters for specific hardware platforms. Typical measurements include the costs of context switches, system calls, etc. However, by only providing this kind of information, the kernel is still offered as a black box. Thus, significant aspects of the kernel internal design actually affecting its timing behaviour (such as private kernel tasks, hidden synchronization mechanisms or sections of code executed with interrupts disabled) cannot be determined. Some RTOSs solve this lack of information by providing the kernel source code to the application designer, usually at a huge price. The designer then has to analyze and measure the application and the kernel, in order to verify the feasibility conditions of tasks. This thesis proposes an alternative approach, based on providing a *complete* feasibility test to the designer. The test is said to be complete because it incorporates the exact timing behaviour of the kernel. As a result, the designer can check the feasibility of the complete system by simply adding the timing information about tasks to the test. This chapter presents a complete feasibility test of the FRTL system and actual run-time measurements of its overhead on different hardware configurations. The system is thus proven to be both predictable and efficient.

8.1 Introduction

In several hard real-time systems, feasibility analysis is the typical method for verifying that the application tasks can always meet their timing requirements. In the context of fixed priority preemptive systems, feasibility is checked by means of an off-line feasibility test that demonstrates if a particular application (or task set) is able to meet its timing constraints.

Feasibility tests can be divided into two categories, depending on their coverage [Aud95]: *sufficient* tests and *sufficient and necessary* tests. The response of the latter tests is exact, in the sense that an application can be scheduled to meet its constraints (i.e., it is feasible or schedulable) if, and only if, the test is satisfactory. However, for realistic task models, these exact tests have an excessive complexity (this problem is NP-Complete in the general case). Hence, their applicability is limited. Sufficient tests have polynomial costs but they are more pessimistic, in the sense that they may reject applications that are schedulable in practice.

In general, feasibility tests study ideal systems, assuming simplifications that do not hold in real applications. The most important limitation of such tests is not to consider the effect of the run-time support system or kernel. The execution of the kernel code produces what is called *overhead*, an extra processor load which is generally related to, but not included in, the application tasks' code. Nevertheless, when this kernel code is actually executed, typically as a result of either a system call invoked by the running task or a hardware interrupt, it produces a real interference to the application tasks. Since this interference obviously provokes a certain impact in the tasks' feasibility conditions, it must be carefully studied and reflected in the test. In this sense, the present chapter distinguishes between *theoretical* tests, which do not consider any run-time overhead, and *complete* tests, which incorporate all kinds of execution sources that cause interference to the tasks. Both theoretical and complete tests can be either exact or only sufficient, depending on their coverage, as explained above.

From the viewpoint of feasibility, introducing the kernel interference in the test is not so straightforward. This code is executed sometimes as a result of a task action (e.g., a system call being invoked) and sometimes because of some internal reasons (e.g., a timer interrupt used for updating the system time), making it difficult to exactly characterize when the interference is produced. The exact interference the kernel produces at each moment and the subset of affected application tasks are also difficult to determine, since several implementation details affect the kernel timing behaviour. Examples of such details are: mechanism(s) used for internal synchronization, maximum amount of hardware interrupts per time unit, the cost of handling these interrupts, execution of internal tasks for system maintenance processes, sections of code executed with hardware interrupts disabled, etc. Hence, how the kernel has been designed and implemented become very important issues for its timing characterization.

This chapter introduces a sufficient, complete feasibility test of the FRTL run-time support system, explicitly including all its particular sources of overhead. These sources are introduced in the test as symbolic constants, because their actual costs depend on the particular hardware the system is running on. In addition to the generic test, actual measurements of these symbolic constants are also presented. By means of these real measurements, the FRTL system is

also shown to produce a reasonably low overhead. The system will therefore be proven to be both predictable and efficient.

The contents of the chapter are organized as follows: the next section presents the relevant previous work in the field of feasibility tests for fixed priority preemptive systems. Section 8.3 adapts this work for the particular computational model proposed in this thesis. Then, Section 8.4 presents a timing characterization of the FRTL run-time system, and introduces each significant factor to the test. This section then presents the complete feasibility test for the FRTL system. Section 8.5 shows actual measurements of the overhead factors introduced in the test for particular hardware platforms. Finally, Section 8.6 concludes and presents the contributions of this chapter.

8.2 Previous Work

This section presents the typical feasibility test for general real-time systems using fixed priority preemptive scheduling, which is based on the computation of the *worst-case response time* of each task [Jos86, Aud90]. The test assumes a task model in which the following conditions hold:

- The task set is ordered according to task priorities, with the subindex of each task τ_i denoting its priority (i).
- Each task τ_i is periodic or sporadic and does not arrive at the system more frequently than a known value (either its period or minimum inter-arrival rate), denoted by T_i.
- Each task τ_i has a known *wcet*, denoted by C_i.
- Each task τ_i can be blocked by lower priority tasks for, at most, the duration of its worst-case blocking time, denoted by B_i.
- Each task τ_i is released as soon as it arrives to the system and does not voluntarily suspend itself.
- All tasks are initially released at the same time.
- Context switches between tasks and kernel overhead are assumed to be zero.

Equations 8.1 and 8.2 show the feasibility test for such a model. The former equation computes the worst-case response time of application task τ_i. This time is defined as the longest time between when the task arrives to the system and when it completes, and its calculation is based on formalizing the run-time situation in which this task suffers the worst-case (highest) interference.

$$R_i^{n+1} = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{8.1}$$

As the factor R_i appears in both sides of the equation, the equation is typically solved by means of a recurrence relation that converges to a concrete value of R_i. Once calculated, the feasibility of task τ_i can be easily evaluated by checking this value against τ_i's deadline (D_i), as stated in equation 8.1.

$$\text{Task } \tau_i \text{ is feasible iif} \quad R_i \leq D_i \tag{8.2}$$

Since equation 8.1 computes the *exact* value of R_i for the task model presented above, the resulting test is both sufficient and necessary. The same test can be used if tasks are allowed to feature arbitrary initial offsets, but then it turns into an only-sufficient test.

8.3 Theoretical Test Adapted for the Framework

The feasibility test expressed by Equations 8.1 and 8.2 cannot be directly utilized for analyzing applications implemented according to this framework (even not considering the FRTL kernel's overhead) because the framework computational model is different from the model assumed above. The purpose of this section is to adapt this theoretical test to the framework task model. In the following section, the new test will incorporate the kernel overhead.

In this framework, each task τ_i is potentially made up of mandatory and optional components. According to both the definition of each type of component and the fixed priority policy applied by the system, only mandatory components of tasks have to be guaranteed, and they only suffer interference from other mandatory components. Because of this, each task's *wcet* which has to be introduced to the test is calculated as the sum of the *wcet*s corresponding to the task's mandatory components. This calculation, initially expressed in terms of the formal task model by Equation 4.1 (in page 39), has been rewritten here for the sake of simplicity and is shown in Equation 8.3.

$$\forall \tau_i \in \mathcal{A}, \quad C_i = \sum_{\forall k \in man(i)} c_{ik1} \tag{8.3}$$

where:

- The term $man(i)$ denotes the set of mandatory components of task i:

$$man(i) = \{k \, : \, \gamma_{ik} \in \Gamma_i \wedge Y_{ik} \in \{\mathcal{M}\}\}$$

- The term c_{ik1} represents the *wcet* of the first (and unique) version of mandatory component γ_{ik}, that is, γ_{ik}'s *wcet*.

Therefore, by including this particular calculation of each term C_i into Equation 8.1, the test above guarantees the execution of all tasks' mandatory components.

The execution of mandatory *wcet* handlers also has to be introduced in the test for two reasons. First, mandatory handlers are application-defined actions whose execution has to be

guaranteed. And second, the execution of mandatory handlers actually causes interference to the application mandatory components. Given that each handler is associated to a given application task, the obvious method of considering this handler in the test would be to incorporate its *wcet* in the computation of its task's *wcet* (in Equation 8.3). However, this simple method is not valid here, because both the handler and its associated task may be assigned different priorities. As a result, the effect of executing mandatory handlers has to be directly considered in the calculation of R_i. This is now presented.

In the worst case, each task produces a *wcet* exception at each release, producing the execution of its handler. Then, the equation computing R_i should contain the execution of all *mandatory* handlers defined with a priority greater than or equal to i. The key issue here is how to characterize the release of handlers, since they behave differently than their associated tasks. A mandatory handler is released exactly when its related task commits a *wcet* exception. Although the task is released periodically, it may produce a timing exception at a different moment at each release. That is, the exception is not a *pure* periodic event. In particular, if task i is released at time t, it may cause a *wcet* exception at any moment between C_i and D_i, depending on the actual interference which task i suffers from higher priority tasks and handlers. As a result, mandatory handlers are introduced to the test as *sporadic tasks*, which are released by their related tasks. This is shown in Equation 8.4.

$$R_i^{n+1} = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j + \sum_{\forall k \in hand(i)} \left\lceil \frac{R_i^n}{T_k^h} \right\rceil C_k^h \quad (8.4)$$

where:

- $hand(i)$ is the set of tasks having mandatory handlers with priorities higher than or equal to i:

$$hand(i) = \{k : \tau_k \in \mathcal{A}, Y_k^h \in \{\mathcal{M}\} \wedge P_k^h \leq i\}$$

- T_k^h is the minimum inter-arrival rate of the handler associated to task k (this is calculated below).
- C_k^h is the *wcet* of the mandatory handler associated to task k.

Therefore, mandatory handlers are considered as sporadic tasks with a minimum inter-arrival rate of T_k^h, calculated below in Equation 8.5. This minimum separation between consecutive releases of the handler is produced in the following scenario, depicted in Figure 8.1. Consider a task τ_k producing a *wcet* exception in two consecutive releases. The first release of the task is produced at time t. In this release, the task produces an exception in the last possible moment, that is, when reaching its deadline at $t + D_k$. In the second release, produced at time $t + T_k$, the task commits the exception in the earliest possible moment, that is, after running for C_k time units with no interference from any higher priority tasks or handlers. This second exception is thus produced at time $t + T_k + C_k$. The minimum inter-arrival rate of the handler

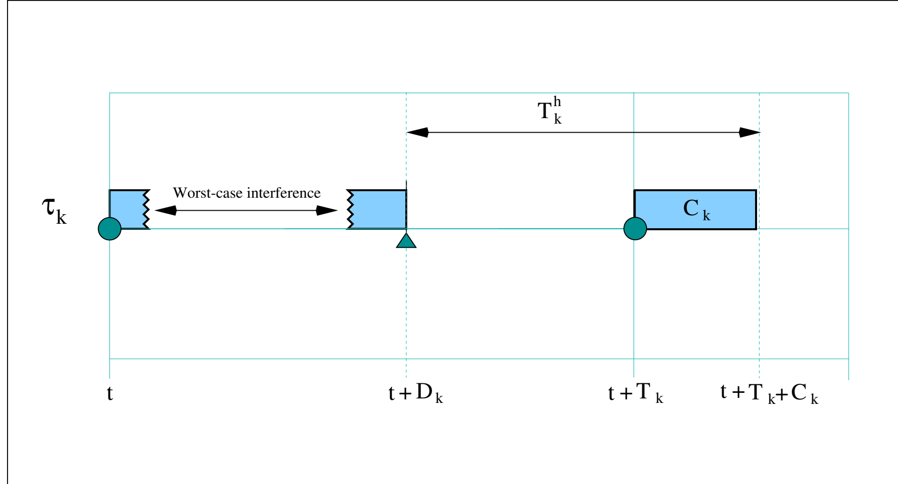


Figure 8.1: Calculation of the minimum-inter arrival rate of a task mandatory handler.

release can thus be calculated by subtracting the two extreme exception times, $t + T_k + C_k$ and $t + D_k$. The result is shown in Equation 8.5.

$$T_k^h = T_k + C_k - D_k \tag{8.5}$$

In this section, the theoretical feasibility test adapted for this particular framework has been presented. This test is formulated by combining the new computation of R_i in Equation 8.4 with the test in Equation 8.2. This theoretical test is turned into a complete test in the next section, by incorporating the timing characterization of the FRTL kernel.

8.4 Developing a Complete Test

This section is focussed on upgrading the framework test presented above by introducing the overhead factors derived from the specific features supported by the FRTL system. In the following discussion, and for the sake of clarity, the kernel will first be assumed to be executed with the hardware interrupt disabled. This is equivalent to considering that the kernel can be preempted at any time. By making this assumption, the overhead directly related to a particular task i may be considered as being executed at that task's priority. The assumption thus permits considering this overhead in the test by directly adding it to the task's *wcet*, C_i. Thus, the following sections calculate a new factor for each task i, containing the theoretical C_i plus the kernel overhead related to this task. This new factor is named *real wcet* and is denoted by C_i^{real}. After considering each source of kernel overhead, the assumption of the kernel being executed with the interrupts enabled is removed at the end of this section, and then the test's final equations are presented.

All the feasibility discussion presented here, as well as the actual measurements taken from the implemented system, take into consideration only the system real-time level. Since the non-real-time level completely runs in slack time, in such a way that it cannot affect the feasibility conditions of the real-time level, its effects do not have to be included in the test. For this reason, the term *kernel* refers hereafter to the FRTL first-level scheduler.

8.4.1 Kernel Design Issues

This section summarizes some of the design aspects of the FRTL system, already presented in Chapter 7, which affect the kernel timing behaviour:

- The first-level scheduler code is executed with the hardware interrupts disabled, as a form of internal synchronization.
- The real-time level is *interrupt driven*, as opposed to being tick driven. This means that the system is not interrupted at each clock tick, but only at certain instants when significant time events are produced.
- The first-level scheduler is internally designed in order to have a homogeneous three-step structure for processing all the system events. This structure is formed by the system *update*, the event's particular *processing* and the task *rescheduling*. In this structure, in spite of the actual event being produced, the first and last steps are common. These steps are executed by calling the `rt_update` and `rt_schedule` functions, respectively.

These three characteristics will be referenced when necessary in the following discussion.

8.4.2 Inclusion of Interrupt Handling

The only hardware interrupt utilized by the FRTL system is the timer interrupt, which is used to signal the three kinds of time events required by the first-level scheduler: task releases, end-of-slack instants and *wcet* exceptions. These three events are handled by the first-level scheduler `rt_timer_handler` function, as explained in Section 7.2.5. The current section is centered on discussing the two former time events, while timing exceptions are later studied in Section 8.4.5.

Task Release

Every application task is released as a result of a timer interrupt. In the worst-case, each release of each task is signaled by a different interrupt, which has to be separately processed, producing the corresponding overhead[1]. When such an interrupt occurs, an automatic context switch is produced from the running rt-task to the `rt_timer_handler` function. The actual

[1] Within any case other than the worst case, a hardware interrupt may signal the release of several tasks, and, hence, the cost of processing the interrupt is then shared by all these tasks.

releasing of the task is then produced inside the `rt_update` function. Assuming that no exception is produced at the same time as the task release, the function then skips all the code until the `rt_schedule` call. Inside this function, the next task to be executed is selected. The worst case overhead is produced when the newly released task is the same one chosen for execution, because, in this case, an explicit context switch to this task has to be performed. The cost of a task release can therefore be expressed as:

$$C_{release} = (2 \times C_{switch}) + C_{update} + C_{sched} \tag{8.6}$$

where C_{switch} is the cost of a context switch and C_{update} and C_{sched} denote the worst-case costs of `rt_update` and `rt_schedule`, respectively.

Since all tasks are potentially released at different times, this release cost executed by the kernel must be added to the *real wcet* of each task i. Equation 8.7 introduces this calculation.

$$C_i^{real} = C_{release} + \sum_{\forall k \in man(i)} c_{ik} \tag{8.7}$$

End of Slack

The end-of-slack event is produced when Linux has been executing during a slack interval and the hardware timer signals the end of that interval, then producing the execution of the `rt_timer_handler` function. In this case, this function first calls `rt_update` and then `rt_schedule`. It is precisely inside this latter function where the exhaustion of the slack interval is detected and treated. The interesting feature about this event is that it is *not* necessary to include its cost in the feasibility test: since this event is produced only when Linux has been executing in slack time, then this cost may be *subtracted* from the slack interval itself. In other words, the code dealing with this event can be considered as being executed in slack time, thus not being part of the task's *wcet*. This can be done at run time by means of subtracting this cost from the duration of the slack interval, before this interval is scheduled, making the hardware timer interrupt earlier.

8.4.3 Inclusion of Implicit System Calls

In FRTL, each application task is executed by an rt-task that internally executes the function `rt_task_runner`, as explained in Section 7.2.5. Inside this function, three *implicit* system calls are invoked: `rt_gain_time_update` is called after executing each mandatory component, `rt_run_optional` is called each time the task has to execute an optional component (or more precisely, a contiguous list of optional components) and `rt_task_end` is called at the end of the task release, when all its components have been scheduled. As system calls, these three functions are internally structured in the usual steps of: a call to `rt_update`, their specific code and a call to `rt_schedule`.

The `rt_gain_time_update` cost contains the execution of these two common functions plus the cost of the code actually performing the gain-time update; this cost is denoted by

C_{gain_code}. There is no context switch involved in this system call, since it is directly invoked by the running task and it cannot produce a change in the task selected to be run next. Its cost can thus be expressed in the following terms:

$$C_{gain} = C_{update} + C_{gain_code} + C_{sched} \tag{8.8}$$

However, the `rt_run_optional` and `rt_task_end` calls can actually produce a change in the running task: as a result of the former call, the `rt_schedule` function may detect some available slack and then run Linux, hence producing a context switch. The latter call always produces a context switch, because it delays the execution of the calling task until its next periodic release. As a result, their respective costs can be expressed by the following two expressions, with C_{opt_code} and C_{end_code} being the costs of their respective specific codes:

$$C_{opt} = C_{update} + C_{opt_code} + C_{sched} + C_{switch} \tag{8.9}$$

$$C_{end} = C_{update} + C_{end_code} + C_{sched} + C_{switch} \tag{8.10}$$

The costs of these three system calls are now introduced into the test. Since the calls produce an overhead which is directly related to each application task i, their costs have to be added to the calculation of C_i^{real} in Equation 8.7. The resulting formula is shown in Equation 8.11.

$$C_i^{real} = C_{release} + C_{end} + \sum_{\forall k \in man(i)} (c_{ik1} + C_{gain}) + \sum_{\forall l \notin man(i)} C_{opt} \tag{8.11}$$

8.4.4 Inclusion of Explicit System Calls

The current support provided by FRTL only includes three *explicit* system calls available for the application mandatory components at the real-time level: the `rt_mutex_lock`, `rt_mutex_unlock` and `rt_task_run_dynamic_mandatory` functions. This section calculates their respective worst-case costs.

The costs of the `rt_mutex_lock` and `rt_mutex_unlock` system calls presented here depend on the particular synchronization protocol the system is using;

$$C_{lock} = \begin{cases} C_{update} + C_{lock_code_{CSP}} + C_{sched} & (CSP) \\ C_{update} + C_{lock_code_{PCP}} + C_{sched} + C_{switch} & (PCP) \end{cases}$$

$$C_{unlock} = \begin{cases} C_{update} + C_{unlock_code_{CSP}} + C_{sched} + C_{switch} & (CSP) \\ C_{update} + C_{unlock_code_{PCP}} + C_{sched} + C_{switch} & (PCP) \end{cases} \tag{8.12}$$

However, these two factors do not have to be added to the computation of C_i^{real}, because this feasibility analysis imposes that these factors must be included in the cost of *each* critical section inside any mandatory component. This is also required when computing the B_i factors discussed in Chapter 5.

The cost of requesting the execution of a dynamic mandatory component includes the execution of the corresponding on-line acceptance test, which is executed by the first-level scheduler. The worst-case overhead occurs when the component is accepted, because it may then be immediately selected for execution, producing a context switch to it. Thus, this cost may be calculated by:

$$C_{dynamic} = C_{update} + C_{accept_test} + C_{sched} + C_{switch} \tag{8.13}$$

Again, the cost $C_{dynamic}$ is not explicitly included in the test, since the framework forces this cost to be included in the *wcet* of any mandatory component using the related system call.

8.4.5 Inclusion of Timing Exception Support

This section completes the discussion of the timer interrupt handling performed by the first-level scheduler, which was partially discussed in Section 8.4.2. The case of *wcet* exceptions being produced is presented below, explicitly addressing how to consider them in the feasibility test.

When a *wcet* exception is produced, it is signaled by means of a timer interrupt. The cost of the kernel processing that interrupt may be computed in the following fashion, with the term C_{except_code} being the actual cost of detecting and raising the exception:

$$C_{except} = C_{update} + C_{except_code} + C_{sched} + C_{switch} \tag{8.14}$$

Although the cost C_{except} is normally related to the release of a certain *wcet* handler (in the same sense that the cost $C_{release}$ is related to the release of an application task), it cannot be added to the handler sum factor in Equation 8.4. This is because that sum factor included the execution of mandatory handlers, while the overhead C_{except} is produced any time a task commits a *wcet* exception, in spite of having a user-defined handler attached or not. For this reason, this overhead can actually be added to the task *real wcet* (in the worst case, each task will produce an exception every time it is released, while it is running one of its mandatory components). Thus, the cost C_{except} is included in the calculation of C_i^{real}. However, this case and the non-exception case, in which the task ends normally (by calling the `rt_task_end` function and producing the corresponding C_{end} cost), are exclusive. The formula has to therefore choose the maximum of these two values, in order to reflect the worst-case scenario:

$$C_i^{real} = C_{release} + \max(C_{end}, C_{except}) + \sum_{\forall k \in man(i)} (c_{ik1} + C_{gain}) + \sum_{\forall l \notin man(i)} C_{opt} \tag{8.15}$$

When a handler is executed, it automatically invokes another implicit system call to indicate its end. This system call, which is analogous to the task's `rt_task_end` function, is called `rt_handler_end`. Its cost can be expressed in the following terms:

$$C_{end_hand} = C_{update} + C_{end_hand_code} + C_{sched} + C_{switch} \tag{8.16}$$

The cost of this system call has to be added to the cost of each mandatory handler C_k^h in Equation 8.4. This will be done at the end of the section, where the final equation for calculating R_i is presented. Please note that the cost of this system call does not have to be included when the call is invoked by an optional handler, since in that case the call is assumed to be run in slack time, just as the handler's code.

Finally, it must be noted here that the support for detecting deadline exceptions is not explicitly included in the feasibility test, because it is assumed to be disabled during the regular execution of the system. Nevertheless, if the application programmer wants to enable this feature, the related overhead is due to maintaining the ordered list of active deadlines, as explained in Section 6.5. This is actually performed in two places: when a task is released and when it finishes its execution. As a result, in this case the actual run-time values of costs $C_{release}$ and C_{end} should contain the related overhead.

8.4.6 Final Equation

The assumption that the kernel can be preempted at any time is now removed, since the first-level scheduler actually executes its code with the hardware interrupts disabled. Two traditional approaches have been used to model this non-interruptible behaviour: the first one is to consider that there is an extra task in the system which executes the kernel code at the highest priority; the second approach is to consider the kernel code as being a part of the task source code (as presented so far in this chapter) and then to study the system feasibility with a special technique allowing variations in task priorities [Kle93]. However, a different approach, which is much more straightforward than these two, is presented here.

The new approach considers the kernel source code which deals with an event related to (or caused by) a certain task, to be executed at that task's priority, with a *side effect* of a potential *delay* in the notification of a timer event. In other words, if the timer was programmed to interrupt at a certain instant, and this instant falls within a non-interruptible section of kernel code, then the interrupt will actually be produced right after the section of code is finished, when the interrupts are enabled again. Considering this, the kernel timing analysis developed in the previous sections is valid, as long as the delay in the time events can also be modeled and introduced to the test.

Specifically, the way this delay is accounted for in the analysis depends on the type of time event that is delayed:

a) Task release. The delay in a task release may be modeled in the test as *release jitter* [Aud95], which is normally denoted by J_i (for task i). The release jitter concept

refers to a situation in which the moments at which a periodic task is released are not always strictly periodic, but may suffer (typically small) variations from one release to the following one. This concept is traditionally used, for example, in order to model situations in which task periods are not exact multiples to the clock rate.

In the FRTL analysis, the jitter for a given task corresponds to the maximum delay that the kernel may cause to a task release. The worst-case scenario refers to the following situation: task i has to be released right after the kernel has started the execution of its longest section of non-interruptible code on behalf of another task j. At this moment task j produces a *wcet* exception. In this scenario, the scheduler executes the section, and when it enables the interrupts again, detects and handles the exception. Therefore, in this case, the actual release of task i is delayed for the duration of that longest section plus the exception handling mechanism.

b) End of slack. This event cannot be delayed because it is only programmed to happen when Linux is put into execution (in order to execute the non-real-time level). Thus, the interrupt will always arrive while Linux is being run, with no possible delay.

c) Wcet exception. This particular delay may be caused when the running task exhausts its *wcet* while the kernel is executing a system call (either implicit or explicit) on behalf of this task. Then, the exception notification will be produced right after the system call is finished and the interrupts are enabled again. As a result, the detection of the fault and the subsequent killing of the faulty task may potentially be delayed for, at most, the kernel's longest section of non-interruptible code and this time has to be added to the task *real wcet*. Note that this case changes some of the conditions stated for Equation 8.15, as explained below.

Taking all this discussion into account, the final equations of the feasibility test can be now presented. The first equation computes the longest non-interruptible section of the kernel code, which is later needed in the rest of equations. This cost is denoted by C_{max}.

$$C_{max} = max(C_{release}, C_{gain}, C_{opt}, C_{end}, C_{lock}, C_{unlock}, C_{dynamic}, C_{except}, C_{end_hand}) \tag{8.17}$$

The second equation corresponds to the final expression calculating task i's *real wcet*. This equation is formulated in the following terms:

$$\begin{aligned} C_i^{real} &= C_{release} + C_{except} + C_{max} + \\ &+ \sum_{\forall k \in man(i)} (c_{ik1} + C_{gain}) + \sum_{\forall l \notin man(i)} C_{opt} \end{aligned} \tag{8.18}$$

Compared with the previous Equation 8.15, this new equation has substituted the $\max(C_{end}, C_{except})$ factor by $(C_{max} + C_{except})$. This is now explained. The situation of a task producing a *wcet*

exception has to include not only the cost C_{except} (as assumed in Equation 8.15) but also the potential delay in the timer interrupt notifying the exception. This delay is modeled by adding the cost C_{max} to the cost of handling the exception, as explained above. On the other hand, a task may end in two alternative cases, as discussed above in Section 8.4.5: the task either ends normally, causing a cost of C_{end}, or else produces a *wcet* exception, producing a cost of $(C_{max} + C_{except})$. The worst case to be considered in the computation of C_i^{real} is obviously produced when the task causes a *wcet* exception, since the maximum between $(C_{max} + C_{except})$ and C_{end} is $(C_{max} + C_{except})$ by definition.

Finally, the system complete feasibility analysis can be presented. This final test has been developed by introducing the tasks' real *wcet*s (calculated by Equation 8.18), the jitter accounting for the delay in tasks releases, and the cost C_{end_hand} into the system theoretical test (Equation 8.4). This test is shown in Equation 8.19. Please note that the term $(C_{max}+C_{except})$ is subtracted in the equation; this is because this term has been added to C_i^{real} and, by the time the kernel is executing the code associated to this cost, the task has actually completed its execution.

$$\begin{aligned} R_i^{n+1} \quad = \quad & C_i^{real} - (C_{except} + C_{max}) + B_i + \\ & + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n + C_{max} + C_{except}}{T_j} \right\rceil C_j^{real} + \\ & + \sum_{\forall k \in hand(i)} \left\lceil \frac{R_i^n + C_{max} + C_{except}}{T_k^h} \right\rceil (C_k^h + C_{end_hand}) \end{aligned} \tag{8.19}$$

Once R_i is calculated, the schedulability condition of each task i can be expressed in the following terms:

$$\text{Task } \tau_i \text{ is schedulable if} \quad R_i + C_{max} + C_{except} \leq D_i \tag{8.20}$$

It must again be noted that this test is only sufficient, since the framework's task model allows tasks to have arbitrary initial offsets.

8.5 Overhead Measurements

According to the previous section, the kernel timing behaviour can be considered in the system feasibility analysis by means of introducing certain *kernel constants* in the feasibility test. As a result, if the designer knows the actual cost of these constants for a given computer, she can easily apply the complete test to the application, just by introducing the application-dependent timing information to the test. This information includes the *wcet* of each mandatory component, the worst-case blocking factor of each task, etc.

This section presents actual measurements of such kernel overhead constants, which are summarized in Table 8.1. In particular, the measured values correspond to the execution of

KERNEL CONSTANTS	
Constant	**Description**
$C_{release}$	Cost of releasing one or several tasks
$C_{exception}$	Cost of handling a timing exception
C_{gain}	Cost of calculating the gain time after running a mandatory component
C_{opt}	Cost of activating an optional component
C_{end}	Cosf of ending the current release of the running task
C_{end_hand}	Cost of ending the running handler
C_{lock}	Cost of processing a mutex lock request
C_{unlock}	Cost of processing a mutex unlock request
$C_{dynamic}$	Cost of processing the request releasing a dynamic mandatory component

Table 8.1: Kernel constants introduced in the feasibility test.

some test applications on different hardware configurations, in order to check if the measured values get significantly smaller as the system runs in a faster computer. The measurements have been collected by the FRTL integrated debug and profiling mechanism and then tabulated by the `atrace` off-line tool. The profiling mechanism has been precisely designed for collecting the kernel information corresponding to the overhead constants required by the feasibility test (please compare the constants in Table 8.1 and the kernel events in Table 7.1, page 124).

This section first describes the test applications and the conditions in which the experiments were made. Then, it presents the actual measurements and comments on some significant aspects about them.

8.5.1 Experiment Design

The main purpose of the test applications is to reproduce the system execution in real conditions, in order to get significant values of the kernel constants. In this sense, the test applications have been designed to test every feature of FRTL. All the experiments were made on PC-like machines running RedHat Linux 5.2 and FRTL version 1.0. In particular, five different hardware configurations were tested. The main characteristics of these machines are summarized in Table 8.2. In this table, each hardware configuration is numbered, from 1 to 5, in order to be referenced afterwards. The order of the configurations inside the table denotes increasing computing power.

Each hardware platform was tested by running three test applications containing 4, 8 and 12 tasks, respectively. In order to get compatible results, the same three applications were

HW No.	Processor type	Mhz.	Cache size	RAM size
1	Pentium	200	256 Kb	64 Mb
2	Pentium II	233	512 Kb	64 Mb
3	Pentium II	450	512 Kb	128 Mb
4	Pentium III	500	512 Kb	128 Mb
5	Pentium III	600	512 Kb	128 Mb

Table 8.2: Description of the hardware platforms.

run in every platform. For the sake of achieving incremental results of the overhead on each platform, the task set of each application included the tasks in the previous set(s) as the higher priority tasks. In other words, there were actually 12 different tasks: the 4 higher priority tasks form the 4-task application, the 8 higher priority tasks form the 8-task application and all tasks form the 12-task application. Table 8.3 summarizes the information about these 12 tasks.

Task	Period	Deadline	Wcet	Dynamic	Exception	Mutex	U
τ_1	10000	8000	300	no	no	yes	0.030
τ_2	25000	15000	600	no	no	yes	0.054
τ_3	25000	20000	1100	yes	no	yes	0.098
τ_4	50000	40000	2200	no	yes	yes	0.142
τ_5	50000	50000	1200	yes	no	yes	0.166
τ_6	100000	80000	1800	no	yes	yes	0.184
τ_7	100000	90000	4000	no	no	yes	0.224
τ_8	200000	180000	6600	no	no	yes	0.257
τ_9	200000	190000	7500	no	no	yes	0.295
τ_{10}	500000	450000	16000	no	no	yes	0.326
τ_{11}	1000000	800000	22500	no	no	yes	0.349
τ_{12}	1000000	900000	30000	no	no	yes	0.379

Table 8.3: Attributes of tasks included in test applications.

Several issues about the application tasks can be pointed out from the information stored in Table 8.3:

- The timing attributes of tasks in the table, as well as every timing attribute and measurement hereafter, are expressed in *ticks*, with one tick corresponding to 0.838 microseconds (μs). This is the actual time measurement in RT-Linux (v1).
- The periods of all tasks are harmonic, leading to a *hyperperiod* of 50000 ticks (0.0419 seconds) for the 4-task set, 200000 ticks (0.1676 seconds) for the 8-task set and 1000000 ticks (0.838 seconds) for the 12-task set.
- The application tasks in each set are ordered by following the Deadline Monotonic

order. The last column in the table expresses the accumulated *utilization* (U) of the mandatory load as tasks are being added to the set. Therefore, the 4-task application has a mandatory utilization of 14.2%, while the 8-task and 12-task applications have a respective utilization of 25.7% and 37.9%.

- The *Dynamic* column informs about which tasks invoke the call for releasing dynamic mandatory components. In particular, tasks τ_3 and τ_5 invoke this system call once at each release. Since task τ_3 belongs to each application task set, measures of this cost are available in every experiment.

- The *Exception* column informs about which tasks commit *wcet* exceptions. In this case, tasks τ_4 and τ_6 have been designed to produce a *wcet* exception each time they are released. This feature is also tested in every experiment, since task τ_4 belongs to all task sets. Each of the tasks which produce an exception was designed to have its own mandatory handler, defined at the same priority as the task.

- The *Mutex* column shows that all application tasks use mutexes. Specifically, each mandatory component of each application tasks locks and unlocks the same CSP mutex. In this particular issue, the test applications do not vary the number of mutexes or the synchronization protocol used, since this study has been already presented in Section 5.6.2.

The information about the tasks in the test applications is completed by Tables 8.4 and 8.5. These two tables summarize the internal structure of the 12 tasks, informing about the actual sequence of components forming each task and the attributes of each component.

The structure of tasks in the three task sets was selected in order to meet a main requirement: each task must contain at least one mandatory and one optional component, in order to make its contributions to all the possible implicit system calls. Tasks were also designed in order to have a reasonable structure.

Finally, this section describes how the experiments were made. As described in Section 7.4, the FRTL integrated debug and profiling mechanism collects run-time timing information and stores it in a piece of shared memory. This information is retrieved afterwards, in order to process it appropriately. In such a method of storing the profiling information, the actual size of shared memory available constrains the amount of events to be collected by the first-level scheduler. In particular, all experiments were performed with 2 megabytes of available shared memory, which resulted in nearly 90,000 events. For the specific case of the test applications, this number of events corresponded to approximately 14 seconds worth of timing information for the 4-task set, 11 seconds for the 8-task set and 9 seconds for the 12-task set. As a result, all applications were measured over several hyperperiods.

8.5.2 Measurements of the Kernel Constants

This section presents the actual measurement results obtained after running the three test applications described above on the five hardware platforms summarized in Table 8.2.

Task	**Component**	**Type**	**Slack Fraction**	**WCET**
τ_1	γ_{11}	M	–	100
	γ_{12}	O	1.0	–
	γ_{13}	M	–	200
τ_2	γ_{21}	M	–	450
	γ_{22}	O	1.0	–
	γ_{23}	M	–	150
τ_3	γ_{31}	M	–	350
	γ_{32}	O	0.3	–
	γ_{33}	M	–	350
	γ_{34}	O	0.7	–
	γ_{35}	M	–	400
τ_4	γ_{41}	M	–	600
	γ_{42}	O	0.4	–
	γ_{43}	M	–	1000
	γ_{44}	O	0.6	–
	γ_{45}	M	–	600
τ_5	γ_{51}	M	–	600
	γ_{52}	O	1.0	–
	γ_{53}	M	–	600
τ_6	γ_{61}	M	–	1200
	γ_{62}	O	1.0	–
	γ_{63}	M	–	600
τ_7	γ_{71}	M	–	1000
	γ_{72}	O	0.5	–
	γ_{73}	M	–	1000
	γ_{74}	O	0.5	–
	γ_{75}	M	–	2000
τ_8	γ_{81}	M	–	2500
	γ_{82}	O	0.6	–
	γ_{83}	M	–	1600
	γ_{84}	O	0.4	–
	γ_{85}	M	–	2500

Table 8.4: Internal structure of tasks in the test application (tasks 1 to 8).

Task	Component	Type	Slack Fraction	WCET
τ_9	γ_{91}	M	–	5500
	γ_{92}	O	1.0	–
	γ_{93}	M	–	2000
τ_{10}	$\gamma_{10,1}$	M	–	6000
	$\gamma_{10,2}$	O	0.5	–
	$\gamma_{10,3}$	M	–	5000
	$\gamma_{10,4}$	O	0.5	–
	$\gamma_{10,5}$	M	–	5000
τ_{11}	$\gamma_{11,1}$	M	–	15000
	$\gamma_{11,2}$	O	1.0	–
	$\gamma_{11,3}$	M	–	7500
τ_{12}	$\gamma_{12,1}$	M	–	20000
	$\gamma_{12,2}$	O	1.0	–
	$\gamma_{12,3}$	M	–	10000

Table 8.5: Internal structure of tasks in the test application (tasks 9 to 12).

The experiment results are presented here. The following five tables (Table 8.6 to 8.10, pages 147 to 151) summarize the results of running the three sample applications on each particular hardware configuration. In each of these tables, there are three subtables. Each subtable thus presents the results of one experiment corresponding to running a particular application on a particular computer. These results include the maximum, minimum, average and variance values obtained for each kernel overhead constant corresponding to the experiment.

Several aspects can be pointed out from the results presented. The following are probably the most significant ones:

- The first and main aspect is that the values are small, being less than 40μs in all cases. This overhead is really very small, being almost the same order as the original RT-Linux v1. No implementation of the framework inside the Linux kernel could have offered such low overhead.

- In each experiment, the measured costs could be broadly separated in two groups: a group of less-costly constants (including C_{gain}, C_{end_hand}, C_{lock}, C_{unlock}) and another group of more-costly constants (including $C_{release}$, $C_{exception}$, C_{opt}, C_{end} and $C_{dynamic}$)[2]. Although there were differences between the values in both groups, it is worth noting that the costs of all the kernel constants were normally in the same order. This is because all the system events share the same three-step sequence (update, event processing, rescheduling). A more specific design could have resulted in a more efficient implementation of some system calls. However, this common sequence has permitted the kernel timing characterization to be much more direct.

[2]The costs in the second group are costlier since they include either the processing of a timer interrupt or slack-related calculations.

- Comparing the three experiments in the same machine, the differences of each measured constant in the three applications were very small. Some constants got slightly greater as the number of tasks increased, but the increase was clearly small compared with the absolute value of the constant.
- Comparing the same task set in the five hardware configurations, the measured costs got clearly smaller as the computing power increased. The differences between the two extreme configurations (a 200 Mhz Pentium processor and a 600 Mhz. Pentium III processor) were notable: almost every worst-case cost in the former configuration got half its value in the latter configuration.

It must be noted here that the maximum values shown in the 12-task experiment for a given computer may *not* be exactly the worst-case costs of the related kernel constants in that computer. These worst-case costs are probably very near to the maximum values measured, but the former may be a little greater the latter. This is because it is very difficult to get the very worst case by running only one application, although each constant get thousands of measures in each experiment. Also, the values to be measured are so tiny in terms of the software measuring mechanisms used by the first-level scheduler, that a difference of one or two microseconds may not be detected. For this reason, either several more experiments or a kernel code analysis should be done in order to strictly characterize the worst-case value of each constant.

However, the presented values do show the magnitude of the FRTL kernel overhead, which is typically a few microseconds per system call or interrupt handling.

8.6 Summary and Contributions

This chapter has presented a complete feasibility analysis for the FRTL system. This analysis includes the characteristics of both the framework for FRTS proposed in this thesis and the FRTL kernel. In fact, the feasibility test has been developed by first introducing the characteristics of the task model, that is, the effect of executing mandatory components and handlers. Then, this test is enhanced by introducing the timing behaviour of the FRTL run-time system. This is done in two steps. In the first step, the kernel code is supposed to be perfectly preemptable. This assumption permits the execution of the kernel to be considered as being run at the priority of the task provoking the kernel execution. In the second step, the assumption is removed by considering that the execution of the kernel with the interrupts disabled can be considered not as blocking but as a delay in the notification of a timer interrupt, which can then be specifically modeled depending on the type of time event. For example, the delay of a task release is modeled here as a release jitter.

The first contribution of this chapter is thus the introduction of a complete feasibility test that considers a realistic scenario in which the kernel is analyzed as is, without making simplistic assumptions. Furthermore, the resulting analysis has presented a reduced set of factors to be measured (or analyzed) in the real kernel.

The second contribution is the measurement of these kernel overhead factors. The measurements were collected by running different applications on different hardware platforms. The presented measurements give an idea of how low is the overhead produced by the FRTL kernel and how well the kernel scales as the processor computing power increases. In particular, the measured overhead values were typically in the order of 10 to 20 microseconds in any modern processor tested (Pentium-III or equivalent). Therefore, the experiments show that the FRTL system is not only predictable but also efficient.

4 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	8	40	17.7446	8.291
$C_{exception}$	24	28	25.3452	0.3570
C_{gain}	8	21	10.6328	1.5199
C_{opt}	6	47	23.9352	4.5915
C_{end}	14	35	20.2760	2.3307
C_{end_hand}	8	12	10.8511	0.4480
C_{lock}	7	17	10.1496	0.4716
C_{unlock}	6	17	9.6103	0.5870
$C_{dynamic}$	12	26	17.7425	0.7030

8 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	10	52	19.6818	9.7462
$C_{exception}$	26	34	28.7403	8.2018
C_{gain}	9	20	12.4907	1.7154
C_{opt}	7	46	26.3334	8.3542
C_{end}	15	48	23.56	15.2551
C_{end_hand}	12	16	13.336	1.325
C_{lock}	8	23	11.7148	0.5676
C_{unlock}	8	20	11.2964	0.6438
$C_{dynamic}$	16	31	25.3193	1.5114

12 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	12	37	21.8914	20.87
$C_{exception}$	27	36	30.3558	8.4991
C_{gain}	10	23	14.1576	1.9901
C_{opt}	8	53	28.6935	12.5830
C_{end}	15	47	24.9405	25.4424
C_{end_hand}	12	18	14.4417	0.8846
C_{lock}	10	18	13.2099	0.7040
C_{unlock}	9	17	12.8891	0.7460
$C_{dynamic}$	17	36	29.1408	6.8530

Table 8.6: Experiment 1: kernel overhead measurements for the three test applications running on a 200 Mhz. Pentium processor.

4 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	7	25	12.5753	1.6394
$C_{exception}$	19	23	19.8313	0.3691
C_{gain}	6	14	8.7809	1.3650
C_{opt}	3	22	17.1079	2.4585
C_{end}	11	22	17.5918	2.2024
C_{end_hand}	9	11	9.8433	0.3851
C_{lock}	5	13	8.8399	0.3797
C_{unlock}	5	12	8.7083	0.4162
$C_{dynamic}$	7	16	14.6881	0.5804

8 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	9	25	14.5443	1.4385
$C_{exception}$	20	27	22.6796	3.2662
C_{gain}	7	15	10.3224	1.5988
C_{opt}	4	34	19.3788	4.6748
C_{end}	11	32	19.9740	11.5463
C_{end_hand}	11	13	11.4271	0.2932
C_{lock}	6	14	10.2217	0.5206
C_{unlock}	6	14	10.1894	0.5257
$C_{dynamic}$	11	23	20.2012	1.1934

12 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	10	35	15.7238	1.9528
$C_{exception}$	21	28	23.6809	4.1927
C_{gain}	8	17	11.5241	1.6614
C_{opt}	5	37	20.8114	7.325
C_{end}	12	38	21.5132	16.5578
C_{end_hand}	10	14	12.6871	0.4358
C_{lock}	7	16	11.2323	0.4823
C_{unlock}	7	15	11.3107	0.5946
$C_{dynamic}$	12	27	23.935	3.9130

Table 8.7: Experiment 2: kernel overhead measurements for the three test applications running on a 233 Mhz. Pentium II processor.

4 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	5	18	9.9253	1.563
$C_{exception}$	13	15	13.5238	0.2851
C_{gain}	4	11	6.9844	1.6916
C_{opt}	8	16	11.9316	0.8598
C_{end}	7	17	13.5123	1.5466
C_{end_hand}	7	10	8.1547	0.2974
C_{lock}	4	9	7.4165	0.3707
C_{unlock}	4	9	7.3545	0.3378
$C_{dynamic}$	10	12	10.7767	0.2745

8 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	6	23	11.1228	1.456
$C_{exception}$	14	18	15.4258	1.9669
C_{gain}	5	12	7.9015	1.6901
C_{opt}	9	18	13.1334	2.290
C_{end}	8	22	15.1195	4.1189
C_{end_hand}	5	10	8.8317	0.3611
C_{lock}	5	10	8.2563	0.3883
C_{unlock}	5	12	8.1496	0.3966
$C_{dynamic}$	7	18	13.9808	2.3010

12 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	7	26	11.6371	1.4459
$C_{exception}$	14	21	16.3999	2.3127
C_{gain}	5	13	8.6684	1.6827
C_{opt}	9	20	13.9412	2.5745
C_{end}	8	26	15.9196	6.2467
C_{end_hand}	9	11	9.5636	0.3186
C_{lock}	5	12	8.8596	0.4235
C_{unlock}	5	11	8.8408	0.4232
$C_{dynamic}$	8	19	15.6282	2.4884

Table 8.8: Experiment 3: kernel overhead measurements for the three test applications running on a 450 Mhz. Pentium II processor.

4 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	5	23	7.757	0.5836
$C_{exception}$	11	14	11.2142	0.2636
C_{gain}	3	8	5.4645	0.8180
C_{opt}	1	13	9.5614	0.8199
C_{end}	6	18	10.5087	1.1205
C_{end_hand}	6	8	6.2500	0.2113
C_{lock}	3	7	5.7322	0.2352
C_{unlock}	3	8	5.6718	0.2623
$C_{dynamic}$	3	9	8.5118	0.3977

8 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	5	23	8.1254	0.7381
$C_{exception}$	12	16	12.9523	1.2834
C_{gain}	4	9	6.3142	0.9288
C_{opt}	8	16	10.9529	1.5495
C_{end}	6	18	11.7688	3.3607
C_{end_hand}	6	8	7.476	0.1024
C_{lock}	4	10	6.4919	0.2837
C_{unlock}	4	9	6.4259	0.3099
$C_{dynamic}$	6	13	11.5491	0.5353

12 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	6	26	8.8501	0.7758
$C_{exception}$	12	16	13.5247	1.5761
C_{gain}	4	10	6.9696	0.9556
C_{opt}	8	18	11.7372	1.9460
C_{end}	7	22	12.6772	5.447
C_{end_hand}	7	9	7.8118	0.1923
C_{lock}	4	10	7.659	0.2048
C_{unlock}	4	9	7.185	0.2573
$C_{dynamic}$	6	15	13.397	2.3458

Table 8.9: Experiment 4: kernel overhead measurements for the three test applications running on a 500 Mhz. Pentium III processor.

4 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	3	21	6.2588	0.4938
$C_{exception}$	9	12	9.7831	0.2903
C_{gain}	3	8	4.8906	0.9178
C_{opt}	1	11	8.3714	0.5472
C_{end}	5	16	9.3873	0.7852
C_{end_hand}	3	9	5.5963	0.4093
C_{lock}	3	6	5.1754	0.1975
C_{unlock}	3	6	5.1247	0.1658
$C_{dynamic}$	3	8	7.6386	0.2967

8 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	4	22	7.965	0.5513
$C_{exception}$	10	13	11.825	0.8135
C_{gain}	3	8	5.6153	0.7440
C_{opt}	6	15	9.4847	1.525
C_{end}	5	15	10.4249	3.1972
C_{end_hand}	6	8	6.2621	0.2031
C_{lock}	3	8	5.8097	0.2355
C_{unlock}	3	7	5.7759	0.2398
$C_{dynamic}$	8	12	10.1115	0.1936

12 TASKS				
	Min.	**Max.**	**Avg.**	**Var.**
$C_{release}$	5	22	7.6504	0.7844
$C_{exception}$	10	14	11.5853	0.9012
C_{gain}	4	9	6.1741	0.8936
C_{opt}	2	18	10.833	1.6457
C_{end}	6	19	11.1489	4.2155
C_{end_hand}	5	8	6.8597	0.1449
C_{lock}	4	9	6.2567	0.2653
C_{unlock}	4	8	6.2641	0.2490
$C_{dynamic}$	5	15	11.1740	2.2323

Table 8.10: Experiment 5: kernel overhead measurements for the three test applications running on a 600 Mhz. Pentium III processor.

9

Conclusions and Future Work

9.1 Conclusions and Contributions

In the development of hard real-time systems, the focus has traditionally been on guaranteeing the system timing correctness. In practice, this guarantee is achieved by means of an off-line feasibility analysis, which mathematically proves the ability of the system tasks in meeting their deadlines at run time. Systems are thus carefully designed, implemented and analyzed with the main goal of having a predictable run-time behaviour, in which tasks always meet their deadlines. For the sake of such predictable behaviour, hard real-time systems normally implement simple applications dealing with well-defined, predictable environments. It seems to be generally agreed within the real-time community that the next generation of hard real-time systems should feature a more intelligent and flexible behaviour while still providing such hard guarantees. However, there is no agreement yet on which specific requirements define a *flexible behaviour* in the context of real-time systems or how to make these systems meet such requirements.

This thesis has coined the term Flexible Hard Real-Time Systems (FRTS) to denote complex systems requiring both hard timing guarantees and an adaptive, intelligent and flexible run-time behaviour. The first achievement of this thesis has been to define both the concept and the requirements of FRTS. In general, these requirements are oriented towards two major topics. First, the system should combine tasks with different criticality levels and different cooperative scheduling policies, in order to achieve both strict timing guarantees (by means of scheduling hard tasks with a hard real-time policy) and an intelligent behaviour (by means of

applying a more sophisticated scheduling to non-hard tasks). And second, the system should feature fault tolerance capabilities, in order to recover from run-time errors which may occur when dealing with complex and unpredictable environments.

After defining FRTS and describing their requirements, this thesis has presented a new general framework for building such systems. The framework is defined in three parts: a computational model, a software architecture and a set of services. The framework's computational model is a generalization of both the models for Imprecise Computations and the ARTIS architecture's low-level task model. The proposed model permits FRTS to be defined in terms of a set of tasks, each of which consists of a sequence of mandatory and optional *components*. Mandatory components, which implement time-bounded algorithms, are guaranteed to always be executed. Optional components, which may implement unbounded algorithms, are executed in the processor spare capacity, in such a way that they never jeopardize the guarantee on the mandatory component execution. Each application task can be implemented by any number of components arranged in any order. The computational model also defines the concept of dynamic component. A dynamic component is not included in any task but rather is explicitly released by the application as a result of a particular run-time situation. Dynamic components can be defined as either mandatory or optional. Mandatory dynamic components are accepted via an on-line acceptance test which verifies whether the component can be executed by its deadline while maintaining all the application tasks' deadlines. Overall, by structuring tasks in mandatory and optional components, the designer is assisted in breaking the solution of a problem down into smaller pieces. Furthermore, the designer is able to choose which pieces will always execute to achieve a minimum-quality solution (mandatory components) and which other pieces may possibly be executed to achieve an enhancement of this solution (optional components).

The second part of this framework is a software organization or architecture in which there are two scheduling levels, one for scheduling mandatory components (real-time level) and another for scheduling optional components (non-real-time level). This decoupling permits the system to have two specialized subsystems, each one devoted to achieving a different goal. The real-time level is only concerned with two actions: scheduling task mandatory components in such a way that tasks never miss their deadlines and extracting the maximum amount of slack time available for executing the non-real-time level. This level has been designed to carry out both actions by utilizing a fixed priority preemptive scheduling policy. In turn, the non-real-time level is only concerned with scheduling the task optional components in such a way that the overall system quality is maximized. This can be achieved by means of a sort of utility-based scheduling policy, which is not imposed by the framework. The framework has also specified the interactions between both levels. These interactions have been designed to be as simple and efficient as possible in order to let each level work as independently as possible. This general architecture provides three advantages. First, the real-time level features a simple real-time scheduling policy, which makes this level both efficient and easy to analyze (from the viewpoint of the feasibility analysis). Second, the non-real-time level scheduling decisions are separated from the real-time level; this makes the real-time level

more robust, since these scheduling decisions cannot jeopardize the real-time level scheduling policy. And third, the application designer is assisted in developing custom scheduling policies for the non-real-time level, according to the application requirements.

The third part of the framework definition is a set of services which can be appropriate for supporting FRTS. Two specific services have been defined in the context of this thesis. The first service is a synchronization mechanism that allows mandatory and optional components belonging to any task to safely communicate by sharing memory objects. In this context, the term *safely* refers to maintaining both the consistency of the data being shared and the deadlines of the application tasks. The thesis has presented two alternative synchronization mechanisms based on the use of mutexes: a variation of the Priority Ceiling Protocol (PCP) and a variation of the Ceiling Semaphore Protocol (CSP). Each protocol has been adapted to meet the framework requirements. In particular, the protocol is provided at both application levels (so that mutexes can be directly accessed by mandatory and optional components) and it is made compatible with the scheduling policy at each level. At the real-time level, such compatibility implies adapting the protocol to the particular version of this level's slack stealing algorithm. At the non-real-time level, compatibility is more difficult to achieve, since the particular scheduling policy is not constrained by the framework. Because of this, the protocol has adopted some design decisions which may be considered as simplistic or drastic, but that allow the protocol to work with any scheduling policy. The thesis has also presented an exhaustive theoretical and empirical evaluation of both adapted protocols. The conclusion is that the CSP is clearly a better synchronization algorithm in the context of this framework, because it provides better scheduling conditions to optional components (that is, more available slack and a higher probability of successfully locking the application mutexes).

The second service is an explicit run-time support for timing exceptions. There is a general and simplistic assumption in many real-time systems that hard tasks never overrun. This may be true for systems in which tasks implement simple algorithms and the environment is well-known and predictable. In the application domain of FRTS, however, this is not the case. Nevertheless, since the framework relies on the accurate estimation of task *wcet*s, it is required to incorporate explicit mechanisms which can deal with potential timing exceptions at run time. The thesis has presented a study which discusses which types of exceptions should be detected and handled, depending on the timing behaviour of the run-time system. As a result of this study, the framework proposes an explicit mechanism for detecting and handling task *wcet* exceptions. This mechanism includes the automatic detection of any *wcet* exception and the the possibility of attaching a user-defined handler to each application task. Handlers can be defined as either mandatory or optional components at any priority, in order to allow the designer to exactly define the urgency of executing the handler (in comparison to any task component). The definition of handlers has thus been completely integrated into the framework's task model. The framework also proposes an optional detection and notification mechanism for deadline exceptions, but only to deal with the case in which the system kernel may exceed its expected maximum overhead.

This thesis has also shown how the framework can be implemented, which was one of the

main goals established at the beginning of this work. In particular, the framework has been designed and implemented as real RTOS called Flexible Real-Time Linux (FRTL). This system has been implemented by enhancing the capabilities of RT-Linux v1, which is a minimum hard real-time kernel. FRTL provides the designer with all the features proposed by the framework, as well as some system tools for analyzing the execution of the application. Furthermore, this system has been designed in a way that preserves the predictability and efficiency of the original RT-Linux. Predictability has been demonstrated by means of a timing characterization of the real-time level, which has permitted the development of a complete feasibility analysis, which includes all types of the kernel overhead. Efficiency has been shown by means of empirical results for this overhead, showing that the longest section of kernel code typically takes about 20 to 25μs in the worst case, when the system is running on a regular current computer (with an Intel Pentium III or equivalent processor).

The work presented in this thesis has been developed within a research group called GTI/IA, from the Technical University of Valencia, which is intensively working on other related topics. These topics include a development tool called Inside (which supports the application designer in creating a FRTL application), and the implementation of two example applications for controlling a simulated chemical plant (using a software simulation control environment called Proximax) and an autonomous robot. All this work has been supported by the following grants of the Spanish government: *"Entornos de sistemas basados en el conocimiento de tiempo real"* (TAP94-0511-C02-01), *"ARTIS: herramienta para el desarrollo de sistemas inteligentes en tiempo real aplicados al control de robots móviles"* (TAP97-1164-C03-01), *"ARTIS: herramienta para el desarrollo de sistemas inteligentes en tiempo real estricto"* (TAP98-0333-C03-01) and *"Soporte de ejecución para sistemas empotrados distribuidos"* (TIC99-1043-C03-02). Another related research project, which is funded in this case by the Valencian Government, is called *"Desarrollo de un entorno de ejecución para la realización de sistemas de tiempo real estricto"* (GV98-14-76).

The main contributions of this thesis have been published in the following magazines and conferences: an implementation of an ARTIS prototype was presented in [Gar96b, Gar97a]. This prototype, which was the predecessor of FRTL, was implemented in Solaris by using POSIX facilities. Both papers presented a complete timing characterization of the prototype's kernel. An early implementation of this prototype in RT-Linux was presented in [Ter98]. In [Esp98], the problem of sharing memory predictably in the prototype was addressed. Finally, the actual framework proposed in this thesis and the corresponding FRTL run-time system have recently been published in [Ter99, Terr00a, Terr00b].

9.2 Future Lines of Work

Future lines of work in this still emerging research field are numerous. In the particular context of the proposals of this thesis, three main areas can be considered for further research. The first area is concerned with the scheduling paradigm at the real-time level. This thesis is centered on using the fixed priority preemptive scheduling policy and a slack stealer algorithm in order

to schedule the application mandatory components at the real-time level. Improvements could be obtained in two lines. On the one hand, some restrictions enforced by the task model, such as not allowing arbitrary deadlines or not directly supporting sporadic tasks, could be removed. Any improvement in this direction has to carefully take into account the concept of slack interval, which is a key aspect in the non-real-time level scheduling policy. On the other hand, the limitations of the scheduling techniques at the real-time level could be overcome by completely changing these techniques for new ones. For example, the Earliest Deadline First (EDF) paradigm could be used instead of the fixed priority scheduling, as it is more flexible, produces more available slack and allows for a higher system utilization. Nevertheless, any new scheduling paradigm to be introduced at the real-time level has to preserve the framework architecture of two scheduling levels dynamically sharing the processor, the proposed set of services and, specially, the ability to perform a complete feasibility analysis of the system.

The second area involves the design and implementation of a library of generic non-real-time level scheduling policies. The design and implementation of a variety of scheduling policies is very interesting for the framework. First, because this will help in validating the proposed interface between the first-level and second-level schedulers, which is supposed to be both efficient and as simple as possible. And second, because having a library of standard utility-based policies will allow the designer to choose the one which is best suited for the application or have a model for developing a custom policy.

The third area corresponds to the design and implementation of real applications using the framework/FRTL system. This area is currently under development. In particular, an application for controlling an autonomous mobile robot is being built. This application includes completely controlling the robot sensors and actuators, as well as obstacle detecting, trajectory planning, communication with other robots, learning about the environment status, etc. This kind of problem is very close to the concept of FRTS, since it includes a dynamic, not-completely-specified and sometimes unpredictable environment, hard deadlines for some tasks (such as obstacle detecting) and high reasoning capabilities for other tasks (such as mission or trajectory planning). This application will be a perfect testbed for validating the capabilities of both the framework and the FRTL system proposed in this thesis. Finally, in the development of such real application, exact worst-case values of the kernel overhead will be required in order to study the application schedulability in real conditions. In this sense, kernel code associated with each overhead factor needs to be measured (or analyzed) more accurately.

Bibliography

[Aud90] Audsley, N.C., Burns, A. (1990). "Deadline monotonic scheduling" Technical Report YCS-146. Department of Computer Science. University of York, 1990.

[Aud91] Audsley, N.C., Burns, A. , Richardson, M.F. and Wellings, A. (1991). "Hard Real-Time Scheduling: The Deadline Monotonic Approach". Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta. May 1991.

[Aud93] Audsley, N.C. (1993). "Flexible scheduling of hard real-time systems". Ph. D. Thesis. Department of Computer Science, University of York. 1993.

[Aud95] Audsley, N.C, Burns, A., Davis, R., Tindell, K., and Wellings, A. (1995). "Fixed priority pre-emptive scheduling: an historical perspective". Real-Time Systems vol. 8, pp. 173–198.

[Aud96] Audsley, N.C, Burns, A., Davis, R.I., Scholefield, D.J., and Wellings, A. (1996). "Incorporating optional software components into hard real-time systems". Software Engineering Journal. May, 1996. pp. 133–140.

[Bak91] Baker, T.B. (1991). "Stack-based scheduling of real-time processes". Real-Time Systems, Vol. 3, No. 1, March 1991, pp. 67–99.

[Bod94] Boddy, M. and Dean, T. (1994). "Deliberation scheduling for problem solving in time-constrained environments". Artificial Intelligence, No. 67, pp. 245–285.

[Bol97] Bollella, G. (1997). "Slotted Priorities: supporting real-time computing within general-purpose operating systems". Ph.D. Dissertation, Department of Computer Science, University of North Carolina.

[Bur95] Burns, A., Wellings, A. (1995). "Engineering a hard real-time system: from theory to practice". Software Practice and Experience, vol. 25(7), pp. 705–726.

[Bur96] Burns, A., Wellings, A.J. (1996). “Dual Priority Scheduling in Ada95 and Real-Time Posix”. Proceedings of the 21st IFAC/IFIP Workshop on Real-Time Programming, pp. 45–50, Gramado, Brazil, 1996.

[Car97] Carpenter, B., Roman, M., Vasilatos, N., and Zimmerman, M. (1997). “The RTX real-time subsystem for Windows NT”. The USENIX Windows NT Workshop Proceedings, August 1997, pp. 33–38.

[Car98] Carrascosa, C., García. A., Julián, V., Terrasa, A., Tomás, V., García-Fornés, A., Botti, V. (1995). “OLA: Una herramienta para análisis off-line en sistemas de tiempo real inteligentes”. Actas de la TTIA, November 15–17 1995, pp. 29–41. (in Spanish).

[Dav93a] Davis, R.I., Tindell, K.W., and Burns, A. (1993). “Scheduling Slack Time in Fixed Priority Preemptive Systems”. Proc. Real-Time Systems Symposium, Raleigh-Durham, North Carolina, December 1-3, pp. 222–231, IEEE Computer Society Press.

[Dav93b] Davis. R.I. (1993). “Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems”. Technical report YCS217, Department of Computer Science, University of York, November 1993.

[Dav94] Davis, R.I. (1994). “Guaranteeing X in Y: On-line Acceptance Tests for Hard Aperiodic Tasks Scheduled by the Slack Stealing Algorithm”. Technical report YCS231, Department of Computer Science, University of York, May 1994.

[Dav95] Davis, R.I. Burns, A. (1995). “Optimal Priority Assignment for Aperiodic Tasks with Firm Deadlines in Fixed Priority”. *Information Processing Letters*, Volume 53, No. 5, pp. 249–254, March 1995.

[Dea88] Dean, T., and Boddy, M. (1998). “An analysis of time-dependent planning”. Proceedings of the 7th National Conference on Artificial Intelligence, pp. 49–54, St. Paul, Minnesota, August 1988.

[Den97] Deng, Z, and Liu, J.W.-S. (1997) “Scheduling Real-Time Applications in an Open Environment”. *Proc. of the IEEE Real-Time Systems Symposium*, San Francisco (California), December 1997, pp. 308–319.

[Dij68] Dijkstra, E. (1968). “The structure of the “THE” multiprogramming system”. Communications of the ACM. Volume 11, No. 5. May, 1968, pp. 341–346.

[Esp98] Espinosa, A., Terrasa, A., Garca-Fornes, A. (1998). “A predictable module for sharing memory in RT-Linux”. Proc. of the 23th IFAC/IFIP Workshop on Real-Time Programming (WRTP’98). Shantou, China.

[Fis98] Fisher, P. (1998). “Building Distributed Real-Time Systems with Windows NT and INtime”. Real-Time Computer Show. September 22, 1998.

[Gar96a] García-Fornes, A. (1996). “ARTIS: Una arquitectura para sistemas inteligentes en tiempo real”. Ph. D. dissertation. October 1996. (In Spanish).

[Gar96b] García-Fornes, A., Terrasa, A., Botti, V., Crespo, A. (1996). “Engineering a tool for building hard predictable real-time artificial intelligent systems”. Proc. of the 21st IFAC/IFIP Workshop on Real-Time Programming, Canela, Brazil, November 1996.

[Gar97a] García-Fornes, A., Terrasa, A., Botti, V., Crespo, A. (1997). “Analyzing the schedulability of hard Real-Time Artificial Intelligence systems”. Engineering Applications of Artificial Intelligence, Vol. 10, No. 4, pp. 369–377.

[Gar97b] García-Fornes, A., Terrasa, A., Botti, V. (1997). “Técnicas de planificación de tareas aperiódicas en sistemas de tiempo real estricto”. Novática, Sep–Oct. 1997, No. 129, pp. 22–30. (In Spanish).

[GNU91] GNU General Public License. GNU. Version 2, June 1991. http://www.gnu.org/copyleft/gpl.html

[Gar93] Garvey, A., and Lesser, V. (1993). “Design-to-time real-time scheduling”. IEEE Transaction on Systems, Man and Cybernetics, No. 23(6), pp. 317–347.

[Hay90] Hayes-Roth, B. (1990). “Architectural foundations for real-time performance in intelligent agents”. The Journal of Real-Time Systems, No. 2, pp. 99–125.

[Hay95] Hayes-Roth, B. (1995). “An architecture for adaptive intelligent systems”. Artificial Intelligence, No. 72, pp. 329–365.

[Hor91] Horvitz, E.J., and Rutledge, G. (1991). “Time-dependent utility and action under uncertainty”. Proc. of the 6th Conference on Uncertainty in Artificial Intelligence, Los Angeles, CA, July, 1991.

[Hum99] Humphrey, M., Hilton, E., and Allaire, P. (1999). “Experiences using RT-Linux to implement a controller for a high speed magnetic bearing system”. Proceedings of the Fifth Real-Time Technology and Applications Symposium (RTAS’99), pp. 121–130, Vancouver, Canada, June 1999.

[Hyp] Imagination System Incorporated. Hyperkernel. http://www.imagination.com/hyperker.html.

[POS90] IEEE Std 1003.1 (1990). Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) [C Language].

[POS93] IEEE Std. 1003.1b (1993). Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language] - Amendment 1: Realtime Extensions.

[POS95] IEEE Std. 1003.1c (1995). Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language] - Amendment 2: Threads Extension.

[POS97] IEEE/ANSI Std. 1003.13 (1997). Draft Standard for Information Technology — Standardized Application Environment Profile— POSIX RealTime Application Support (AEP).

[POS96] ISO/IEC 9945-1 (1996). Information technology – Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) [C Language].

[Jos86] Joseph, M., and Pandya, P. (1986). "Finding response times in a real-time system". BCS Computer Journal, 29(5), October 1986, pp. 390–395.

[Jul00] Julián, V., González, M., Rebollo, M., Carrascosa, C., and Botti, V. (2000). "InSiDE: una herramienta para el desarrollo de agentes ARTIS". Proceedings of the SEID'2000. To appear. (In Spanish).

[Kat93] Katcher, D., Arakawa, H. And Strosnider, J. (1993). "Engineering and analysis of fixed priority schedulers". IEEE Transactions on Software Engineering, vol. 19, pp. 920–934.

[Kle93] Klein, M.H., Ralya, T., Pollak, B., Obenza, R., and González-Harbour, M. (1993). "A practitioner's handbook for real-time analysis". Kluwer Academic Publishers. ISBN 0-7923-9361-9.

[Lam97] Lamastra, G., Lipari, G., Butazzo, G. Casile, A., and Conticelli, F. (1997). "HARTIK 3.0: A portable system for developing Real-Time Applications". Proc. of the IEEE Fourth International Workshop on Real-Time Computing Systems and Applications, Taipei (Taiwan), October 1997, pp. 43–50.

[Leh87] Lehoczky, J.P., Sha, L., and Strosnider, J.K. (1987). "Enhanced aperiodic responsiveness in hard real-time environments". Proceedings of the IEEE Real-Time Systems Symposium, San Jose, CA, December 1987, pp. 261–270.

[Leh90] Lehoczky, J.P. (1990). "Fixed priority scheduling of periodic task sets with arbitrary deadlines". Proceedings of the 11th IEEE Real-Time Systems Symposium, Florida (USA), 1990, pp. 201–213.

[Leh92] Lehoczky, J.P., and Ramos-Thuel, S. (1992). "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems". Proc. of the Real-Time Systems Symposium, December 1992, pp. 110–123.

[Les88] Lesser V., Parlin J., Durfee E. (1988). "Approximate processing in real-time problem solving". The AI Magazine, No. 9(1) spring, pp. 44–61.

[Liu73] Liu, C.L., and Layland, J.W. (1973). "Scheduling algorithms for multiprogramming in a hard real-time environment". Journal of the ACM, Vol. 20, No. 1, pp. 44–61. January, 1973.

[Liu91] Liu, J.W.S., Lin, K.J.L., Shih, W.K., Yu, A.C., Chung, J.Y., and Zhao, W. (1991). "Algorithms for Scheduling Imprecise Computations". Computer IEEE May(1991), pp. 58–68.

[Loc86] Locke, C.D. (1986). "Best-Effort Decission Making for Real-Time Scheduling". CMU-CS-86-134 (PhD. Thesis). Computer Science Department, CMU, May 10, 1986.

[Lun99] Lundqvist, T. and Sternström, P. (1999) "Timing anomalies in dynamically scheduled microprocessors". Proc. of the 20th Real-Time Systems Symposium, Dec. 1-3, 1999, Phoenix, pp. 12–21.

[Lyc78] Lycklama, H., and Bayer, D.L. (1978). "The MERT operating system". The Bell System Technical Journal, Vol. 57, No. 6, July-August, 1978, pp. 2049–2086.

[Mou92] Mouabdid, A., Charpillet, F. and Haton, J.P. (1992). "Approximation and progressive reasoning". Proc. AAAI Workshop on Approximation and Abstraction of Computational Theories, San Jose, July 1992.

[Mus93] Musliner, D.J., Durfee, Shin, K.G. (1993). "CIRCA: a cooperative intelligent real-time control architecture". IEEE Transactions on Systems, Man and Cybernetics, No. 23(6), pp. 1561–1574.

[Mus95] Musliner, D.J., Hendler, J.A., Agrawala, A.K., Durfee, E.H., Strosnider, J.K., and Paul, C.J. (1995). "The Challenges of Real-Time AI". Computer, January(1995), pp. 58–66.

[Raj89] Rajkumar, R., Sha, L., Lehoczky, J.P. (1989). "An experimental investigation of synchronization protocols". Proc. of the IEEE Workshop on Real-Time Operating Systems and Software. May, 1989, pp. 11–17.

[Ram94] Ramamritham, K, Stankovic, J.A. (1994). "Scheduling Algorithms and Operating Systems Support for Real-Time Systems". Invited paper, Proceedings of the IEEE, Jan 1994, pp. 55–67.

[Rea00] Real, J. (2000). "Protocolos de Cambio de Modo para Sistemas de Tiempo Real". Ph. D. Dissertation. Universidad Politécnica de Valencia, March, 2000. (In Spanish).

[RTE] The Real-Time Encyclopaedia. `http://www.realtime-info.be/encyc`.

[Sil94] Silberschatz, A., and Galvin P. (1994). "Operating systems concepts". 4th edition. Addison-Wesley, 1994.

[Sha90] Sha, L., Rajkumar, R., and Lehoczky, J.P. (1990). “Priority inheritance protocols: An approach to Real-Time Synchronization”. IEEE Transactions on Computers, vol. 39, n° 9, Sep(1990), pp. 1175–1185.

[Spu90] Sprunt, B. (1990). “Aperiodic Task Scheduling for Real-Time Systems”. Ph. D. Dissertation. Department of Electrical and Computer Engineering. Carnegie Mellon University. August 1990.

[Sta90] Stankovic, J.A., and Ramamritham, K. (1993). “EDITORIAL: What is Predictability for Real-Time Systems?”. Real-Time Systems Journal, Vol. 2, pp. 247–254. December 1990

[Sta93] Stankovic, J. and Ramamritham. K. (1993).“Advances in Hard Real-Time Systems”. IEEE Computer Society Press, September 1993, 777 pages.

[Sta98] Stankovic, J., Spuri, M., Ramamritham, K., and Butazzo, G.C. (1998). “Deadline scheduling for real-time systems”. Kluwer Academic Publishers. ISBN 0-7923-8269-2. 1998.

[Ste92] Steward, D.B., Schmitz, D.E., and Khosla, P.K. (1992). “The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications”. IEEE Transactions on Systems, Man and Cybernetics, v. 22, n. 6, pp 1282–1295. Nov./Dec. 1992

[Ste96] Stewart, D.B. and Khosla, P.K. (1996). “Policy-independent real-time operating system mechanisms for timing error detection, handling and monitoring”. Proc. of the HASE’96, Oct. 1996, Niagara-on-the-Lake, Canada.

[Sto88] Strosnider, J.K. (1988). “Highly responsive real-time token rings”. Ph. D. thesis. Department of Electrical and Computer Engineering. Carnegie Mellon University. August 1988.

[Tan97] Tanenbaum, A.S., Woodhull, A.S. (1997). “Operating systems: design and implementation”. Prentice-Hall, ISBN: 0136386776.

[Ter98] Terrasa, A., Espinosa, A., García-Fornes, A. (1998). “Extending RT-Linux to support flexible hard real-time systems with optional components”. Lecture Notes in Computer Science. Vol. 1474, pp. 41–50.

[Ter99] Terrasa, A., and García-Fornes, A. (1999). “Real-Time synchronization between hard and soft tasks in RT-Linux”. Proc. of the 6th Real-Time Computing Systems and Applications (RTCSA’99), Dec. 1999, Hong-Kong, pp. 434–441.

[Terr00a] Terrasa, A., García-Fornes, A. and Botti, V. (2000). “Flexible Real-Time Linux: A flexible hard real-time environment”. Journal of Real-Time Systems, special issue on Flexible Scheduling of Real-Time Systems. (Accepted).

[Terr00b] Terrasa, A., García-Fornes, A. and Botti, V. (2000). "Including user-defined timing exception handling in FRTL". Proc. of the 7th Real-Time Computing Systems and Applications (RTCSA'00). (Accepted).

[Tin93] Tindell, K. (1993). "Fixed Priority Scheduling of Hard Real-Time Systems". Ph. D. Thesis. Department of Computer Science. University of York. December, 1993.

[Win92] Winston, P.H. (1992). "Artificial Intelligence". 3rd. Edition. Addison-Wesley. ISBN 0201533774. 1992.

[Yod99] Yodaiken, V. (1999). "An RT-Linux Manifesto". Proceedings of the 5th Linux Expo, Raleigh, North Carolina, May 1999.

[Zil95] Zilberstein, S. (1995). "Operational rationality through compilation of anytime algorithms". The AI Magazine, No. 16(2), pp. 79–80.

[Zil96] Zilberstein, S. (1996). "Using anytime algorithms in intelligent systems". The AI Magazine, No. 17(3).