M 2014

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Task management with a real-time operating system for an automated vehicle


**ANA RITA FERNANDES LOBO PEREIRA**
DISSERTAÇÃO DE MESTRADO APRESENTADA
À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
SISTEMAS EMBARCADOS DE TEMPO-REAL

**U.PORTO**

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

**MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES** **2013/2014**

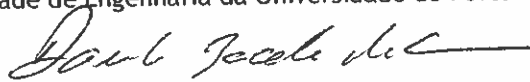
A Dissertação intitulada

"Task Management With a Real-Time Operating System for an Automated Vehicle"

foi aprovada em provas realizadas em 26-09-2014

**o júri**

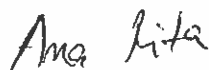
Presidente Professor Doutor Paulo José Lopes Machado Portugal
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto



Professor Doutor Paulo Bacelar Reis Pedreiras
Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Professor Doutor Luis Miguel Pinho de Almeida
Professor Associado do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.



Autor - Ana Rita Fernandes Lobo Pereira

Faculdade de Engenharia da Universidade do Porto

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Task management with a real-time operating system for an automated vehicle

**Ana Rita Pereira**
ee09199@fe.up.pt

Graduation Thesis

Supervisor: Prof. Doutor Luis Almeida

Co-supervisor: Prof. Doutor Glauco Caurin

September 30, 2014

# Abstract

This dissertation was developed in the scope of Real-time Embedded Systems and discusses a limited preemption scheduling technique known as **Fixed Preemption Points**. The purpose of this method is to limit the number of preemptions in the execution of a multitasking application, improving the response time of the longest low priority tasks as well as the overall system schedulability.

This work was developed within an autonomous vehicle project and the Fixed Preemption Points technique is tested in an application running on the VxWorks real-time operating system.

The results show evidence that, despite the benefits demonstrated in theoretical studies, the practical implementations of the referred technique are not always efficient. Instead, real-time operating systems may include, and VxWorks does, optimized implementations of fully-preemptive scheduling that provide the tasks shorter execution times and, for most of them, better response times.

At the end, the dissertation suggests improvements for implementing the Fixed Preemption Points technique.

**Keywords:** Real-time systems, Multitasking, Preemption, Limited-preemptive scheduling, Autonomous vehicle, Response time analysis

# Resumo

Esta dissertação foi desenvolvida no âmbito de Sistemas Embarcados de Tempo-Real e explora uma técnica de escalonamento com preempção limitada conhecida como ***Fixed Preemption Points*** (Pontos de Preempção Fixos). O objetivo deste método é limitar o número de preempções que ocorrem durante a execução de uma aplicação, diminuindo o tempo de resposta das tarefas menos prioritárias e melhorando a escalonabilidade do sistema como um todo.

O trabalho foi enquadrado no projeto de um veículo autónomo e a técnica referida foi testada numa aplicação baseada em VxWorks, um sistema operativo de tempo-real.

Os resultados evidenciam que, apesar dos benefícios demonstrados em estudos teóricos, as implementações da referida técnica poderão não ser muito eficientes. De facto, o escalonamento sem restrições de preempção, para o qual o sistema operativo de tempo-real poderá estar otimizado, como é o caso do VxWorks, fornece às tarefas tempos de execução mais curtos e um melhor tempo de resposta para a maioria delas.

No final, a dissertação inclui algumas sugestões para o melhoramento da implementação da referida técnica.

**Palavras chave:** Sistemas de tempo-real, *Multitasking*, Preempção, Escalonamento baseado em limitação de preempção, Veículo autónomo, Análise do tempo de resposta

# Acknowledgements

Prof. Luis Almeida, who teached me all the project theory, and is constantly guiding me in the work. In addition, his lessons were highly appreciated.

Prof. Glauco Caurin, who helped me breaking through various difficulties, always made useful suggestions and provided an unbeatable optimistic spirit.

My parents, who encourage me and constantly give me the best advice.

My brothers David and Pedro, who, one way or another, always manage to bring cheer to my day.

Rômulo, who cheers and supports me full-time and helps me solve the problems every time anything goes wrong.

Prof. Dr. José Martins Jr., Prof. Dr. Daniel Magalhães, Jorge and Kelen, and everyone else who spent their time helping me with the project.

Everyone in the lab who shared their quotidian with me, as well as all the friends that I met during my stay in São Carlos.

# Contents

**Contents** ix

**List of Figures** xi

**1 Introduction** 1
- 1.1 Motivation . . . 1
- 1.2 Problem definition and related work . . . 2
- 1.3 Objectives . . . 4
- 1.4 Outline of this document . . . 4

**2 Limited preemption scheduling** 5
- 2.1 Overview of limited preemption scheduling techniques . . . 5
- 2.2 Response time analysis . . . 9

**3 Runtime environment** 11
- 3.1 The operating system: VxWorks . . . 11
- 3.2 The software: Wind River Workbench 3.3 IDE . . . 17
- 3.3 The hardware: PowerQuicc II board . . . 20

**4 Implementations** 23
- 4.1 Functions that spawn tasks periodically and sporadically . . . 23
- 4.2 Fully-preemptive, limited-preemptive and non-preemptive tasks implementation . . . 30
- 4.3 Analysis program . . . 41

**5 Practical experiments** 45
- 5.1 The autonomous vehicle . . . 45
- 5.2 Experimental task set . . . 47

**6 Experiments and results** 55
- 6.1 Experiment 1 - Changing priorities . . . 57
- 6.2 Experiment 2 - Disabling the scheduler . . . 66
- 6.3 Experiment 3 - Disabling the scheduler using timers . . . 75
- 6.4 General discussion of the results . . . 83

**7 Conclusion** 85

**Bibliography** **87**

**A. Summary of the response times of all tasks** **88**

# List of Figures

2.1 Example of FPP implementation . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 For different limited preemption scheduling techniques: Ratio of feasible task sets and average number of preemptions . . . . . . . . . . . . . . . . . 9

3.1 Possible VxWorks task states . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Example of task creation and activation . . . . . . . . . . . . . . . . . . . 14
3.3 Example of task delaying . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.4 Example of modifying the priority of a task . . . . . . . . . . . . . . . . . 15

4.1 State diagram of periodic task $\tau$ . . . . . . . . . . . . . . . . . . . . . . . . 26
4.2 Request generation for aperiodic or sporadic tasks . . . . . . . . . . . . . . 29
4.3 Situation where a task is not resumed because it is not in SUSPENDED state 29
4.4 Possible situation where taskSuspend() is not performed immediately after the priority lowering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.5 Undesired preemptions in non- or limited-preemptive tasks . . . . . . . . . 37
4.6 Desynchronization of tasks due to scheduler disabling . . . . . . . . . . . . 37
4.7 Output file with the simplified log . . . . . . . . . . . . . . . . . . . . . . . 43
4.8 Gantt diagram of the application tasks . . . . . . . . . . . . . . . . . . . . 43

5.1 SENA software architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.2 Abrupt and linear gain change . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.3 Example of tChange and tUpdate operations . . . . . . . . . . . . . . . . . 52

6.1 Exp. 1, short tasks - Execution times of the PID tasks . . . . . . . . . . . . 57
6.2 Exp. 1, short tasks - Execution times of tChange and tUpdate . . . . . . . 57
6.3 Exp. 1, short tasks - Execution times of tLocal (total value and longest subjob) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.4 Experience 1, short tasks - Deadline miss rate of the PID tasks . . . . . . . 58
6.5 Exp. 1, short tasks - Response time of the PID tasks . . . . . . . . . . . . . 59
6.6 Exp. 1, short tasks - Response time of tChange and tUpdate . . . . . . . . 59
6.7 Exp. 1, short tasks - Response time and suffered preemptions of tLocal . . 60
6.8 Exp. 1, short tasks - Theoretical response times . . . . . . . . . . . . . . . . 61
6.9 Exp. 1, long tasks - Execution times of the PID tasks . . . . . . . . . . . . 62
6.10 Exp. 1, long tasks - Execution times of tChange and tUpdate . . . . . . . . 62

6.11 Exp. 1, long tasks - Execution times of tLocal (total value and longest subjob) . . . 63
6.12 Exp. 1, long tasks - Deadline miss rate of the PID tasks . . . 63
6.13 Exp. 1, long tasks - Response time of the PID tasks . . . 64
6.14 Exp. 1, long tasks - Response time of tChange and tUpdate . . . 64
6.15 Exp. 1, long tasks - Response time and preemptions of tLocal . . . 65
6.16 Exp. 1, big tasks - Theoretical response times . . . 66
6.17 Exp. 2, short tasks - Execution time of the PID tasks . . . 67
6.18 Exp. 2, short tasks - Execution time of tChange and tUpdate . . . 67
6.19 Exp. 2, short tasks - Execution times of tLocal (total value and longest subjob) . . . 68
6.20 Exp. 2, short tasks - Deadline miss rate of the PID tasks . . . 68
6.21 Exp. 2, short tasks - Response time of the PID tasks . . . 69
6.22 Exp. 2, short tasks - Response time of tChange and tUpdate . . . 69
6.23 Exp. 2, short tasks - Response time and preemptions of tLocal . . . 70
6.24 Exp. 2, short tasks - Theoretical response times . . . 70
6.25 Exp. 2, long tasks - Execution time of the PID tasks . . . 71
6.26 Exp. 2, long tasks - Execution time of tChange and tUpdate . . . 71
6.27 Exp. 2, long tasks - Execution times of tLocal (total value and longest subjob) . . . 72
6.28 Exp. 2, long tasks - Deadline miss rate of the PID tasks . . . 72
6.29 Exp. 2, long tasks - Response time of the PID tasks . . . 73
6.30 Exp. 2, long tasks - Response time of tChange and tUpdate . . . 73
6.31 Exp. 2, long tasks - Response time and preemptions of tLocal . . . 74
6.32 Exp. 2, big tasks - Theoretical response times . . . 74
6.33 Exp. 3, short tasks - Execution time of the PID tasks . . . 76
6.34 Exp. 3, short tasks - Execution time of tChange . . . 76
6.35 Exp. 3, short tasks - Execution times of tLocal (total value and longest subjob) . . . 76
6.36 Exp. 3, short tasks - Deadline miss rate of the PID tasks . . . 77
6.37 Exp. 3, short tasks - Response time of the PID tasks . . . 77
6.38 Exp. 3, short tasks - Response time of tChange . . . 78
6.39 Exp. 3, short tasks - Response time and preemptions of tLocal . . . 78
6.40 Exp. 3, short tasks - Theoretical response times . . . 79
6.41 Exp. 3, long tasks - Execution time of the PID tasks . . . 79
6.42 Exp. 3, long tasks - Execution time of tChange . . . 80
6.43 Exp. 3, long tasks - Execution times of tLocal (total value and longest subjob) . . . 80
6.44 Exp. 3, long tasks - Deadline miss rate of the PID tasks . . . 81
6.45 Exp. 3, long tasks - Response time of the PID tasks . . . 81
6.46 Exp. 3, long tasks - Response time of tChange . . . 82
6.47 Exp. 3, long tasks - Response time and preemptions of tLocal . . . 82
6.48 Exp. 3, long tasks - Theoretical response times . . . 83

# 1. Introduction

This chapter explains the context of the dissertation, which addresses automated vehicles, their potential role in modern societies and the requirements put upon them. From such requirements we show the importance of improving the real-time behaviour of such systems. Then, after discussing some exiting related studies on the subject, we formulate the work to be done in this dissertation as a problem to be solved, its objectives and the outline of this document.

## 1.1 Motivation

In order to make the practice of driving as safe as possible robotics are being seriously considered in the automotive industry. Because of the high speeds automobiles achieve, they require a very responsible usage, and, since the human driver is always liable to fail, there is a great concern with vehicle safety. One possible way of improving safety is having the car alone taking care of some functions that were previously handled by the driver.

An important question is how automated is the vehicle intended to be: Is the car supposed to move completely unmanned? Or should it facilitate the driving practice, overcoming some of the driver's limitations? Ideally the car should be able to detect whether the driver is being able to control the car and adapt the level of autonomy according to the situation. Not only would such a car help the common driver in his/her practice, but it would also make the driving activity possible for people that were not able to do it before, for example, due to partial blindness, deafness or poor reflections, which is the case of many elderly people. Hopefully, such a vehicle would avoid many road accidents caused by lack of attention or poor responsiveness of the driver. [SEN]

Since a defective behaviour in this kind of systems can cause substantial damage to equipment and people, even putting lives at risk, these systems are known as **safety-critical systems**. In their operation, one fundamental aspect is their capacity to properly execute the control functions, with sampling and actuations at the right moments in time. In such systems, missing temporal constraints may have catastrophic consequences and thus they are also classified as **hard real-time systems**.

Hard real-time systems typically involve two fundamental concepts:

**Timeliness:** The vehicle must respond to expected and unexpected situations on time.

This implies that the sampling of operational environment data, e.g., distance to obstacles or speed of wheels, its processing, the response calculation and the actual actuation, e.g. changing direction or reducing speed, have to be done on time to achieve proper vehicle behaviour and ultimately avoid a collision or crash.

**Dependability:** Since human lives depend on the vehicle's correct operation, this machine must provide a trustworthy service. This implies high reliability, i.e., the ability to fulfil the specification for a certain operating time interval, high availability, i.e. expectation of continued system operability, and safety, i.e., capability of avoiding damage to surrounding people or equipment. Other dependability attributes that are also relevant in this context include security, i.e., the preservation of data privacy, authenticity, integrity and correctness against malicious failures, maintainability, i.e., the easiness of system repair in case of defective behaviour, testability, i.e., the capacity to provide access to certain internal parameters, and performance, i.e., bounds to performance degradation in case of faults. [Taisy Silva Weber, 2002]

Each functionality that the vehicle provides is typically composed by a complex set of tasks which are responsible for data collecting, processing, response calculation and actuation. Each task must be executed within its deadline in order to fulfil the timeliness requirements. Situations such as processor overload and the consequent undetermined waiting time of ready tasks, often referred to as *best effort* behaviour, are not tolerated. Thus, to schedule the tasks in such critical systems, a special kind of operating system must be used, namely a **Real-Time Operating System (RTOS)**. These provide adequate task scheduling mechanisms and allow precise measuring of time intervals, such as the execution time of a task. RTOS also support timed actions with bounded jitter, i.e., the fluctuation around the arrival instants of periodic events. Thus, RTOS are very important components in the design of hard real-time systems.

## 1.2 Problem definition and related work

The challenge of this project consists of studying the impact of a specific scheduling technique in a practical situation, concerning meeting timeliness requirements. The application to test is inspired by an autonomous vehicle known as SENA (Embedded System for Autonomous Navigation), developed in the Mechatronics Laboratory of the Mechanical Engineering Department of the Engineering School of São Carlos (EESC), São Carlos, São Paulo, Brazil, under supervision of Profs. Marcelo Becker and Glauco Caurin. At the core of its computing architecture, the SENA project uses the RTOS *VxWorks*, by *WindRiver*.

The scheduling technique we will focus on aims at assuring the timeliness of the application by reducing the response time of the application tasks that have less slack. This technique consists in **limited preemption scheduling** and is considered for the case of fixed priority scheduling. The featured limited preemption scheduling algorithm is known as **Fixed Preemption Points**.

In order to apply it to the vehicle scheduler the RTOS VxWorks shall also be studied, particularly concerning the mechanisms and tools that it offers.

The benefits of limiting preemption have been explored in many scientific works.

The works in [Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013], [Baruah, 2005], [Bertogna and Baruah, 2010] and [Bertogna et al., 2010] explain the drawbacks of both fully- and non-preemptive scheduling policies. The authors agree that, while a non-preemptive scheduling easily spoils the feasibility of a task set, fully-preemptive systems are liable to suffer a significant run time overhead due to the possibly large number of preemptions that occur, turning worst-case execution times (WCET) less predictable, and enlarge the response time of low priority tasks beyond necessary. They also require the usage of complex protocols that deal with the access to shared resources.

In order to avoid a time consuming access to the main memory every time data had to be stored or restored, caches were invented, providing a quicker access to data. However, caches also introduce a delay known as cache interference, which must be cautiously taken into account when dealing with hard real-time systems ([Lee et al., 1999]). The cache must be accessed every time a context switch occurs, as a consequence of a preemption. In order to establish an upper bound to the tasks' WCET in a preemptive system, many studies have focused in analysing the cache interference. In [Ramaprasad and Mueller, 2006] upper bounds are established for the cache-interference in a realistic manner. In [Li et al., 2007], two types of context switch costs are considered, namely direct costs, which include the lag of the operations the processor performs every time a preemption occurs, and indirect costs, which exist when a cache is shared by several processes. In [Chang-Gun Lee, 1998] the cache-related delay is determined by first estimating the preemption cost at each execution point of a task and then, using these results, calculating the worst-case delay. By limiting the number of preemptions held by the tasks, the number of cache-related delays is also limited and more predictable WCETs are obtained. The work in [Lee et al., 1999] proposes a limiting preemption strategy which is an optimal trade-off between preemption cost and blocking delay of higher priority tasks.

When several tasks share a resource in a fully-preemptive system, complex resource access protocols must be followed. They assure that, if a task holding a shared resource is interrupted, the altered or read data is not corrupted in the meantime. By defining the critical zone (execution area where the resource is used) as a non-preemptive region, avoiding preemptions while the resource is held, simpler protocols may be used. The work in [Ramaprasad and Mueller, 2008] establishes upper bounds for the worst-case response time of tasks which are mainly preemptive, but own a non-preemptive region. For this, the WCET of the tasks is estimated, considering cache-related delays. In [Gang Yao, 2009] three methods for calculating the longest non-preemptive region of each task that maintains the system feasible are proposed and exemplified by means of simulations. The work in [Baruah, 2005] proves that, for Earliest Deadline First (EDF), the longest non-preemptive interval of a task depends solely on the WCET, deadline and period of the tasks in the set, and not on the other tasks' non-preemptive sections. EDF is a sheduling technique for dynamic priorities where the highest priority is dynamically assigned to the task with the shortest time to the deadline.

Limited-preemption algorithms are suggested in [Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013] (fixed priority based), [Baruah, 2005] and [Bertogna and Baruah, 2010] (EDF based). The Fixed Preemption Points algorithm is one of the methods suggested

in [Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013] and shall be implemented and experimentally tested in this project. The work in [Yao et al., 2010] provides an efficient and simplified response time analysis when using preemption points in a fixed priority based system. The article also proposes an algorithm for determining each task's longest non-preemptive interval that keeps the task set feasible. In [Bertogna et al., 2010] the authors provide an algorithm that determines an optimal placement of preemption points, considering a static preemption overhead. They prove that, if a task set is not feasible under this method, any other placement scheme will maintain it unfeasible. In [Bertogna et al., 2011] the same authors improve their work to a more complex and realistic modelling by considering that the preemption overhead varies along the task code.

## 1.3 Objectives

This work aims at analysing a specific system with hard real-time requirements, namely an application similar to a particular module of the autonomous vehicle SENA, and verify in practice the potential improvements in the temporal behaviour achieved with the limited preemption scheduling technique.

## 1.4 Outline of this document

The first chapter presented an introduction discussing the motivation of the project, related work and the problem definition. The second chapter approaches several limited preemption scheduling techniques and compares theoretical results. The third chapter provides background information about the practical tools, namely the operating system where the application shall run, the software used to develop the program and the hardware, i.e., the board running the mentioned OS. The implementation details of all developed functions are explained in the fourth chapter. The fifth chapter briefly introduces an overview of the architecture of the autonomous vehicle and describes with detail the task set used for testing. The sixth chapter, after explaining the conditions in which the tests are performed, describes the experiments and discusses the results. A global conclusion of the project is provided in chapter seven.

# 2. Limited preemption scheduling

This chapter explains the scheduling strategy mentioned in the first chapter, namely limited preemption scheduling. This section reveals the advantages of using this method and provides an overview of different existing techniques.

## 2.1 Overview of limited preemption scheduling techniques

There are many ways to assign priority to tasks. A common method is known as *Rate Monotonic* priority assigning. According to this algorithm, the highest priority is assigned to the task with the highest rate (the shortest period), while the lowest priority goes to the task with the longest period. Thus, it is usual that high priority tasks are the ones with shorter execution times and periods, while low priority tasks tend to be longer and slower.

Preemptive real-time systems allow a task to execute immediately after it becomes ready, as long as there is no higher priority task already running. From the point-of-view of the lower priority task that could be running at that moment, this means that it is interrupted (or **preempted**), in order to give in its CPU time to the newly arrived task. Alternatively there are non-preemptive systems, where any newly arriving task has to wait until the currently executing task finishes its execution. Both scheduling techniques have their pros and cons. Preemptive systems are favourable for high priority tasks, but also imply many costs and disadvantages. Every time a task is preempted its state has to be stored in memory so it can resume its execution from the point it has stopped. This implies difficulties in the achievement of mutual exclusion because of possible race conditions among concurrently executing tasks, larger memory requirements and a large overhead related to the process of storing and fetching data. Because of context switch delays, it is more complicated to calculate the WCET of the tasks, thus making the system behaviour harder to predict. Conversely, not only are non-preemptive systems simpler to implement, but they also have a more predictable behaviour. On the other hand, and considering the worst case, if a task is requested right after the longest lower priority task has just begun its execution, in a non-preemptive scenario it must wait for the whole execution of the ongoing task minus a clock tick (because the task arrived after the lower priority task started running). The situation in which a higher priority task cannot execute

because a lower priority one is running is known as **blocking**. Taking into account that low priority tasks may have large execution times, high priority tasks may suffer long periods of blocking, preventing them to finish within their deadlines.

An optimal system should not be non-preemptive because of the long blocking periods of high priority tasks, yet preempt as little as possible in order to make the behaviour more predictable and reduce the overhead. Three approaches for limited preemption scheduling are discussed in [Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013], namely **Preemption Thresholds Scheduling (PTS)**, **Deferred Preemptions Scheduling (DPS)** and **Fixed Preemption Points (FPP)**.

In the formulae indicated in this chapter the following convention is used:

- The index of the parameters indicate the task to which those parameters are concerned.
- The index $i$ corresponds to the task being analysed in the formula, while the index $h$ corresponds to tasks with a higher priority than the analysed task.
- $N$ refers to the maximum number of preemptions that the task indicated by $N$'s index can suffer.
- $P$ refers to the priority of the task.
- $R$ refers to the response time of the task, i.e. the time between a task's request and its finishing instant.
- $T$ refers to the period of the task.
- $C$ refers to the execution time of the task.
- $D$ refers to the relative deadline of the task.
- $\theta$ is a parameter specific to the DPS algorithm and refers to the preemption threshold of the task.
- $Q$ (specific to the PTS algorithm) refers to the maximum non-preemptive interval of the task.
- $q$ (specific to the FPP algorithm) refers to the size of a non-preemptive subjob.
- $N_{\mathrm{pp}}$, specific to the FPP algorithm, refers to the number of preemption points of the task.

Other terms which are specific to particular algorithms are explained in the following corresponding sections.

### 2.1.1 Preemption Thresholds Scheduling (PTS)

According to this method each task keeps a preemption threshold $\theta$, usually larger than its own priority. If task A is running at a given instant and a higher priority task B arrives, B can only preempt A if its priority is higher than A's preemption threshold.

As we can see this method reduces the set of tasks that may preempt a given task. Nevertheless, if B's priority is higher than A's preemption threshold, B can interrupt A for an undetermined number of times.

In order to implement the PTS technique the only necessary action to take is to increase a task's priority to its preemption threshold as soon as it starts executing.

In PTS, it is hard to predict where a preemption will occur, since it depends on the preemption threshold of the running task and on the arrival instants of higher priority tasks.

The largest number of interruptions a task can suffer is the number of times each task with priority greater than the task's threshold can execute during the task's response time: ([Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013])

$$N_i = \sum_{h:P_h>\theta_i} \left\lceil \frac{R_i}{T_h} \right\rceil \tag{2.1}$$

### 2.1.2 Deferred Preemptions Scheduling (DPS)

In this method, each task defines a maximum period $Q$ during which it cannot be preempted. There are two possible implementations for this method: According to the **floating model** the programmer defines a specific code area where preemption is prohibited. The non-preemptive interval may not exceed $Q$ time units and, since the instants of its limits cannot be determined off-line, it is difficult to estimate the effect of this method. In fact, we do not know whether the running time of the non-preemptive region overlaps with the arrival of higher priority tasks or not. In the **activation-triggered model** the non-preemptive region starts as soon as a higher priority task arrives and lingers for a maximum time $Q$. The ready task starts executing after $Q$ time units have passed or earlier, if the running task terminates in the meantime. Once the newly arrived task preempts the running task, it may also prevent preemptions of higher priority tasks.

The implementation of the floating model consists in calling preemption disabling and enabling kernel primitives in the code, in order to establish the boundaries of the non-preemptive regions. Instead, these regions can also be defined by increasing the task's priority to the greatest possible value and decreasing it back to its original value. In the activation-triggered model both techniques can also be used to disable and enable preemption. In this case a timer is set when a higher priority task arrives, keeping count of the maximum non-preemptive interval $Q$.

While in the floating model the non-preemptive regions have unknown positions, in the activation-triggered model we know that, if a higher priority task arrives, an interruption may occur $Q$ time afterwards. As in the case of PTS, the preemption position depends on which task is executing (which is the size of $Q$) and the arrival times of higher priority tasks. Hence both models provide behaviours which are hard to predict.

A pessimistic upper bound of the number of interruptions a task can suffer is the number of times $Q$ fits in the task's execution time. ([Buttazzo, G.C. and Bertogna, M.

and Gang Yao, 2013])

$$N_i = \left\lceil \frac{C_i}{Q_i} \right\rceil - 1 \tag{2.2}$$

### 2.1.3 Fixed Preemption Points (FPP)

This method allows a task to run sections of code without being interrupted. This is achieved by inserting a desired number of preemption points in certain positions of the code. Preemption is disabled between these points. Assuming that a task A is running and a higher priority task B arrives, B must wait until A's code reaches a preemption point. The non-preemptive sections of code are known as *subjobs* and have a maximum execution time of $Q$.

In order to implement this method the programmer needs to insert the preemption points in the desired places. Every time such a point is reached, the scheduler is called and allows higher priority tasks to execute.

Since the allowed preemption situations are defined while the programmer is designing the tasks, it is possible to have a precise notion of the timing overhead caused by the interruptions.

A task can be preempted as many times as the minimum between its number of preemption points and the number of activations of higher priority tasks during the task's response time. ([Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013])

$$N_i = min\left(Npp_i, \sum_{h:P_h>P_i} \left\lceil \frac{R_i}{T_h} \right\rceil \right) \tag{2.3}$$

Consider the example illustrated in Figure 2.1.

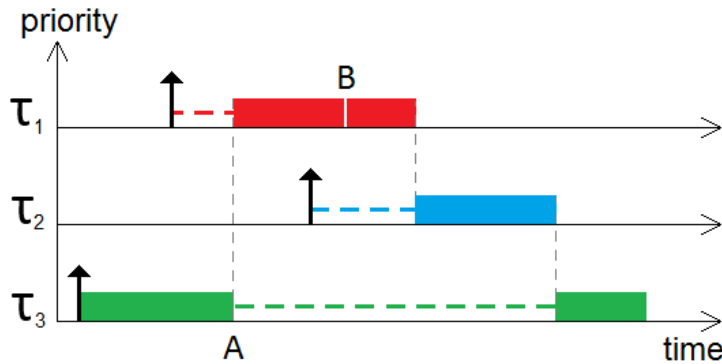



Figure 2.1: Example of FPP implementation

In the example $\tau_1$ is the highest priority task and $\tau_3$ has the lowest priority. Both $\tau_3$ and $\tau_1$ have preemption points, marked by "A" and "B" respectively. In the beginning $\tau_3$ executes without being preempted by $\tau_1$, whose request happens in the meanwhile. Once it encounters a preemption point (situation A), $\tau_1$, which has a higher priority, preempts $\tau_3$. $\tau_2$, which has an intermediate priority, is requested during the first subjob of $\tau_1$. However, once $\tau_1$'s preemption point occurs (situation B), the task is not preempted,

since all waiting tasks ($\tau_2$ and $\tau_3$) have a lower priority. Once $\tau_1$ finishes executing, $\tau_2$ takes the CPU, since it has the highest priority among the waiting tasks. $\tau_3$, having the lowest priority, only resumes execution after $\tau_2$ finishes.

### 2.1.4 Comparison between the three discussed techniques

G. Buttazzo et al demonstrate in [Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013] that, for some task sets, the three limited preemption scheduling techniques satisfy all timeliness requirements, while non-preemptive (NPS) or fully preemptive scheduling (FPS) do not. EDF stands for "Earliest Deadline First".

These results are shown in Figure 2.2.

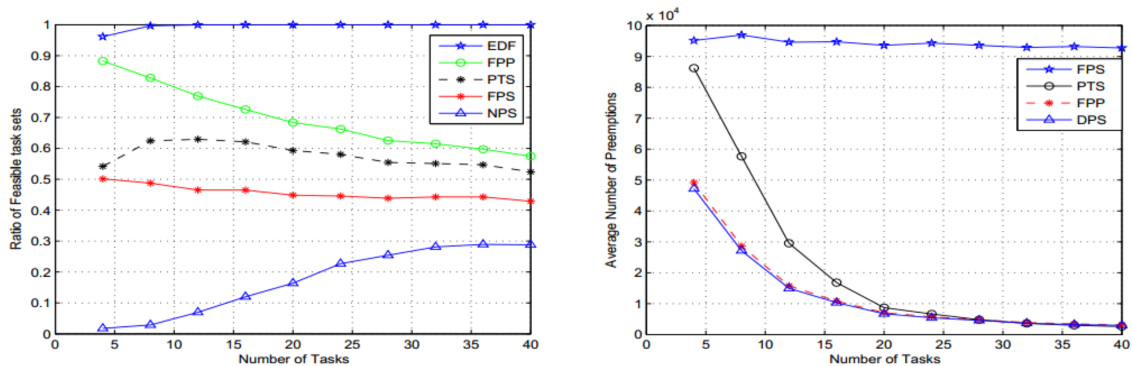



Figure 2.2: For different limited preemption scheduling techniques: Ratio of feasible task sets (left) and average number of preemptions (right). Reproduced from [Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013]

A fixed preemption points scheduling strategy (FPP) leads to a larger ratio of feasible sets, while non-preemptive scheduling has the worst results. Among limited preemption scheduling techniques, the FPP method leads to the largest number of feasible task sets, with the largest advantage over PTS observed for small task sets.

FPP, together with DPS, also brings forward the smallest number of preemptions. This number decreases as the task sets increase.

Since, according to this work, the FPP algorithm provides the best theoretical results, this method was chosen to be implemented in this project.

## 2.2 Response time analysis

This section provides theoretical calculations for the response time of tasks. The equations are used to estimate the behaviour of the tested task set for the non-preemptive, limited- and fully-preemptive situations.

### 2.2.1 Fully-preemptive scheduling

The worst-case response time analysis for a preemptive system is obtained iteratively according to the equations bellow. [Almeida, 2011]

$$\begin{cases} R_i^{(0)} = \sum_{h:P_h \geq P_i, h<>i} C_h + C_i \\ R_i^{(s)} = \sum_{h:P_h \geq P_i, h<>i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h + C_i \end{cases} \tag{2.4}$$

### 2.2.2 Limited-preemptive scheduling with the Fixed Preemption Points algorithm

In [Buttazzo, G.C. and Bertogna, M. and Gang Yao, 2013] the worst-case response time of the tasks with fixed preemption points is deduced to the formulae bellow. If non-preemptive scheduling is classified as a particular case of limited preemption where there are no preemption points, these calculations may as well be used to estimate the worst-case response time of tasks under non-preemptive scheduling.

The longest blocking a task $i$ can suffer ($B_i$) corresponds to the largest non-preemptive section among all lower priority tasks minus one clock tick.

$$B_i = \max_{j:P_j \leq P_i, j<>i} \{q_j^{max} - 1\} \tag{2.5}$$

The starting time of the last subjob of instance $k$ of task $i$ ($s_{i,k}$) is obtained iteratively. In order to obtain the worst-case response time, the starting time of the task's last subjob must be calculated for all instances that occur from the beginning of the application's execution until the end of the longest active period.

$$\begin{cases} s_{i,k}^{(0)} = B_i + C_i - q_i^{last} + \sum_{h:P_h \geq P_i, j<>i} C_h \\ s_{i,k}^{(l)} = B_i + kC_i - q_i^{last} + \sum_{h:P_h \geq P_i, j<>i} \left( \left\lfloor \frac{s_{i,k}^{(l-1)}}{T_h} \right\rfloor + 1 \right) \end{cases} \tag{2.6}$$

where $k$ is the number of instances that run in the longest active interval $L_i$, which is obtained iteratively by

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \sum_{h:P_h \geq P_i, h<>i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{2.7}$$

$$k = \left\lceil \frac{L_i}{T_i} \right\rceil \tag{2.8}$$

Once no task can preempt the last subjob, the finishing time of task $i$, instance $k$, ($f_{i,k}$) is given by

$$f_{i,k} = s_{i,k} + q_i^{last}. \tag{2.9}$$

The response time of task $i$ ($R_i$) is defined according to the following formula:

$$R_i = \max_{k \in [1, K_i]} \{f_{i,k} - (k-1)T_i\} \tag{2.10}$$

and the task set is feasible if, for all tasks, the response time does not exceed its relative deadline.

$$R_i \leq D_i \quad \text{for i = 1, ..., n} \tag{2.11}$$

# 3. Runtime environment

The purpose of this chapter is to introduce the reader into all the relevant tools that were used along this project. They can be listed as the Real-Time Operating System VxWorks, the software Workbench and the hardware, namely the board running VxWorks.

## 3.1 The operating system: VxWorks

This section discusses the main characteristics of the operating system. In order to understand all implementation steps described in chapter 4, this section also provides a summary of some important possible task operations, covering those that are used in this work, too.

### 3.1.1 Overview and global features

Since this project focuses on the implementation of a complex (multitasking) real-time system, it naturally makes use of a special kind of operating system, namely a **real-time operating system (RTOS)**. This type of operating systems allow processing input data with low latency, without buffering delays, and measuring time intervals in a relatively accurate manner.

VxWorks is a popular RTOS optimized for embedded systems. It was developed by **Wind River Systems**, of Alameda, California, and had its first release in 1987.

Before version VxWorks 6.0, the kernel had only one memory space containing both operating system and the user applications. This provided high performance and great flexibility in applications' development, but also great danger of causing interference between applications and kernel features. Since version VxWorks 6.0, the RTOS allows the development of applications in user and kernel mode, without interference between both spaces. Using the Wind River Workbench 3.3, a user mode application is created through a **Real-Time Process** Project, while a kernel mode application is developed by means of a **Downloadable Kernel Module** Project. In the case of the real-time processes, several can be run concurrently. Applications in kernel mode do not assure protection against the programmer's mistakes, but offer a few advantages over user space applications, such as an inferior overall size of the system (which does not contain the features that are provided for processes), a faster execution, features that only the kernel can provide (watchdog timers, interrupt service routines (ISR), etc.) and direct hardware access.

VxWorks' kernel is configurable, meaning that it is possible to add or remove features. Thereby the user may have his/her application run on a kernel that includes just the required features, without wasting any resources on unneeded ones. It is also possible to extend the OS by adding components and application modules to the kernel (file systems, networking protocols or device drivers). System call features may also be available on process applications by adding an API to the interface. VxWorks provides POSIX PSE52 support for user space applications. This can be achieved by configuring the kernel properly. [WindRiver, 2012a]

### 3.1.2 Multitasking

A VxWorks task is the basic schedulable unit in the operating system and executes as a process. Real-time applications may be composed of several independent and concurrent tasks. Since VxWorks supports a multitasking environment, it is possible to develop and run this kind of applications. Though it seems as the tasks are running in parallel, the VxWorks scheduler, which is fully preemptive, is actually calling each task individually, quickly switching from task to task. This means that, if a new task arrives whose priority is higher than the executing task, the newly arrived task preempts the currently running one, so that at each moment the executing task is always the one with the highest priority among the ready tasks in the set.

#### Task states

There are five states that a task can be in. In the **READY** state a task is ready to execute, i.e., if it is not already running, it is waiting for available CPU because a higher priority task is executing. If we command the task to sleep for a certain interval of time, it enters the **DELAYED** state, switching back to the READY state after that time expires. If a task is blocked due to unavailability of a resource other than the CPU, it is **PENDED**. This happens, for example, if the task tries to take a semaphore which is already locked, or is expecting to receive a message from a message queue. Finally, a task can be **SUSPENDED** simply because the user does not want it to execute for an undetermined amount of time, for example, for debugging purposes. There is also a **STOP** state that is only used for debugging or in case of error detection. A task can also accumulate two or more states, for example, if it is PENDED waiting for a semaphore to be unlocked and some other task SUSPENDS it. Figure 3.1 summarizes the main possible task states.

#### Round-robin scheduling

Once a task is in the READY state, it is placed in a ready queue corresponding to its priority. There is a ready queue for each priority level and, as long as there are tasks in a queue, they all execute before any task in the lower priority queues.

In case there are several tasks of equal priority in a ready queue, the user has two options on how they are scheduled: Either they wait for the running task to finish execution, or they adopt a **round-robin scheduling** scheme, i.e., each task executes for

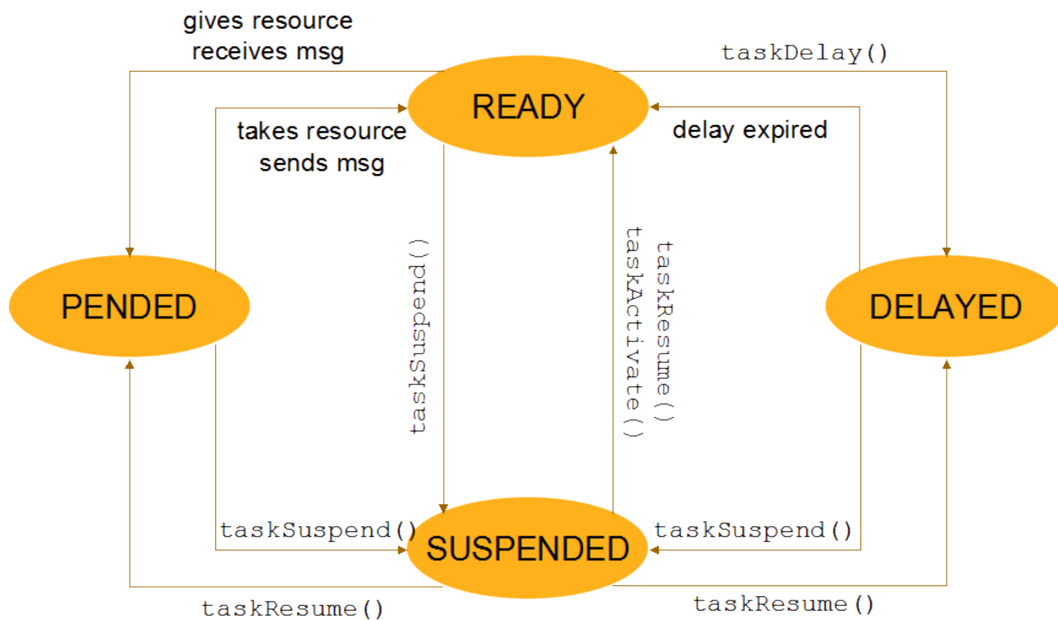



Figure 3.1: Possible VxWorks task states

an equal amount of time, specified by the user. Thereby a bigger fairness among equal priority tasks is achieved. In order to activate the round robin scheduling, the function **kernelTimeSlice()** is called, receiving the maximum number of clock ticks each task shall execute during its turn. The round-robin scheduling may only be performed among tasks of the same priority.

### Task functions

This section provides a global knowledge about the existing functions that allow the user to create, activate, delete, delay, change, and obtain information about tasks.

There are three ways to create a task:

- **taskSpawn()** creates and activates a task, i.e., it immediately enters the READY state. This function allows specifying the task name, priority, options (private or public task, floating point support, stack filling or not, stack protection against underflow or overflow), stack size and up to ten arguments to be passed to the task.
- **taskCreate()** is very similar to taskSpawn(), except that the task is created in the SUSPENDED state. It can become READY by calling **taskActivate()** or **taskResume()**.
- **taskOpen()** offers the most flexibility for task creation. In addition to all the options provided by the other task creating functions, the user can chose whether the task is created in the READY or SUSPENDED state, as well as many stack details, for example, the start address of the execution stack.

All creating functions return the newly created task's ID. Tasks may be created as private or public. When working with real-time processes, if a task is private it is only

visible in its own process, thus it may happen that different processes contain different private tasks with the same name. A public task is visible in the whole system and, in order to avoid misconceptions, its name shall be unique. Figure 3.2 shows an example of task creation in VxWorks.

```
id_tx = taskCreate(name_x, priority_x, opts_x,
                   stack_size_x, func_x, args_x);

taskActivate(id_tx);
```

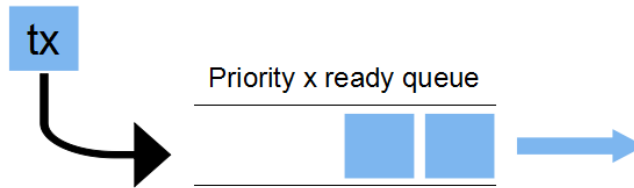



Figure 3.2: Example of task creation and activation

As for delaying a task, there are two possible ways to achieve it:

- **taskDelay()** receives the number of clock ticks the user wants the task to sleep. The function **sysClkRate()** returns the number of clock ticks that occur in a second and is useful if the user wants to specify the delay interval in seconds.
- **nanosleep()** has the same purpose as taskDelay(), except that, instead of receiving the delay in clock ticks (an integer), operates with a structure, timespec, that is formed by two integers: The number of seconds and the number of nanoseconds of the delay.

The delaying process is explained in Figure 3.3.

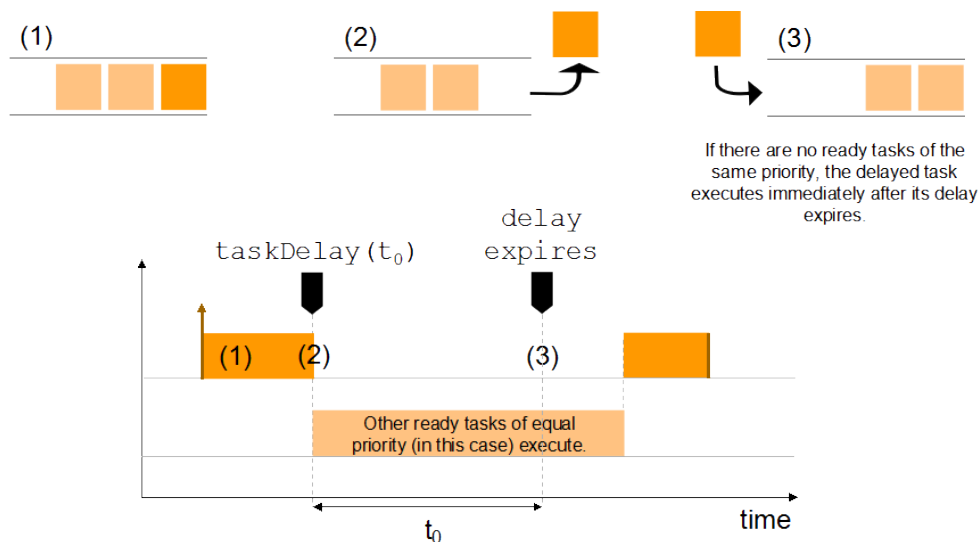



Figure 3.3: Example of task delaying

The calling of any of those functions momentarily removes the task from the ready queue. Once the time is expired, the task is placed at the tail of the ready queue. Supposing that there are other tasks of equal priority in the ready queue, if the user wants to give the non-executing tasks an opportunity to execute, the current task can simply call taskDelay(0), which will remove the task from the ready queue and immediately place it at the tail of the same queue. The ready task which is now at the head of the queue will execute. If there are no tasks of equal priority in the ready queue, the 0 ticks delayed task resumes its execution immediately. The same cannot be done with nanosleep(), since this function returns an error for a null argument.

Within this work is was experimentally observed that the use of taskDelay() is more recommended than nanosleep(), since nanosleep() introduces an undesired extra delay. Besides, taskDelay() is more in line with the speed capacity of the system clock, since it operates directly with the number of clock ticks.

At the moment of creation, the priority of a task must be defined. However, it is possible for any task to change it at any time.

- **taskPriorityGet()** allows finding out the current priority of a specified task.

- **taskPrioritySet()** receives a task ID and the priority to be assigned to that task.

Once the priority of a task is changed, the task is removed from its ready queue and placed at the tail of the ready queue corresponding to the new priority, as shown in Figure 3.4. VxWorks has 256 levels of priority, being 0 the highest level and 255 the lowest. Thus, when comparing priorities, by saying that priority A is superior to priority B, it is implied that the value of A is inferior to the value of B.

```
taskPrioritySet(tx, newP)
```

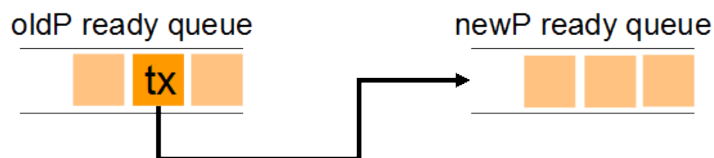



Figure 3.4: Example of modifying the priority of a task

As to task deletion or invalidation, VxWorks offers a few functions:

- Once a task calls **taskSafe()**, it cannot be deleted by other tasks using taskDelete(). This protection may be cancelled by **taskUnsafe()**.

- **taskDelete()** eliminates the specified task, except if it is protected against deletion under the taskSafe() action.

- **taskForceDelete()** deletes the specified task, even if it protected against deletion.

- **taskClose()** closes a task by turning its ID invalid. It does not delete it.

- **exit()** deletes the calling task.

- **taskUnlink()** removes the specified task from its namespace. Thus, it will no more be recognized in searching operations.
- **taskRestart()** terminates and restarts a task, passing to it the same arguments as when it was created.

VxWorks also offers a few functions whose purpose is to provide information about a task:

- **taskName()** returns the name of the task with the specified ID, if that task resides on the same real-time process as the calling task. **taskNameGet()** is similar, but also works for tasks in other processes.
- **taskInfoGet()** fills up a structure with general information about the specified task, such as the ID, the ID of its process, details on the exception and execution stacks, entry point, arguments, options, priority, state, error status, etc. The returned information is related to the instant when taskInfoGet() is called and may not maintain its correctness by the time the user consults the structure.
- **taskOptionsGet()** provides the options chosen for a task.
- **taskNameToId()** does the contrary of taskNameGet(): It receives a name and returns the ID of the task with that name. Notice that, if the task is private, its name might not be unique.

As explained before, VxWorks is constantly rescheduling the task set, making sure that the ready task with the highest priority is the one that takes the CPU. However, any task may disable and reactivate the scheduler using the following routines:

- **taskLock()** disables task rescheduling, causing the calling task to execute without being preempted.
- **taskUnlock()** shall be called in order to reactivate context switching, turning the task vulnerable to preemptions again.

### 3.1.3 Timing

#### Timers and system clock

The system clock frequency determines the step of the delays, i.e., all delays must be multiples of the inverse of the system clock frequency. In this project there are three functions related to the system clock and time that may be used:

- **sysClkRateGet()** returns the current rate of the system clock.
- **sysClkRateSet()** allows changing the system clock rate to a value shorter or equal than 8000Hz in the PowerQuicc II board or 100Hz using the VxWorks simulator provided by Workbench 3.3.
- **clock_gettime()** receives a CLOCK_ID and a pointer to a variable where it saves the current clock value. The time values are timespec structures, where the number of seconds and nanoseconds are saved.

#### POSIX timers

It is possible to configure VxWorks in order to profit from the POSIX timers provided by the RTOS. The following list indicates some of the existing timer functions and their use.

- **timer_create()** creates a timer based on a real-time clock and offers the option to associate the time expiring to a signal event. It returns the ID of the created timer.
- **timer_cancel()** cancels and **timer_delete()** eliminates the specified timer.
- **timer_connect()** associates a timer to a routine which is called every time the specified timer expires. This routine receives as arguments the clock ID and an integer, which can be whatever the programmer wants. Once the routine that calls timer_connect() ends, the timer is automatically deleted. Therefore, if the programmer wants the timer to work for an indefinitely long time, the calling routine usually has an infinite idling loop at the end of the code, which prevents it from ending.
- **timer_settime()** associates a pointer to a value of type *struct itimerspec* to a timer. This structure, which must be defined before, is composed by an offset and a period. Both parameters are structures with two integers, namely the number of seconds and nanoseconds to wait. While the offset determines the time interval between the calling of timer_settime() and the first expiration, the period is the interval between further consecutive time-outs.
- **timer_gettime()** returns, for the specified timer, the remaining time until the next expiration as well as the reload value.
- **timer_getoverrun()** returns the number of expirations of the specified timer.

This section covers all the functions that are used in this project. However, VxWorks offers a large variety of libraries that can be consulted in [WindRiver].

## 3.2 The software: Wind River Workbench 3.3 IDE

Along the whole project Wind River Workbench 3.3 IDE has been used to develop the programs to run on the VxWorks target. This Eclipse based software is optimized to create real-time and embedded applications. It can operate on Windows, Linux and Solaris and allows the user to develop device software to run on VxWorks and Wind River Linux systems. [WindRiver, 2013a]

### 3.2.1 Overview of basic features

This IDE allows easy creation of different types of projects, code analysis, controlling several targets at the same time and contains a debugger that can be used on various processes on one or more targets. Debugging features include breaking points, code stepping and structure analysing.

While developing applications it is possible to download the not fully built project into the target, in order to test it bit by bit. Using the host shell, it is possible to invoke an application or just a task.

Under the Help contents, Workbench also provides several manual and tutorial documents with general and feature-specific information.

By default, when the user launches Workbench, the appearing window is divided in five inner windows: A larger central one, where the project code is developed, the outline of the project, the project explorer (it exposes all projects in the workspace, indicating whether they are closed or opened), a remote systems window and a set of consoles and terminals for building and problem reporting. However, the user may choose various inner windows with different purposes to be displayed in the window.

### 3.2.2 Projects types

In the Project Explorer it is possible to create structures of projects, i.e. projects within projects, in case the user wants to define dependencies among them. There are four types of projects: Linux-specific, VxWorks-specific, user-defined and native application projects. While user-specific projects are totally defined by the user, i.e, the user is responsible for the project organization, how the project is build, the makefile, and so on, the Linux, VxWorks and native host projects are aimed to run on a Linux, VxWorks and the host's OS respectively. In this dissertation only VxWorks projects were used, and, although there are seven types of VxWorks projects, only three of them were explored along this work.

- **VxWorks Image Project (VIP)**
  This type of project allows the user to create and build a kernel image to boot the target with. Here the user may choose the features he/she wants the kernel to support and add them to the image, remove unnecessary existing features and change kernel parameters. It is possible to link various projects to the same image project, so that all of them will run on VxWorks configured with that same image.

- **Downloadable Kernel Module (DKM)**
  DKM projects allow the user to create an application that will run in kernel space. The application where the studied technique is implemented was developed using a DKM.

- **Real-Time Process Project (RTP)**
  RTPs are used to create executables that will run outside the kernel space, i.e. in user mode.

The other VxWorks project Types include **VxWorks ROMFS File System Projects**, whose purpose is to serve as subprojects of other project types that require target-side file system functionality. **VxWorks Source Build Projects (VSB)** offer the possibility to rebuild VxWorks libraries so that they can support products included in the user's platform or exclude unnecessary components and thereby reduce the kernel image's size. The

user shall develop a **VxWorks Boot Loader / BSP Project** if he/she wants to create or adapt a boot loader to be used to boot the kernel into the target. Finally there are **VxWorks Shared Library Projects**, which allow creating libraries that are dynamically linked to RTPs during run-time.

Detailed explanations and tutorials can be found on [WindRiver, 2013a].

### 3.2.3 Tools

Wind River's Workbench 3.3 offers many tools that are useful for code testing and analysis of the results. VxSim and System Viewer are two tools that stood out along this project, since they were strongly used.

#### VxSim

In case the user does not want to test his/her application code on the hardware target until being sure that it works well, or in case a board running VxWorks is not available, the user may perform his/her experiments using the VxWorks Simulator (VxSim) provided by Workbench. This tool consists in a simulated hardware target embedded in the IDE, which runs on the host. This means that the user can test his/her applications in a VxWorks environment, but in his host OS, i.e. without needing the external VxWorks target. Even in distributed applications, using several VxSim sessions, it is possible to set up a simulated network between them or between a VxSim session and the host.

The VxWorks simulator may not support all the features or present such a high performance as an actual hardware target (for example, the system clock frequency's maximum value is much smaller than for the hardware target PowerQuicc II, used along this project). However, VxSim offers a satisfying set of VxWorks standard features and is as capable as a hardware target to interact with external applications. In order to configure a simulated VxWorks kernel, a VxWorks Image Project shall be created and modified according to the application's requirements, just the same way as if it was for the actual VxWorks kernel. In [WindRiver, 2013b] details about VxSim's features and limitations, as well as tutorials for several basic and more specific actions, can be consulted.

#### System Viewer

When developing embedded systems it is important to be sure that all application tasks are running as expected, verifying, for example, if no deadlock or any other type of unexpected blocking is occurring. This tool provided by Workbench offers the user the possibility to visualize what tasks are causing what events and when.

When establishing a connection (it can be any type of connection, like, for example, a VxSim or a hardware target connection), it is possible to launch an associated System Viewer session. When the user runs an application on the target (simulated or not), he/she can command System Viewer to start registering the events happening on the target. The tool logs the events that occur during the execution of the application with the corresponding timestamps, performing tasks, details about the event and total duration of the operation. The user can stop the registering anytime until the storing buffer reaches its

limit. Afterwards a window with a timeline indicating the event density appears, as well as a table containing, for the selected time interval, all timestamped events that occurred during that interval. If the user only wants to see the events related to a particular set of tasks, he/she may filter the table by selecting those tasks.

An event is any action performed by a task or an ISR that may change the state of a real-time system. Examples of events are listed below:

- Task spawns, deletions and other types of state altering, such as priority changing
- Interrupts and exceptions
- System calls
- Signal operations
- Locking and unlocking of semaphores
- Watchdog and non-watchdog timer activity, such as expirations
- Sending and receiving of message queues
- I/O and networking activity
- Memory allocation and freeing

System Viewer allows to choose between three event logging levels, each of them corresponding to a different depth of detailing. The deeper the logging level, the more intrusive this feature is. The **Context Switch level**, as the name suggests, registers task context changes, as well as task priority changing actions. The **Task State Transition level** logs all state changing events (mainly the actions listed above). Finally the **Additional Instrumentation level** is the most intrusive level and is configurable, allowing logging OS-specific event types.

The user may also choose different types of uploading and the size of the storing buffer.

System Viewer is a useful tool for detecting possible CPU issues and problems related to task concurrency, measuring an application's performance (it has a time resolution of tens of nanoseconds) and many other useful affairs the user might be interested in. In this project System Viewer is used to track the application tasks, whose response time shall be measured. [WindRiver, 2012b]

## 3.3 The hardware: PowerQuicc II board

The PowerQuicc II board is the hardware target where the project's applications are set to run. It contains an MPC8260 communications processor, which integrates a RISC microprocessor, a system integration unit and several communications peripheral controllers. Those allow developing several applications that involve communication and networking. Its communications processor module support a variety of protocols, including Fast Ethernet and 155-Mbps ATM. The microprocessor supports frequencies between 100 and 200MHz.

The resolution of the waiting intervals is as high as the maximum rate of the system clock permits. In the case of the PowerQuicc II board, the rate can take any value until 8000Hz, allowing a minimum time step of 125 microseconds.

For more details about the characteristics of board, as well as its features, [Motorola, 1999] can be consulted.

# 4. Implementations

In order to implement the Fixed Preemption Points algorithm in an application, a few functions were developed. Some of them are not related to the algorithm, but are required to satisfy the project's goals. This chapter describes everything that was implemented on top of the existing features provided by VxWorks.

Note: The code presented in this section is merely illustrative. The exposed functions are simplified to their main functionality, ignoring a few necessary details. The purpose is to facilitate their understanding, as well as to maintain the code reading as intuitive, light and simple as possible.

## 4.1 Functions that spawn tasks periodically and sporadically

The application consists mainly of periodic tasks. This section describes the existing VxWorks functions that call routines periodically and explains how they are adapted in order to develop a more suitable function for the purpose.

### 4.1.1 Primitives to set up periodic activities in VxWorks

VxWorks offers a function **period()** that calls a routine periodically, which can be used to create a periodic task. It receives a pointer to the function to be called, the duration of the period in seconds and eight arguments to be passed to the periodic routine. This function spawns another function, **periodRun()**, which contains an infinite loop where the periodic routine is called, followed by a delay function that waits the specified number of seconds. The user cannot specify the priority, options or stack size of the routine to be called periodically. Instead those parameters are assigned with default values.

```
TASK_ID period (int period_s, FUNCPTR func, int arg1, ..., arg8)
{
   TASK_ID tid;
   tid = taskSpawn((char *)NULL, default_priority, default_options,
       default_stack_size, periodRun, period_s, (FINCPTR *)func, arg1, ...,
       arg8);
```

```
   return tid;
}

void periodRun(FUNCPTR func, int period_s, int arg1, ..., arg8)
{
   while(1)
   {
      (* func)(arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8);
      taskDelay(period_s * sysClkRateGet());
   }
}
```

### 4.1.2 Spawning tasks periodically and with an initial offset

It would be, though, desirable for this project to be able to count on a more flexible function to spawn our tasks, so this routine suffered a few adaptations. In addition to the period, one can also define an initial offset (an interval before the calling of the first instance of the task), both characterised in microseconds. Since the taskDelay() function receives the number of clock ticks it is supposed to wait, a function that provides the number of clock ticks corresponding to a given number of microseconds, **usec2ticks()**, was created.

```
unsigned int usec2ticks(unsigned int time){
   unsigned int aux = time, mult, ticks, rate, i=0;
   rate = sysClkRateGet();  // current frequency of the system clock
   while(1){
      if(aux%10 != 0){      // discard least significant zeros
         break;             // to avoid big numbers
      }
      aux = aux / 10;
      i++;                  // ... but keep count of them!
   }
   mult = 1000000 / power(10, i);
   ticks = rate * aux / mult; // (mult * aux) shall be a value in seconds
   return ticks;
}
```

Note that the system clock rate must be defined according to the desired time step for the application:

$$TimeStep = \frac{1}{SystemClockRate} \tag{4.1}$$

Waiting times, such as offsets and periods, are restricted to be integer multiples of the time step. The main advantage of reducing the time units is that it allows the function to be used for a vast range of application rates, from slower to faster ones.

The reason for this choice is that the periods of the tasks featured in the autonomous vehicle are foreseen to lay in the range between 100 microseconds and tens of milliseconds.

According to the adopted strategy to spawn routines periodically, all periodic tasks have a common structure. After declaring the required variables, they contain an infinite loop where, after executing its operations, the task suspends itself. Our spawning function, **offsetPeriodSpawn()**, creates the periodic task without activating it, using taskCreate(). Here the user may define its priority, task options, stack size and until ten arguments. Afterwards it calls **offsetPeriodRun()**, passing as arguments the defined offset, period and ID of the task, obtained through taskCreate(). In turn, offsetPeriodRun() waits the specified offset using taskDelay() before entering an infinite loop. In every iteration of the loop the periodic task is resumed and taskDelay() is called, letting offsetPeriodRun() sleep for an amount of time equivalent to the specified period. This way the periodic task, which suspends itself at the end of execution, is resumed periodically, returning to the beginning of the next iteration of its loop, and starting a new instance. If offset_us is zero, the task is requested straight away and, if period_us is zero, it only runs once. Figure 4.1 illustrates the operation of this process.

```
void periodic_task(void)
{
   // declare variables and initialization code
   while(1)
   {
      // execute task operations
      taskSuspend(0);
   }
}

int offsetPeriodSpawn(char *name, int offset_us, int period_us, int priority,
    int options, int stack_size, FUNCPTR func, int arg1, ..., int arg10)
{
   TASK_ID tid;
   tid = taskCreate(name, priority, options, stack_size, func, arg1, ..., arg10);
   taskSpawn((char *)NULL, default_priority, default_options,
       default_stack_size, offsetPeriodRun, tid, offset_us, period_us, 0, 0, 0,
       0, 0, 0, 0);

   return tid;
}

void offsetPeriodRun(TASK_ID tid, int offset_us, int period_us)
{
   taskDelay(usec2ticks(offset_us));
   while(1)
   {
      taskResume(tid);
```

```
      taskDelay(usec2ticks(period_us));
   }
}

int main(void)
{
   TASK_ID id;
   int offset = 1000000, period = 3000000;
   id = offsetPeriodSpawn("periodicTask", offset, period, 104, 0x100, 2000,
       periodic_task, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
   return OK;
}
```

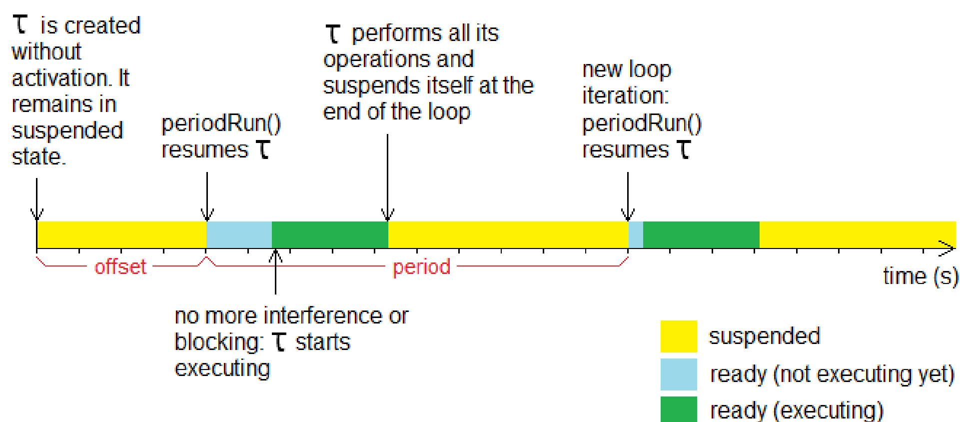



Figure 4.1: State diagram of periodic task $\tau$

### 4.1.3 Spawning tasks periodically to run for a limited number of times

In our application task set there are cases where a task needs to run periodically for a limited number of times and then suspend for a longer period. That number is unknown a priori and only defined by a sporadic task, from time to time. In order to implement that, the function in question, whose structure is identical to the regular periodic task structure, is created (but not activated) in main(). Another task, which is called sporadically, determines how many times the function is to execute and calls **runNTimes()**, which in turn, after a given offset, resumes the function periodically (with a given period) and for the specified number of times.

```
void sporadicTask(void)
{
   // define n_times
   runNTimes(offset_periodicNTimes, period_periodicNTimes, idPeriodNTimes,
       n_times);
}
```

```
void runNTimes(int offset_us, int period_us, TASK_ID tid, int n_times)
{
   int i;
   taskDelay(usec2ticks(offset_us));
   for(i=0; i<n_times; i++)
   {
      taskResume(tid);
      taskDelay(usec2ticks(period_us));
   }
}

int main(void)
{
   TASK_ID idPeriodicNTimes = taskCreate("periodicNTimesTask", priority, 0x100,
       2000, periodic_n_times_task, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
   return OK;
}
```

### 4.1.4 Spawning tasks at random instants

In hard real-time systems the calling of aperiodic or sporadic tasks are consequences of events whose occurring time is unpredictable - for example, the activation of an alarm. The aperiodic task spawning function has only simulation purposes: The idea is to simulate unpredictable events that could happen in the real-life operation of the application, allowing to test the implemented strategies in more realistic (non-deterministic) situations, without actually having them operate in real-life.

Remembering the difference between the concepts of **aperiodic** and **sporadic**, aperiodic tasks can happen at any time, while in the case of sporadic tasks there must be a minimum amount of time between two consecutive requests (**MIT: minimum interarrival time**).

As for the task structure, an aperiodic or sporadic task is framed as a periodic one: In order to allow being called several times, the task consists of an infinite loop that suspends itself at the end and, as for periodic tasks, the reactivation of the task is achieved by resuming it. The difference now consists in determining the waiting interval until the next task request, which for offsetPeriodSpawn() is constant (the specified period), but for **offsetSporadicSpawn()** follows a different logic.

In order to provide as much freedom as possible to the user when he/she creates his/her aperiodic or sporadic tasks, the new function offsetSporadicSpawn() accepts, in addition to all arguments to be sent to taskCreate(), an **offset**, a **lower limit**, an **upper limit** and a **step**, all quantities in microseconds. The offset is the initial interval when no aperiodic/sporadic requests may happen. If it is desired that those requests may happen since the beginning of the execution, its value shall be zero. The lower and upper limits are the minimum and maximum amount of time, respectively, that can pass between two consecutive requests (minimum and maximum interarrival times). In order to implement an aperiodic task, the lower limit shall be a very small number (but not zero). As for the

sporadic task, the lower limit corresponds to the MIT.

The function offsetSporadicSpawn() does the same as offsetPeriodSpawn(), with the slight difference that it calls **offsetSporadicRun()** instead of offsetPeriodRun(), and here is where the main differences reside. First, the function waits offset microseconds to guarantee that there are no requests before that time. Afterwards it enters an infinite loop that waits a certain amount of time and resumes the aperiodic task. That amount of time is a random value between the upper and lower interval bounds. The role of the step parameter is to discretize the resulting waiting interval. As an example, being the lower and upper limits 2 and 15 seconds respectively, and the step 0.1 seconds, the resulting waiting interval can be 2, 2.1, 2.2, ..., 14.8, 14.9 or 15 seconds. If the step is 0.5 seconds, the result can take values of the set 2, 2.5, 3, 3.5, ..., 14, 14.5 and 15 seconds. If complete non-determinism is desired, the step shall be as small as possible, namely the inverse of the system clock frequency. Since stdlib.h's rand() function is being used, the use of offsetSporadicSpawn() requires a time seed to be planted in the main() function, in order to obtain pseudo-random values.

```
int offsetSporadicSpawn(char *name, int offset_us, int low_lim_us, int
    upp_lim_us, int step_us, int priority, int options, FUNCPTR func, int
    stack_size, int arg1, ..., int arg10)
{
   TASK_ID tid;
   tid = taskCreate(name, priority, options, stack_size, func, arg1, ..., arg10);
   taskSpawn((char *)NULL, default_priority, default_options,
      default_stack_size, offsetSporadicRun, tid, offset_us, low_lim_us,
      upp_lim_us, step_us, 0, 0, 0, 0, 0);
   return tid;
}

void offsetSporadicRun(TASK_ID tid, int offset_us, int low_lim_us, int
    upp_lim_us, int step_us)
{
   int n_levels, next_interval_us;
   taskDelay(usec2ticks(offset_us));
   n_levels = 1 + ((upp_lim_us - low_lim_us) / step_us); // # of possible results
   while(1)
   {
      next_interval_us = rand() % n_levels; // random result
      next_interval_us *= step_us;          // convert it to microseconds
      next_interval_us += low_lim_us;       // shall be >= to the lower limit

      taskDelay(usec2ticks(next_interval_us));
      taskResume(tid);
   }
}

int main(void)
```

```
{
  TASK_ID tid;
  int offset = 3000000, low_lim_us = 1000000, upp_lim_us = 9000000, step_us =
      2000000;
  srand(time(NULL));
  tid = offsetSporadicSpawn("aperiodicTask", offset, low_lim_us, upp_lim_us,
      step_us, 104, 0x100, 2000, aperiodic_task, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  return OK;
}
```

In this example, and remembering that all quantities are represented in microseconds, an aperiodic task is created with an offset of 3 seconds. The interval between two consecutive requests lies between 1 and 9 seconds and may take values of 1, 3, 5, 7 and 9 seconds. Figure 5.3 illustrates the possible instants where the request may occur.

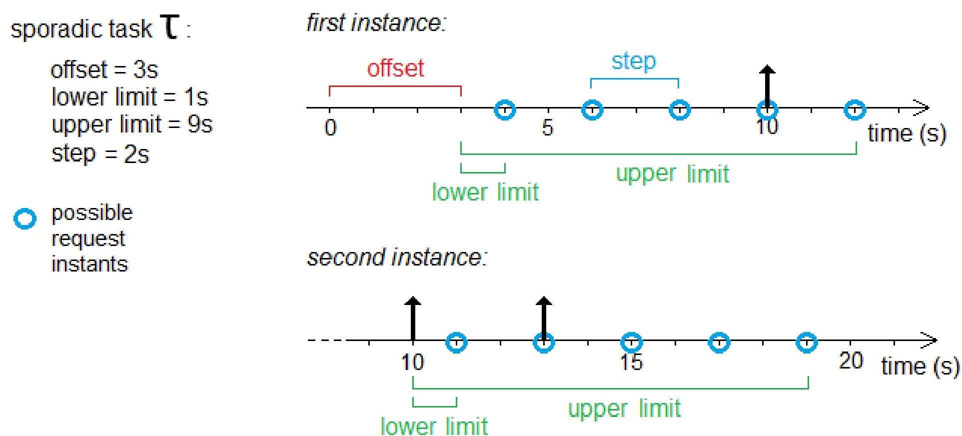



Figure 4.2: Request generation for aperiodic or sporadic tasks

Note 1: We have seen that the periodic task calling consists in periodically resuming a task that repeatedly suspends itself. However, the taskResume() command only actuates if the specified task is in SUSPENDED state. Due to task concurrency, it may happen that the task finds itself in the READY state by the time it is resumed – because it is still running or is waiting to run since the previous taskResume(). Either way this means that the task has missed its deadline, since the latter is smaller or equal to the period. In these cases taskResume() will have no effect and there will be no new request. This situation corresponds to skipping one or more activations. A possible example of such a situation in shown in Figure 4.3.

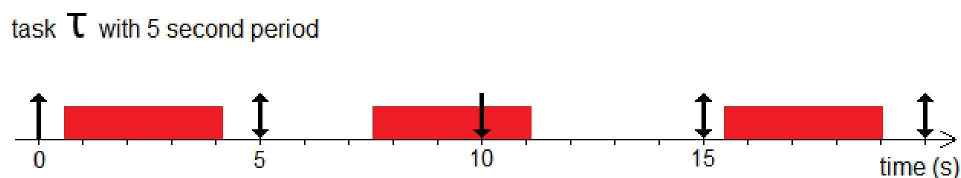



Figure 4.3: At instant 10 there is no new request since the task is not in suspended state

Note 2: It shall be kept in mind that the periodic task spawning routine does not assure interarrival times between consecutive task requests rigorously equal to the task's period. In fact it waits a minimum interarrival time equal to the desired period, and, afterwards, requests the task as soon as it can execute. Thus, the interval between two consecutive task requests may be slightly longer than the desired period. Despite this uncertainty, the concerned application tasks are considered periodic along this document.

## 4.2 Fully-preemptive, limited-preemptive and non-preemptive tasks implementation

The goal of this project is to compare fully-preemptive, non-preemptive and limited preemptive systems for a given set of tasks. In order to force the respective expected behaviours, the tasks are structured differently for each type of scheduling.

### 4.2.1 Fully-preemptive tasks

Since the traditional VxWorks scheduler is fully preemptive itself, the task structure does not require any modifications:

```
void periodic_task(void)
{
   // declare variables and initialization code
   while(1)
   {
      // execute task operations
      taskSuspend(0);
   }
}
```

### 4.2.2 The Fixed Preemption Point (FPP) algorithm implementation

The FPP algorithm, as explained before, consists in having the tasks to run non-preemptively, except for some points of the code, the so called **preemption points**.

Two implementations are explored in this work, namely one based in **controlling priorities** of the application tasks and another one based on **disabling and reactivating the scheduler**.

#### Changing priorities

Since VxWorks supports a fully-preemptive scheduling and offers the possibility of changing any task's priority at any time, both non-preemption and preemption points can be implemented by increasing and decreasing the task's priority respectively, using taskPrioritySet().

In order to turn a task non-preemptable, its priority is increased to a value which is higher than the highest priority among the task set. This value is globally known in the system and shall be referred to as **PRIOR_NOPREEMP** in this document. In order to implement a preemption point, the original priority **PRIOR_ORIG** shall be momentarily assigned to the task. Just before the task ends, or at least before the remaining tasks have the opportunity to start, the task's priority must return to its original value.

Here is the structure of a periodic task with preemption points:

```
void periodic_task(void){
   // declare variables and initialization code
   while(1){
      taskPrioritySet(my_tid, PRIOR_NOPREEMP); // 1st subjob
      // execute 1st subjob's operations
      taskPrioritySet(my_tid, PRIOR_ORIG);  // 1st preemption point

      taskPrioritySet(my_tid, PRIOR_NOPREEMP); // 2nd subjob
      // execute 2nd subjob's operations
      taskPrioritySet(my_tid, PRIOR_ORIG);  // 2nd preemption point

      ...
      taskPrioritySet(my_tid, PRIOR_NOPREEMP); // nth subjob
      // execute nth subjob's operations

      taskPrioritySet(my_tid, PRIOR_ORIG);  // return to original priority value
      taskSuspend(0);          // end of task instance
   }
}
```

### Disabling the scheduler

It is also possible to disable the scheduler in order to start a non-preemptive section. This way, even if there are other ready tasks, and regardless of their priorities, they will not be called and set to run. When the non-preemptive section ends, the scheduler is reactivated, and gives the CPU to the ready task with the highest priority.

```
void periodic_task(void){
   // declare variables and initialization code
   while(1){
      taskLock();                      // 1st subjob
      // execute 1st subjob's operations
      taskUnlock();                    // 1st preemption point

      taskLock();                      // 2nd subjob
      // execute 2nd subjob's operations
      taskUnlock();                    // 2nd preemption point
```

```
    ...
    taskLock();                    // nth subjob
    // execute nth subjob's operations

    taskUnlock();                  // reactivate the scheduler
    taskSuspend(0);          // end of task instance
  }
}
```

### Ending the task

Although both ideas are simple, they do not work for a particular situation: Consider that task A is executing and, during A's last subjob, task B (with higher priority than A) is requested. As soon as A's last taskPrioritySet() or taskUnlock() command finishes, task B starts executing, not giving task A the chance to perform its own suspension. Since a task can only be resumed if it is in SUSPENDED state, it might happen that, although task A finished all its operations, it will not be resumed again and considered to have missed its deadline. This happens if, by the time a new request of A occurs, task A has not retaken execution to suspend itself yet, as exemplified in Figure 4.4.

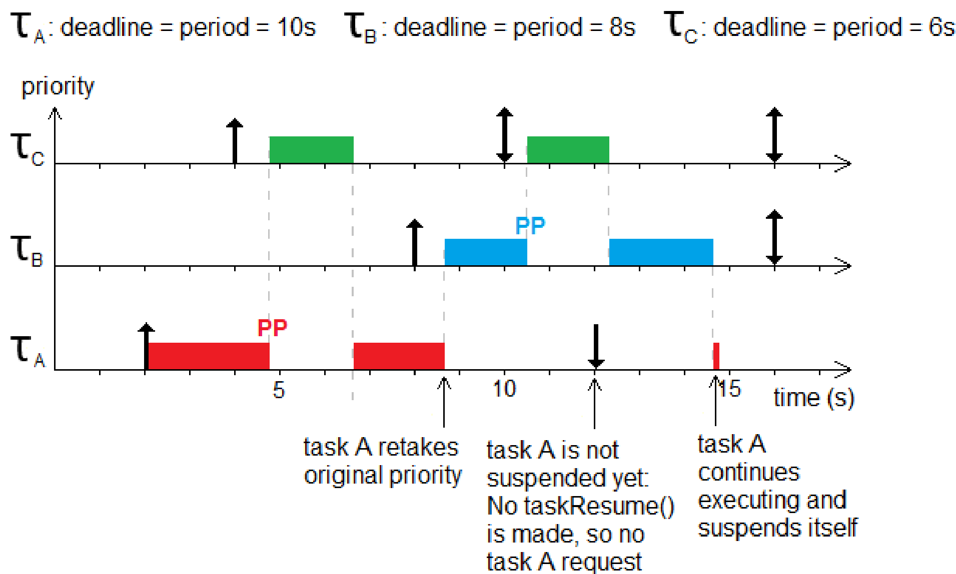



Figure 4.4: Possible situation where taskSuspend() is not performed immediately after the priority lowering

To avoid such situations, the priority lowering / scheduler re-enabling and the task suspension must be performed atomically, i.e., without letting any application task run in between. The idea of locking a semaphore before and releasing it after those operations quickly comes to mind. However, since the last operation happens to be a self-suspension, the task would not be able to unlock the semaphore. A possible solution is to call upon a higher priority task for the purpose. Thus, a special task was implemented, **taskEnd()**,

whose only intention is to end application tasks properly.

Depending on whether we are using priority manipulation or scheduler disabling, taskEnd() has a different operating principle.

If using priority manipulation, every application task, upon starting a non-preemptive section, must store its ID and original priority in global variables (**global_id** and **global_prior**). At the end of the execution, instead of suspending itself, it resumes taskEnd(), whose priority is higher than PRIOR_NOPREEMP. The ending routine consists of an infinite loop where it lowers the priority of the task, suspends it and finally suspends itself. In order to change the priority of the right task and suspend it, it uses the values stored in global_id and global_prior. Since taskEnd()'s priority is higher than the priority of any task in the set, no ready task can preempt it, and it executes immediately after it is resumed.

```
TASK_ID global_id;
int global_prior;

void periodic_task(void)
{
   // ...
   while(1){
      taskPrioritySet(my_id, PRIOR_NOPREEMP); // start non-preemptive subjob
      global_id = my_id;                      //
      global_prior = PRIOR_ORIG;              //
      // ...
      taskResume(id_end);               // finish task instance
   }
}

void task_end(void){
   while(1){
      taskPrioritySet(global_id, global_pr);
      taskSuspend(global_id);
      taskSuspend(0);
   }
}
```

As for the case where the scheduler is turned off and on, taskEnd() must be modified since we cannot simply replace the taskPrioritySet() for a taskUnlock() command. Once a task disables the scheduler using taskLock() it must be the same task that unlocks it, or the system will block. The adopted solution consists of having the periodic task to resume taskEnd() and than unlocking the scheduler. Once the scheduler is activated, taskEnd() executes and suspends the periodic task, before suspending itself. In this case, since we are not dealing with priorities, it is only necessary to store the task's ID in global_id.

```
TASK_ID global_id;
```

```
void periodic_task(void)
{
   // ...
   while(1){
      taskLock();                // start non-preemptive subjob
      global_id = my_id;         //
      // ...
      taskResume(id_end);         // finish task instance
      taskUnlock();               //
   }
}

void task_end(void){
   while(1){
      taskSuspend(global_id);
      taskSuspend(0);
   }
}
```

#### Simplifying the code

In order to keep the code as intuitive as possible, all sequence of operations that are performed at the beginning of a subjob, at the end of the task instance and in a preemption point are encapsulated in functions.

Implementation by manipulating priorities:

```
TASK_ID id_end;

// Start non-preemptive subjob
void startSubjob(TASK_ID tid, int prTask){
   taskPrioritySet(tid, PRIOR_NOPREEMP);
   global_id = tid;
   global_pr = prTask;
}

// Preemption point
void preempPoint(TASK_ID tid, int prTask){
   taskPrioritySet(tid, prTask);
}

// End instance
void endJob(){
   taskResume(id_end);
}

// Basic structure of a periodic/aperiodic task using those functions
```

```
void periodic_task(void){
   // declare variables and initialization code
   while(1){
      startSubjob(my_id, PRIOR_ORIG, PRIOR_NOPREEMP);
      // execute 1st subjob's operations
      preemptionPoint(my_id, PRIOR_ORIG);

      startSubjob(my_id, PRIOR_ORIG, PRIOR_NOPREEMP);
      // execute 2nd subjob's operations
      preemptionPoint(my_id, PRIOR_ORIG);

      ...
      startSubjob(my_id, PRIOR_ORIG, PRIOR_NOPREEMP);
      // execute nth subjob's operations

      endJob();
   }
}
```

Implementation by disabling the scheduler:

```
TASK_ID id_end;

// Start non-preemptive subjob
void startSubjob(TASK_ID tid){
   taskLock();
   idGlobal = tid;
}

// Preemption point
void preempPoint(void){
   taskUnlock();
}

// End instance
void endJob(void){
   taskResume(idEnd);
   taskUnlock();
}

// Basic structure of a periodic/aperiodic task using those functions
void periodic_task(void){
   // declare variables and initialization code
   while(1){
      startSubjob(my_id);
      // execute 1st subjob's operations
      preemptionPoint();
```

```
        startSubjob(my_id);
        // execute 2nd subjob's operations
        preemptionPoint();

        ...
        startSubjob(my_id);
        // execute nth subjob's operations

        endJob();
    }
}
```

Finally, in order to simplify the main() function, all operations that are required at the beginning of the execution are also encapsulated in an **init()** function. This function is the same for both FPP implementations. It receives the desired system clock rate and does the following:

- Changing the main() function's priority to the maximum possible value (0 in VxWorks), so that the application tasks may have very high priorities without ever exceeding main()'s.
- Defining the seed to be used in the rand() function.
- Changing the system clock rate to the specified rate.
- Creating the function taskEnd() using taskCreate(), i.e., without activating it, and storing its ID in the global variable id_end.

### 4.2.3 Non-preemptive tasks

Non-preemptive tasks have an identical structure as limited-preemptive tasks, except that they contain no preemption points.

```
void periodic_task(void){
    // declare variables and initialization code
    while(1){
        startSubjob(...);
        // execute task's operations
        endJob();
    }
}
```

Note: Although it is intended to avoid preemption during the whole execution of the tasks, this is not possible when applying any of the presented methods. Figure 4.5 explains an undesirable, but possible situation that may happen.

The phases of the execution (A, B, C and D) are represented in the graphic. Once preemption is forbidden by increasing the priority or disabling the scheduler (A) until

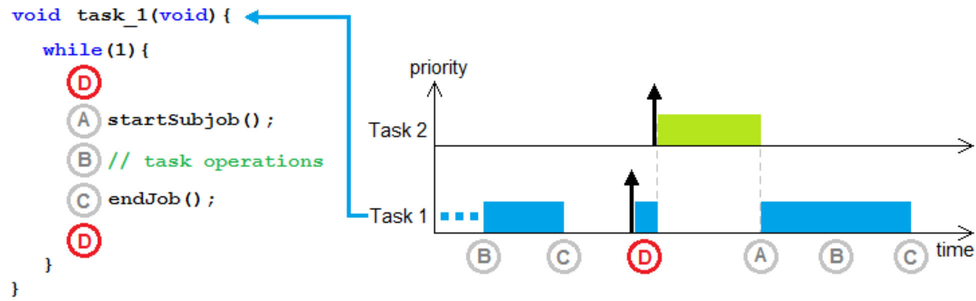



Figure 4.5: Undesired preemptions in non- or limited-preemptive tasks

this action is cancelled and the task is suspended (C) there is no risk of preemption by any other application task. But, just after the task is resumed to its next instance until the startSubjob() in the next iteration, there are a few moments where the task can be preempted (D). If a higher priority routine is requested during that time, it will preempt the task, even if its instance has already started.

### 4.2.4 Drawbacks when disabling the scheduler

When creating a set of periodic or sporadic tasks it is implied that, for each application task, another task, which is responsible for waking up the application task at the right moment, is also created. If the task is periodic, the wake-up task is offsetPeriodRun() and, if sporadic, it is awaken by offsetSporadicRun(). As shown in the respective sections, those tasks are basically composed by an infinite loop that resumes the periodic / sporadic task and sleeps for the proper amount of time. If the programmer disables the scheduler in order to avoid preemption, the wake-up tasks will be unable to execute, since they are regular tasks. This behaviour is not desired, since it causes the application tasks to miss activations. Figure 4.6 illustrates a possible example. Suppose that all tasks 1, 2 and 3 are requested at instant 0. Task 1 and Task 2 have periods of 3 and 5 units of time respectively. Each wake-up routine, whose priority is higher than all application tasks, are marked with a lighter tone of the colour of the corresponding application task.

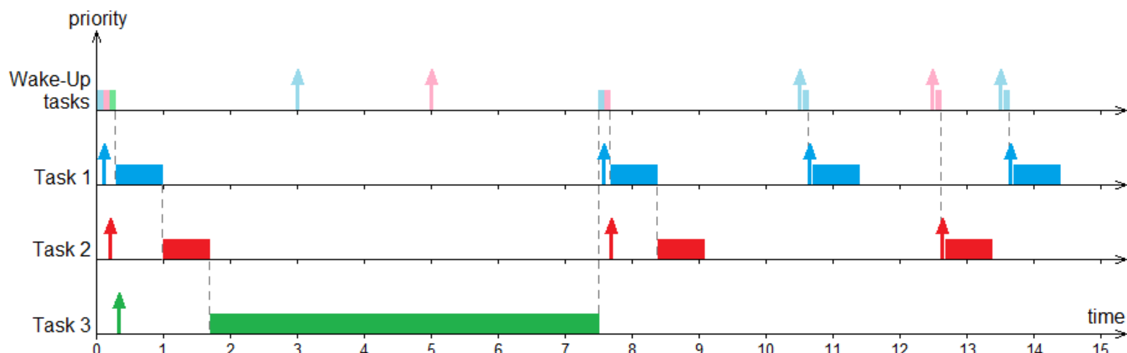



Figure 4.6: Desynchronization of tasks due to scheduler disabling

When Task 3 executes, it disables the scheduler, preventing any task to run. At moments 3 and 5 respectively, the wake-up routines of Task 1 and 2 become ready, but

they only execute after Task 3 finishes, reactivating the scheduler, at moment 7.5. They resume Tasks 1 and 2 and delay themselves for the tasks' periods. This way, Task 1 will be resumed at $7.5 + 3 = 10.5$ and Task 2 at $7.5 + 5 = 12.5$. Ideally, the activation instants of Task 1 would be 0, 3, 6, 9, 12, etc. and of Task 2 would be 0, 5, 10, 15, etc. This does not happen as, every time another task activates the scheduler, all blocked wake-up routines run immediately afterwards and reset the phases of the application tasks.

One possible solution is the use of interrupts, since the operations taskLock() and taskUnlock() only affect tasks and not ISRs. However, it is not recommended to use ISRs that may cause blocking to the system, such as occupying semaphores. In our case the ISRs would resume tasks that lock the scheduler, which is considered a blocking operation.

Another possible solution is the use of regular tasks, but, instead of having a wake-up task that delays itself, POSIX timers take care of the delays. Timer expiration and reloading are performed independently of whether the scheduler is on or off, so they prevent the application tasks from desynchronizing. Thus, different versions of the periodic and sporadic task spawning routines were made and are detailed in the following sections.

### Spawning tasks periodically with an initial offset using a POSIX timer

As explained in chapter 3, VxWorks provides POSIX timers, which allow the programmer to define an initial offset until the first time-out, and a period, which is the waiting time between consecutive time-outs afterwards. This feature is appropriate for the periodic task spawning function, since there are similarities between the operation of the timer and the desired behaviour of the routine.

There are three auxiliary routines to implement the operation. First, **timerOffset-PeriodSpawn()** creates the periodic task and spawns **timerOffsetPeriodRun()**. In turn, the latter creates the timer, associates it with with the routine **wakeUpPeriodic()**, assigns the specified offset and period to a value of type struct itimerspec and starts the timer, using the created composed variable. The infinite cycle at the end of timerOffset-PeriodRun() only idles the CPU to other ready tasks and keeps the routine permanently running; otherwise the timer would automatically be deleted as the routine ended. Finally wakeUpPeriodic(), which is the timer handler associated to it by means of timer_connect(), resumes the periodic task. It takes as arguments the timer ID and the ID of the periodic task to be resumed.

```
void timerOffsetPeriodSpawn(TASK_ID *tid, char *name, int offset_us, int
    period_us, int priority, int options, int stackSize, FUNCPTR entryPt, int
    arg1, ..., int arg10){
   *tid = taskCreate(name, priority, options, stackSize, entryPt, arg1, ...,
       arg10);
   taskSpawn((char *)NULL, 0, 0x100, 2000, timerOffsetPeriodRun, *tid,
       offset_us, period_us, 0, ..., 0);
}

void timerOffsetPeriodRun(TASK_ID tid, int offset_us, int period_us){
```

```
    timer_t timerid;

    struct itimerspec value;
    // set value to have the specified offset and period

    timer_create(CLOCK_REALTIME, &timerid);
    timer_connect(timerid, (VOIDFUNCPTR)wakeUpPeriodic, tid);
    timer_settime(timerid, &value);

    while(1){
        taskDelay(1);    // the routine that creates the timer
    }                    // cannot end, else the timer is canceled
}

void wakeUpPeriodic(timer_t timerid, TASK_ID tid){
    taskResume(tid);
}
```

#### Spawning tasks at random instants using a POSIX timer

Once again, we use three auxiliary functions. **timerOffsetSporadicSpawn()** has the same purpose as timerOffsetPeriodSpawn(), with the difference in the arguments it receives. **timerOffsetSporadicRun()** has a similar operation as timerOffsetPeriodRun(), but it also stores the data needed for determining a random period (lower and upper limits and step) in global variables. At the beginning of the application execution, it is desired that no requests happen before the specified offset. Thus, at assigning the itimerspec variable, the offset component is defined according to the desired offset. Since it is not intended that the sporadic task runs periodically, zero is assigned to the period component. After the offset has passed, it is not intended that the first instance of the sporadic task runs straight away, since this way the moment of the first instance would not be uncertain. In order to distinguish the first instance of the sporadic task from the remaining the global flag **flg_firstSporadic** is used. The routine creates the timer, connects it to **wakeUpSporadic()** and starts it. Finally, wakeUpSporadic() resumes the sporadic task in case it is not the first time it is being called. Afterwards, and independently of the case, it calculates a random interarrival time with the saved global data and assigns it to the offset component of a new itimerspec variable, while the period component remains zero. It starts the timer with that new time value. Notice that every time wakeUpSporadic() is called, a new random interarrival time is calculated, leading to non-deterministic request moments of the sporadic task, as intended.

The use of global variables implies that this function is restricted to the existence of only one sporadic task in the application. A possible solution to this problem would be the use of global vectors with user-defined dimensions (the vectors would have to allocate a number of values equivalent to the desired number of sporadic tasks), instead of single global variables.

```
int lowLim, uppLim, step, flg_firstSporadic;

void timerOffsetSporadicSpawn(TASK_ID *tid, char *name, int offset_us, int
    low_lim_us, int upp_lim_us, int step_us, int priority, int options, int
    stackSize, FUNCPTR entryPt, int arg1, ..., int arg10){
   *tid = taskCreate(name, priority, options, stackSize, entryPt, arg1, ...,
       arg10);
   taskSpawn((char *)NULL, 0, 0x100, 2000, timerOffsetSporadicRun, *tid,
       offset_us, low_lim_us, upp_lim_us, step_us, 0, ..., 0);
}

void timerOffsetSporadicRun(TASK_ID tid, int offset_us, int low_lim_us, int
    upp_lim_us, int step_us){
   timer_t timerid;
   struct itimerspec value;

   lowLim = low_lim_us;
   uppLim = upp_lim_us;
   step = step_us;

   // Set value to have the specified offset.
   // The period is zero, since a periodic time-out is not intended.

   flg_firstSporadic = 1;

   timer_create(CLOCK_REALTIME, &timerid);
   timer_connect(timerid, (VOIDFUNCPTR)wakeUpSporadic, tid);
   timer_settime(timerid, &value);

   while(1){
      taskDelay(1);
   }
}

void wakeUpSporadic(timer_t timerid, TASK_ID tid){
   int next_rqst;
   struct itimerspec value;

   if(flg_firstSporadic == 0){
      taskResume(tid);
   }

   if(flg_firstSporadic == 1){
      flg_firstSporadic = 0;
   }

   n_levels = 1 + ((uppLim - lowLim) / step);
   next_rqst = (rand() % n_levels) * step;
```

```
    next_rqst += lowLim;

    // Set value with the new offset.
    // The period remains zero, since, once a value is
    // assigned to it, it remains constant.

    timer_settime(timerid, &value);
}
```

Note 1: When using POSIX timers, if a task is executing and is requested in the meanwhile, the situation described in the note at the end of section 4.1 does not happen anymore, since no taskResume() is performed during an application task's execution. Instead, the waking task runs as soon as an opportunity arises, namely right after the blocking application task has been suspended and reactivated the scheduler.

Note 2: The solution of constantly yielding the CPU at the end of timerOffsetPeriodRun() and timerOffsetSporadicRun() using the command taskDelay(0) was used in order to avoid that the wake-up task ends, cancelling the timer. This implies a constant processing effort and introduces a significant overhead in the system. Since the wake-up tasks have a higher priority than the application routines, they will interrupt them, if the scheduler is enabled, namely in the fully-preemptive case.

## 4.3 Analysis program

As mentioned in chapter 3, System Viewer is a tool provided by Wind River Workbench and has been strongly used along this project, programmed in a mode that allows to detect and register task state changes. However, many events that are not directly related to the application are registered and, although they may help to find out the cause for eventual unexpected delays, it is intended to analyse solely the application task set in this project.

An analysis program was developed aiming to read a System Viewer log file, simplify it to show only the events related to the task set in a lighter arrangement and extract a time analysis of the global system, as well as an individual analysis of the execution times of each application task.

### 4.3.1 Simplifying the System Viewer log file

The System Viewer log file is organized in columns. The first column shows the timestamp, the second one the task that causes the event, the third one the operation, the following four columns show a few details about the event and the last column presents the parameters of the operation. The simplifying function reads that log and extracts, in another log file, the following events:

- the moment an application task is requested
- the moment an application task starts its $n^{th}$ non-preemptive subjob
- the moment an application task has its $n^{th}$ preemption point

- the moment an application task finishes an instance
- the moment of an application task's deadline

The user must provide the tasks' names, relative deadlines and number of preemption points to the analysis program. In the case of fully-preemptive systems the latter piece of information does not make sense, since the number of preemptions that a task may suffer is not known a priori. The deadline of a task is not definable in VxWorks, so it is the function's job to calculate the moment of the next deadline by adding the relative deadline to the moment of the task request.

The timestamps in the log file are relative to the moment when the registering starts. However, in order to examine the instants of the events in an easy manner, it is intended that the instant zero is the moment when main() suspends itself, allowing the application tasks to start. The interval from the beginning of the registering until the suspension of main() is referred here as **offset** and all timestamps in the output file are relative to the moment main() suspends.

In order to keep the output file as light as possible, the following symbolism was adopted:

| | |
|---|---|
| Task request | -> |
| Task finishes | - |
| Task deadline | <- |
| Task starts subjob | [number of subjob] |
| Preemption point | *[number of preemption point] |

Table 4.1: Meaning of the symbolism used in the simplified log file

As an example, suppose that three tasks run concurrently: task_1 is the highest priority task, followed by task_2 and lastly task_3. All tasks are equal in the sense that their worst case execution time is approximately 6 seconds and they contain only one preemption point in the middle of the execution (Each subjob takes approximately 3 seconds.). The task characteristics are displayed in the following table:

The resulting output file is shown in Figure 4.7.

Figure 4.8 is an illustrated interpretation of the output file:

Notice that, around the 41$^{st}$ second, task_3 has not finished executing, since at that point only the first subjob is finished. There is a task request, but, since task_3 is not in the SUSPENDED state, it has no effect and is ignored. In fact, the execution that occurs at the 54$^{th}$ second is the second subjob of the second instance (requested at 21s).

| Name | task_1 | task_2 | task_3 |
|---|---|---|---|
| Type | sporadic | periodic | periodic |
| Offset | 5s | 2s | 1s |
| Period | - | 20s | 20s |
| Lower lim. | 4s | - | - |
| Upper lim. | 10s | - | - |
| Step | 0.5s | - | - |
| Deadline | 10s | 20s | 20s |

Table 4.2: Timing characteristics of the task set in the example

```
Offset: 185.568741
1.015152      task_3 ->
1.015213      task_3 1
2.015152      task_2 ->
4.051926      task_3 *1
4.051987      task_2 1
7.088089      task_2 *1
7.088089      task_2 2
10.126663     task_2 -
10.126709     task_3 2
11.548492     task_1 ->
13.162201     task_3 -
13.162247     task_1 1
16.197250     task_1 *1
16.197250     task_1 2
19.232391     task_1 -
21.015152     task_3 <-
21.031815     task_3 ->
21.031876     task_3 1
21.548492     task_1 <-
22.015152     task_2 <-
22.031830     task_2 ->
24.066864     task_3 *1
24.066910     task_2 1
26.098495     task_1 ->
27.102844     task_2 *1
27.102905     task_1 1
30.139313     task_1 *1
30.139313     task_1 2
33.175690     task_1 -
33.175735     task_2 2
36.098495     task_1 <-
36.115158     task_1 ->
36.212158     task_2 -
36.212204     task_1 1
39.248611     task_1 *1
39.248611     task_1 2
41.031815     task_3 <-
41.048477     task_3 ->
42.031830     task_2 <-
42.048477     task_2 ->
42.283798     task_1 -
42.283844     task_2 1
45.318817     task_2 *1
45.318817     task_2 2
46.115158     task_1 <-
46.181824     task_1 ->
48.353867     task_2 -
48.353912     task_1 1
51.388870     task_1 *1
51.388870     task_1 2
54.423691     task_1 -
54.423737     task_3 2
56.181824     task_1 <-
57.459793     task_3 -
END OF LOG
```

Figure 4.7: Output file with the simplified log

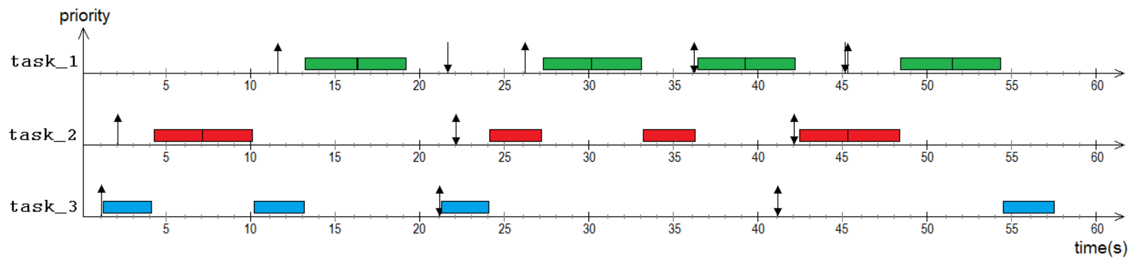



Figure 4.8: Gantt diagram of the application tasks

If the experience continued for a few more seconds, the next task_3 request would occur at instant 61s.

### 4.3.2 Time analysis of the global system

The analysis program includes a function that reads the simplified log described above and extracts a time analysis of the global system. First it lists all detected deadline misses and preemptions, indicating the concerned task and timestamp. Afterwards it presents, for each task, the following information:

- The total number of task requests, missed deadlines and the percentage of deadline

misses relatively to the number of requests.

- The total number of suffered preemptions and the average, standard deviation and maximum number of suffered preemptions per instance.
- The number of complete instances.
- The average, standard deviation and maximum of the response time (time interval between task request and finishing time).

### 4.3.3 Execution times

The analysis program was complemented with another function that reads the simplified log file and creates, for each application task, an auxiliary file with the total execution time, as well as the duration of the non-preemptive sections, in case of having preemption points. The average, standard deviation and maximum of the total execution time is displayed. In case of limited preemption scheduling, the same statistical information is provided for the last and the longest subjob in the instance.

The execution time of a task is deduced in the following manner: The moment the task starts running is registered, opening the first segment of execution. If another application routine starts running, but the task has not finished yet, it means that the task has been preempted and the segment is closed at that moment. In case of having preemption points, the segment is closed when one is found. Once the task becomes suspended it means it has finished execution. This instant closes the last segment. The total execution time is obtained by adding all segments.

In this process only application tasks are considered, but there are many short operating system tasks and interrupts that run in the system constantly. These tasks are responsible for communication between computer and board, clock tick announce, etc. and may have very high priorities. When disabling the scheduler, these tasks cannot execute. However, when a task's priority is raised to avoid preemption, these tasks may interrupt if their priority is higher. In the fully-preemptive case, where nothing is done to avoid preemption, any such system task with priority higher than the application tasks' can interrupt. These system tasks are ignored when computing the execution time of the application tasks and end up causing interference in the measurements.

# 5. Practical experiments

This chapter discusses the autonomous vehicle application. The top-level structure of the vehicle is exposed, indicating its main functionality. Afterwards, a more practical approach is made by describing with detail the task set that is actually used in the experiments.

## 5.1 The autonomous vehicle

This section provides background information on the case study that motivated this work, namely the SENA autonomous vehicle, to make the reader aware of its main features.

### 5.1.1 Software architecture

In terms of software, the system is divided in three levels: A deliberative layer (upper level), a reactive layer (lower level) and a middle layer. Figure 5.1 provides a visual representation of the software architecture. All three levels count on an integrity and security monitoring unit.

The low level is responsible for every action directly related to the external interface of the vehicle. One important function is sampling outside data by means of sensors. For this purpose, this layer is provided with laser sensors and an inertial unit. The data are received and processed and the results are sent to a low level sensor fusion unit. Another function is the actuation, changing the behaviour of the vehicle. For this function the system counts with different kinds of actuators, which are directly controlled by smart power drivers (approximately six units). These drivers receive commands from the upper layers (velocity, position, etc.) and respond with information about the actuators' state. As mentioned before, these drivers are smart, meaning that they receive a command with a desired velocity or position and calculate the necessary voltage and current to be applied to the actuators in a local feedback loop.

The middle level provides an abstraction layer for the upper level. Here the sensor data fusion results of the lower layer are interpreted by a similar but more abstracted unit.

The upper layer is the one that makes the general decisions. It receives the results of the middle layer sensor fusion and further refines it to an even more abstract vehicle representation. Here, a suitable sensor interpretation is carried out to be sent to the mapping

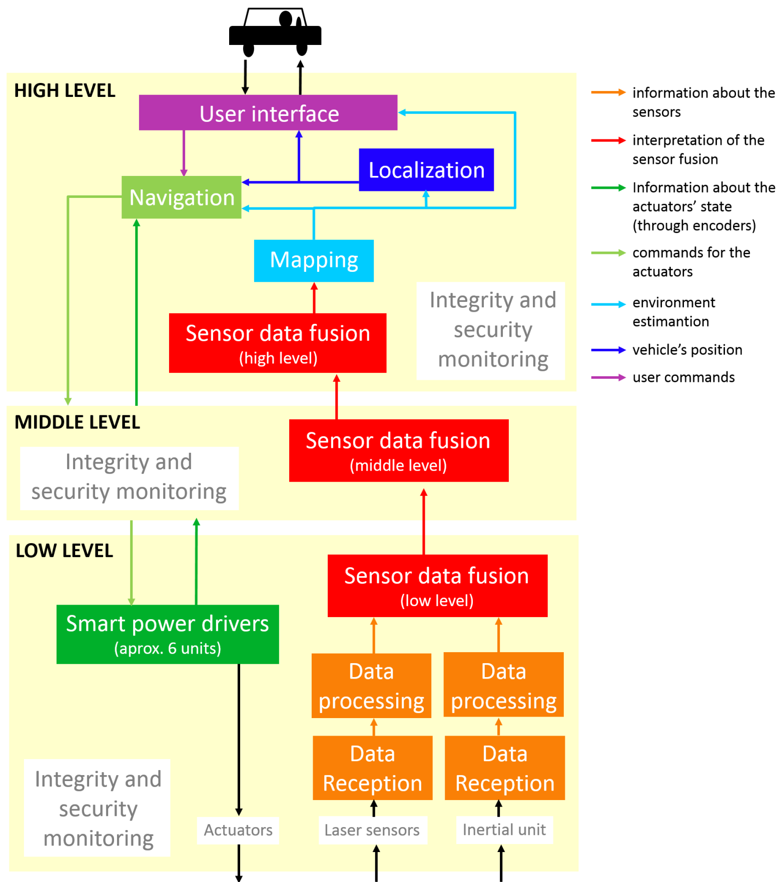



Figure 5.1: SENA software architecture

unit, which estimates the surroundings of the vehicle. This information helps the localization of the car and the navigation unit that makes the driving decisions. This unit also receives commands from the user and, depending on the safety level of each concrete situation, decides whether to follow the driver's commands or to take autonomous decisions. The navigation unit knows the state of the actuators and issues actuation commands according to what was decided. Finally the upper layer also includes an user interface. The driver has access to the information derived from the mapping and localization units and may send his/her commands to the navigation unit.

For this kind of systems to work properly, the various tasks involved in the feedback control loops must be activated periodically with relatively low jitter. Furthermore the interference and blocking each task suffers must be small enough for the task to terminate its execution before its deadline.

The individual SENA features have been finished or are under development by several programmers in a collaborative fashion, each focusing on a different subsystem. It is desired that these subsystems then become consolidated in the same platform in a reliable and deterministic manner using the real-time multitasking capabilities of the RTOS. At this moment the team is focused on progressively taking a step apart from the existing desktop implementation of the autonomous vehicle to a more integrated, embedded approach.

The presented architecture includes several functionalities. In this experiment only a few features were taken into account, namely six tasks that provide input for the actuators, one task for localization and a set of two tasks which sporadically change the parameters of one of the six tasks mentioned before.

Since the SENA project is under development, a few tasks are not ready to run on VxWorks yet. The reason for that is that they contain certain operations, such as Ethernet communication commands and floating point manipulation, whose support is not included in the current configuration of the board PowerQuicc II. Due to lack of time, neither the board was configured to support those features nor the functions were adapted to avoid those requirements. The considered alternative was to test the limited preemption technique with tasks that do not perform the operations that the real SENA tasks do, yet provide a similar behaviour towards the other application tasks in the sense that they have similar (or at least plausible) periods and execution times.

## 5.2 Experimental task set

This section focuses on the practical task set used for experimenting. After describing the usage given to the tools mentioned in chapter 3, it provides details on each task's implementation and information about the chosen timing characteristics, such as offsets and periods.

### 5.2.1 Tools usage

For the configuration of the operating system a **VxWorks Image Project** was created and the required components for the support of System Viewer and a particular console were added to the image. The application was created within a **Downloadable Kernel Module** and was set to run in the PowerQuicc II board. The communication between computer and board is mainly achieved by an Ethernet connection. The chosen system clock rate is 2000Hz, which grants a system clock resolution of 500$\mu$s. System Viewer was configured to register task state change events in a buffer with a memory capacity of 512Kbyte.

### 5.2.2 The task set

This section discusses the task set that served as basis for the experiments, including their implementation and the chosen timing characteristics (offsets and periods).

#### tLocal - A task for vehicle localization

The localization routine to be used in SENA is based on the Markov algorithm. It receives a two-dimensional map, the sensors of the car (what the vehicle can see) and the last movement of the car, including distance and direction. The successful localization of the vehicle consists of determining three parameters: The x and y component of the position in the map and the car's orientation. Depending on the dimensions and resolutions of the map, and also the resolution of the orientation angle, this process can be more or less

longstanding. For each possible combination of those factors the probability of the vehicle to be in that spot with that orientation is calculated, making the number of probability calculations the following:

$$n_{\text{ProbablilityCalculations}} = \frac{totalSize_x}{step_x} \cdot \frac{totalSize_y}{step_y} \cdot \frac{2\pi}{step_{\text{angle}}} \quad (5.1)$$

where the step is inversely proportional to the resolution. The total execution time of this function can be manipulated by experimenting with different values of the map sizes and steps.

Although a version of this function is ready and functional by this time, for it to execute in VxWorks it would be necessary to configure the operating system to support certain required features, or, alternatively, adapt the localization function to not needing these features. Due to lack of time, none of the alternatives was fulfilled and, instead, a simplified and illustrative version of this function was implemented. In this version only two dimensions are considered, rather than three, and, instead of performing a probability calculation, other senseless operations are made, with the only purpose of consuming processor time. Also, instead of having a map, there is only one vector which is crossed from beginning to end in the outer loop in the contrary direction in the inner loop. The performed calculations consist in swapping the values in the positions indicated by the outer and inner iterators.

```
/* Defines for the localization function */
#define LOCAL_VEC_SIZE 544 // defined accordingly to the desired execution time
    of the task

/* Localization function */
void tLocalization(void){
   int i, j, vec[LOCAL_VEC_SIZE], aux;
   while(1){
      // initialize the vector
      for(i=0; i<LOCAL_VEC_SIZE; i++){
         vec[i] = i;
      }
      for(i = 0; i < LOCAL_VEC_SIZE; i++){
         for(j = LOCAL_VEC_SIZE-1; j >= 0; j--){
            // swap vec[i] with vec[j]
            aux = vec[j];
            vec[j] = vec[i];
            vec[i] = aux;
         }
      }
      taskSuspend(0);
   }
}
```

### Placing of preemption points

Since this function can take long execution times, preemption points are placed in its code. In fact, it will soon become clear that it is the only task with an execution time worth to be divided, since all other tasks have relatively short execution times. The idea is to test the behaviour of the system for non-preemptive subjobs of different sizes. The most preemptions points tLocal has, the smaller are the non-preemptive sections, since we place them approximately evenly along the (unfolded) task's code, as the following example shows.

```
#define LOCAL_N_PP 3
#define LOCAL_VEC_SIZE 544 // defined accordingly to the desired execution time
    of the task

void tLocalization(void)
{
// ...
   while(1){
      // ...
      for(i = 0; i < LOCAL_VEC_SIZE; i++){
         for(j = LOCAL_VEC_SIZE-1; j >= 0; j--){
            // swap elements
         }

         if (i % (LOCAL_VEC_SIZE/(LOCAL_N_PP+1)) == 0){
            if(i > 0){
               if(i > LOCAL_N_PP*(LOCAL_VEC_SIZE/(LOCAL_N_PP+1))){
                  continue;
               }
               preempPoint(...);
               startSubjob(...);
            }
         }
      }
      // ...
   }
}
```

Depending of the value of the outer iterator i, a preemption point shall or not happen. Let us recall that a task with X preemption points is divided in X+1 non-preemptive subjobs.

Since the task is divided in sections of approximately similar sizes, the value of i at which a preemption point happens must be a multiple of the total vector size divided by the desired number of non-preemptive sections (number of preemption points (LOCAL_N_PP) plus one), and that is what the first if-statement suggests. Since this condition is satisfied for a null value of i and it is not desired that a preemption point happens right after the task begins, the second if-statement filters out i's initial zero value. Finally,

the third if-statement has the following purpose: Suppose that LOCAL_VEC_SIZE is 100 and LOCAL_N_PP is two – It is desired that the task is divided by three similar non-preemptive sections. However, 100 divided by 3 is 33.333(...), but, since an iterator only takes integer values, the preemption point happens for multiples of 33, in this case, i = 33, i = 66 and i = 99. In the cases where the total number of iterations is multiple of the number of subjobs, for i equal to the total number of iterations (which is a multiple of LOCAL_VEC_SIZE/(LOCAL_N_PP+1)) no preemption point occurs, since this value is out of the loop scope (The scope includes values from zero to LOCAL_VEC_SIZE-1). However, due to the truncation of, in this case, 33.333(...), the total number if iterations is never a multiple of that amount, but a slightly smaller value is, which lies within the loop scope, thus causing an undesired preemption point. The third if-statement jumps straight to the next iteration immediately after the last desired preemption point occurs, thereby avoiding a preemption point too close to the end of the execution.

In the experiment, tLocal embraces as many preemption points as needed in order to have zero deadline misses, i.e., until the system is feasible.

#### tPID_X - A PID for each axis of the vehicle

The vehicle has six axes, being each one of them individually controlled by a PID controller. This task, which is instantiated six times with different names (tPID_1, tPID_2, ..., tPID_6) performs the common PID operations, namely calculating the proportional, integrative and derivative components and adding them to form the output. Each parameter, input and output of the PIDs is organized through a vector, being the data in the first position concerned to tPID_1, the second to tPID_2 and so on. Thereby, and taking advantage of the fact that all PIDs perform the same operations only with different values, instead of having six individual functions, there is only one common function. This function receives as only parameter the index of the vector: For tPID_1 the index is zero and for tPID_6 it is five. This allows the function to know what data to read (gains, reference and sensor input) and the vector location where it should write the output. The main() function creates six tasks with taskCreate(), assigning tPID_1, tPID_2, etc. to the name, the function tPID to the entry point and the corresponding vector index (0, 1, ...) as a parameter to be passed to the entry point.

```
#define N_AXIS 6

/* Global variables for the PIDs */
int Kp[N_AXIS] = {10, 60, 30, 70, 40, 50};
int Ki[N_AXIS] = {20, 10, 40, 30, 100, 30};
int Kd[N_AXIS] = {50, 80, 20, 90, 60, 70};
int reference[N_AXIS] = {100, 200, 300, 400, 500, 600};
int input[N_AXIS] = {50, 150, 250, 350, 450, 550};
int output[N_AXIS];
```

```
/* IDs, offsets, periods and priorities */
TASK_ID id_PIDs[N_AXIS];
int offsetPID[N_AXIS] = {500, 500, 500, 500, 500, 500};
int periodPID[N_AXIS] = {3000, 5000, 5500, 6000, 6500, 7000};
int priorityPID[N_AXIS] = {3, 4, 5, 6, 7, 8};

/* PID function */
void tPID(int index){
   int error, error_sum = 0, error_delta, error_prev = reference[index];
   while(1){
      error = reference[index] - input[index];
      error_sum += error;
      error_delta = error - error_prev;
      output[index] =
          (Kp[index]*error)+(Ki[index]*error_sum)+(Kd[index]*error_delta);
      error_prev = error;

      taskSuspend(0);
   }
}

/* Main function */
int main(){
   ...
   int i;
   char pid_name[10] = "tPID_", num[2];
   for(i=0; i<N_AXIS; i++){
      sprintf(num, "%d", i+1);      // turn integer into string
      strcat(pid_name, num);        // concatenate PID_ and number
      id_PIDs[i] = offsetPeriodSpawn(pid_name, offsetPID[i], periodPID[i],
         priorityPID[i], 0x100, 2000, tPID, i, 0, 0, 0, 0, 0, 0, 0, 0, 0);
      strcpy(pid_name, "tPID_");
   }
   ...
}
```

### tChange and tUpdate - Tasks that change the PID gains sporadically

In real-life applications, it is common to have a task running in a different processor that, knowing the surroundings of the vehicle, estimates that a certain controller is not adequate for the upcoming situation. The concerned controller, which resides in the application, is than replaced. For simplicity purposes, instead of having the function of the controller task completely replaced (this would imply deleting the current task and creating a new one with a different entry point function), only the proportional, integrative and derivative gains are replaced by new ones.

In order to decide what PID to change and what the new gains shall be, the task

tChange uses nothing but random calculations by means of stdlib.h's function rand(). Since in real-life applications the moments where a controller needs to be changed is not predictable, this task shall be called at non-deterministic moments. For this purpose the functions offsetSporadicSpawn() or timerOffsetSporadicSpawn() are to be used.

Simply replacing the values in the gain vectors is not a good real-life solution, since this could cause an abrupt change in the output to the actuator, possibly damaging the dynamic system. In order to avoid that, it is desired that the gain changes in a softer manner, as illustrated in Figure 5.2.

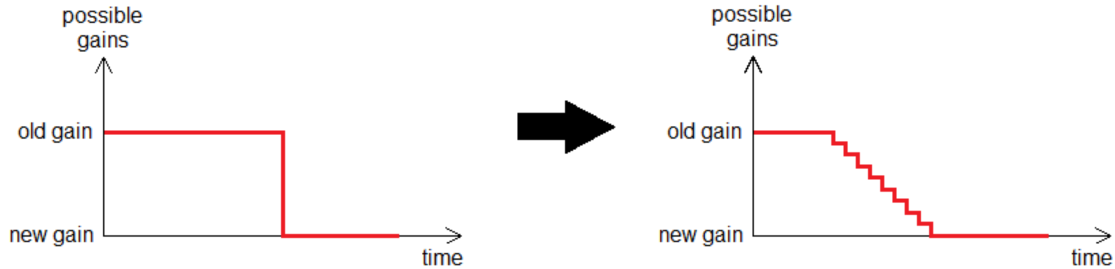



Figure 5.2: Abrupt (left) and linear (right) gain change

In order to obtain a gradual change of gains, the task tChange, after randomly determining the PID to be altered and the new proportional, integrative and derivative gains, calls another task, tUpdate, which periodically runs for a limited number of times (the function runNTimes() is used for this purpose) and gradually updates the gains in a linear manner. Figure 5.3 shows an example of a possible situation.

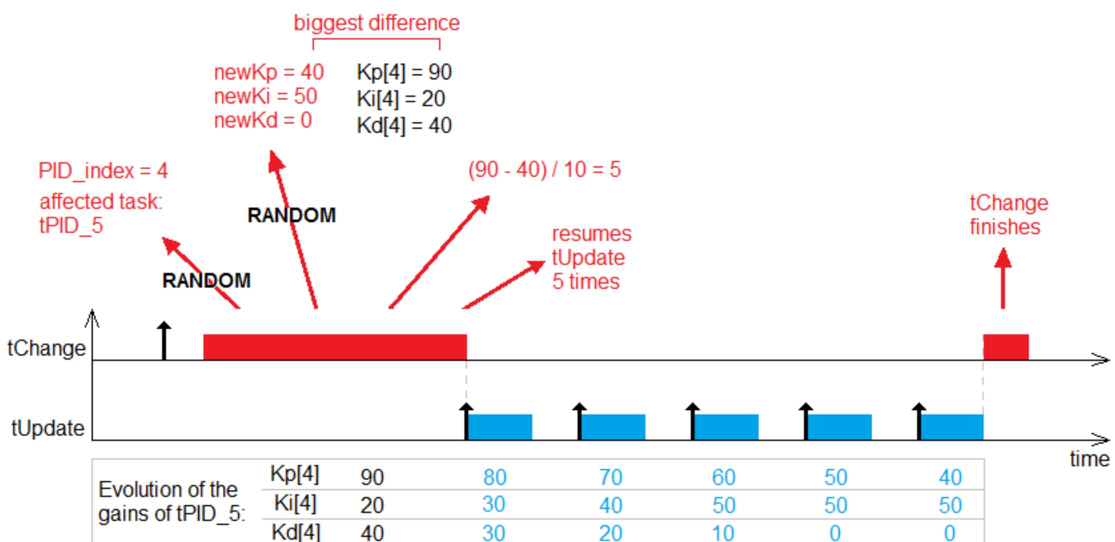



Figure 5.3: Example of tChange and tUpdate operations

In the example tChange randomly chooses tPID_5 to be altered and the values 40, 50 and 0 for the new proportional, integrative and derivative gains, respectively. Assuming that the current gains are 90, 20 and 40, respecting the same order, and the desired step for the linear progress is 10, tChange calculates the biggest difference between new and old values and divides by the step. The result corresponds to the required number of tUpdate

executions until all gains are updated. Notice that the integrative gain takes its final value already at the third execution of tUpdate and the derivative gain at the fourth.

```
/* Global variables for the gain updating functions */
int newKp, newKi, newKd;
int step;
int index;
int offset_update, period_update;

void tChangeRqst(void){
   int difKp, difKi, difKd, maxDif;
   int n_times;

   while(1){
      // obtains index of the PID to be changed and gains
      index = random() % 6;             // possible indexes: 0, 1, ..., 5
      Kp = ((random() % 10) + 1) * 10;  // possible gains: 0, 10, 20, ..., 100
      Ki = ((random() % 10) + 1) * 10;
      Kd = ((random() % 10) + 1) * 10;

      // obtain absolute difference between old and new values
      difKp = abs(newKp - Kp[index]);
      difKi = abs(newKi - Ki[index]);
      difKp = abs(newKd - Kd[index]);

      // obtain maximum difference
      maxDif = max(difKp, difKi, difKp);

      // number of required times for tUpdate to execute
      n_times = maxDif / step;

      // calls tUpdate repeatedly
      runNTimes(offset_update, period_update, idUpdate, n_times);

      taskSuspend(0);
   }
}

void tUpdateGains(void){
   while(1){
      if(Kp[index] > newKp){
         Kp[index] = Kp[index] - step;
      }
      else if(Kp[index] < newKp){
         Kp[index] = Kp[index] + step;
```

```
        }
        // do the same for the integrative and proportional gains

        taskSuspend(0);
    }
}
```

After obtaining the PID to update and the new gains, tChange becomes tUpdate's waking task, exclusively resuming it periodically. Upon measuring tChange's execution time, it is considered that it ends execution after resuming tUpdate for the first time. This convention was adopted as no other tasks' waking routine's execution time is being considered either.

### Offsets and periods

The tables below expose the chosen timing values and priorities of the application tasks. These values are the ones that are passed to the waking routines.

**Periodic tasks**

| Task name | Offset ($\mu$s) | Period ($\mu$s) | Priority |
|---|---|---|---|
| tPID_1 | 500 | 3000 | 3 |
| tPID_2 | 500 | 5000 | 4 |
| tPID_3 | 500 | 5500 | 5 |
| tPID_4 | 500 | 6000 | 6 |
| tPID_5 | 500 | 6500 | 7 |
| tPID_6 | 500 | 7000 | 8 |
| tUpdate | 500 | 10000 | 10 |
| tLocal | 500 | 40500 | 11 |

Table 5.1: Timing characteristics and priorities of the periodic tasks

Notice that tUpdate is considered a periodic task in the sense that, when it is set to execute, it is requested with regular intervals.

**Sporadic tasks**

| Task name | Offset ($\mu$s) | Minimum interarrival time ($\mu$s) | Maximum interarrival time ($\mu$s) | Step ($\mu$s) | Priority |
|---|---|---|---|---|---|
| tChange | 500 | 300 000 | 800 000 | 50 000 | 9 |

Table 5.2: Timing characteristics and priorities of the sporadic tasks

Recalling the meaning of these terms, tChange is requested sporadically and the time between two consecutive requests belongs to the set 300, 350, 400, 450, ..., 750 and 800 milliseconds.

# 6. Experiments and results

Along this project three types of experiments were made. First, in order to inflict non-preemptivity, the priority of the tasks is increased (experiment 1). Afterwards, the same goal is achieved by disabling the scheduler (experiment 2). In the first two experiments, the periodic and sporadic tasks are set to run at the proper moments using the wake-up routines offsetPeriodSpawn(), runNTimes() and offsetSporadicSpawn(), described in sections 4.2.4, 4.1.3 and 4.1.4 respectively. Finally, and since disabling the scheduler causes the tasks to desynchronize relatively to their periods, as explained in section 4.2.4, in order to avoid that problem, the waking functions timerOffsetPeriodSpawn() and timerOffsetSporadicSpawn(), described in sections 4.2.4 and 4.2.4, are used (experiment 3).

For each of the tree groups, two experiments were made: At first, a task set exactly as described in section 5.2.2 is tested. As demonstrated in this chapter, the tasks tPID_x, tChange and tUpdate have very short execution times (short tasks), thus consisting in a small interference for the long, low priority task tLocal. In order to increase the impact of preemptions, the tasks tPID_x and tUpdate are artificially expanded by adding a futile loop with as many iterations as wanted to achieve a certain execution time (long tasks). At the end of each of the three experiments, the results are briefly discussed.

An experiment consists of setting the task set to run in the following conditions:

- **Non-preemptively:** Once a task starts executing, no other task may preempt it, independently of its priority.

- **With preemption points:** The low priority, time-consuming task tLocal is evenly divided in non-preemptive subjobs by preemption points until no task in the set misses their deadlines.

- **Fully-preemptively:** A task may interrupt any other task that has a lower priority than its own.

All presented graphs in this chapter include all these stages of the experiment: In the x-axis *N.P.* stands for **non-preemptive**, *F.P.* for **fully-preemptive** and all the numerical values in between stand for the number of preemption points placed in tLocal. The last number before F.P. corresponds to the necessary number of preemption points needed in tLocal in order to keep the task set feasible.

For each experiment the execution times of the tasks, measured along the tests, are presented. In case of tLocal, not only the total execution time is brought up, but also the execution time of the longest non-preemptive subjob in the instance is shown. For the fully-preemptive case, where no subjobs are featured, the equivalent result corresponds to the maximum interval in the instance during which tLocal suffered no preemptions.

Afterwards the behaviour of the system is evaluated by analysing the following parameters:

- **Deadline miss rate**, i.e. number of deadline misses relatively to the total number of requests. The system is considered feasible when no task misses its deadline (0% of deadline misses).
- **Response time**, i.e. interval between the moment of the request and the task's finishing time. It is desired that the response time of the tasks is as short as possible, without compromising the feasibility of the system.
- In case of the time-consuming task tLocal, the **number of suffered preemptions per instance**.

Ultimately, the response times observed in the experiments are compared to theoretical estimations based on the worst-case response time analysis presented in sections 2.2.2 (non- and limited-preemptive cases) and 2.2.1 (fully-preemptive case). This analysis normally considers worst-case execution times of the tasks and non-preemptive subjobs. However, because we observed large and rare worst-case execution times and we were interested in analytical results that would be tighter, i.e., closer to the average case, we took the average execution times as inputs for those calculations. Consequently, the estimations are not absolute upper bounds of the worst-case response times and there are cases in which the estimations are even optimistic in the sense that they provide a value that is lower than the observed one. Nevertheless, we believe this value represents better the actual temporal behaviour of the task and thus we use it in this section.

In experiment 1 (priority manipulation) and 2 (scheduler locking and spawning routines using taskDelay()) the tasks are set to execute for about one minute in the non- and limited-preemptive cases and nearly 30 seconds more in the fully-preemptive situation. In experiment 3 (scheduler locking and spawning routines using timers) the task set executes for about 20 seconds in all situations. The duration of the experiments is related to the size of the System Viewer buffer, where the timestamped events are registered. The fact that it takes a short time to fill the buffer denotes a large event density. Thus, one concludes that the implementation in experiment 3 causes more events per second than the ones in the other experiments. In fact this gap happens due to the events corresponding to the constant taskDelay() actions performed by the waking tasks, as explained in Note 2 at the end of section 4.2.4.

When comparing the behaviour of the PID tasks, darker colours are used to mark the PIDs with shorter periods and higher priorities, while lighter colours are used to mark the lower priority and less frequent ones.

## 6.1 Experiment 1 - Changing priorities

### 6.1.1 Short tasks

This task set is composed by the original application tasks.

#### Execution times

In this experiment the PID tasks take average execution times around 40$\mu$s and maximum values up to 200$\mu$s. In the fully-preemptive case they execute for an average of 15$\mu$s and maximum under 50$\mu$s. Such differences may arise from the overhead caused by the taskPrioritySet() commands. See Figure 6.1.

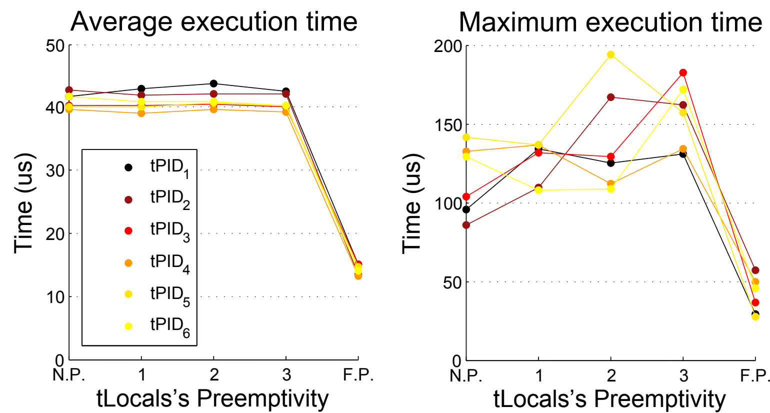



Figure 6.1: Exp. 1, short tasks - Execution times of the PID tasks

Figure 6.2 shows the execution times of the sporadic set tChange and tUpdate. tChange executes for approximately 30$\mu$s with maximums of 120$\mu$s. tUpdate has an average execution time of 30$\mu$s, maximum 110$\mu$s, except for the fully-preemptive case, where both maximum and average values lie around 20$\mu$s.

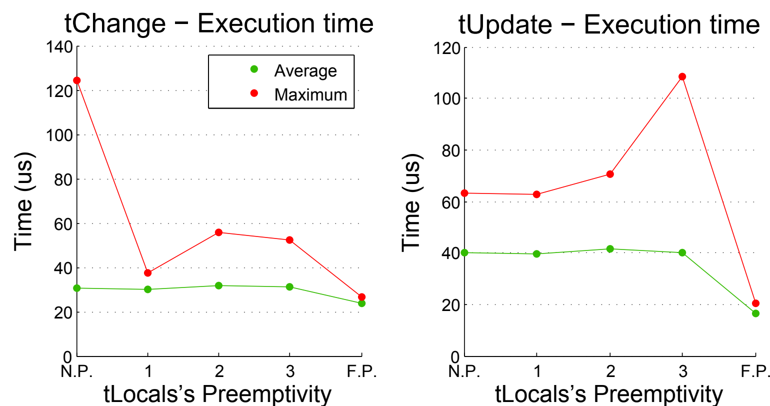



Figure 6.2: Exp. 1, short tasks - Execution times of tChange and tUpdate

tLocal executes for an average between 8650$\mu$s and 8780$\mu$s, with maximum values between 8830$\mu$s and 8970$\mu$s (Figure 6.3).

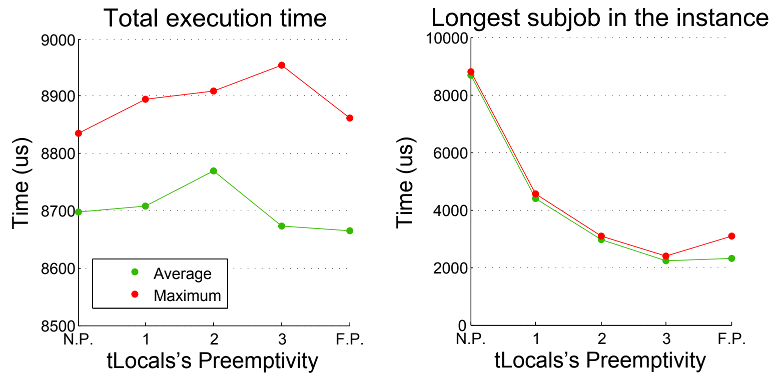



Figure 6.3: Exp. 1, short tasks - Execution times of tLocal (total value and longest subjob)

### Results

The deadline miss rate of the PID tasks, which are the only tasks that suffer deadline misses in all experiments, is shown in Figure 6.4. tPID_1, having the shortest period, is the task that suffers more deadline misses. In fact, one can see in the graph that, the higher the priority (the shorter the periods), more deadlines are missed. At 3PPs no task misses a deadline and the task set is considered feasible.

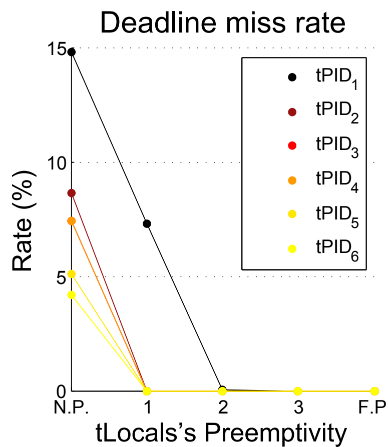



Figure 6.4: Experience 1, short tasks - Deadline miss rate of the PID tasks

Figure 6.5 illustrates the PID's response times. The average and maximum values decrease as PPs are placed in tLocal. Since the PIDs are relatively short tasks, their maximum response times are not much longer than tLocal's maximum subjob. The average response time is much shorter than the maximum. In fact, since tLocal has a relatively big period, the PIDs are only seldom affected by it. During the remaining time they are mainly affected by the higher priority PIDs. The best behaviour is observed in the fully-preemptive case.

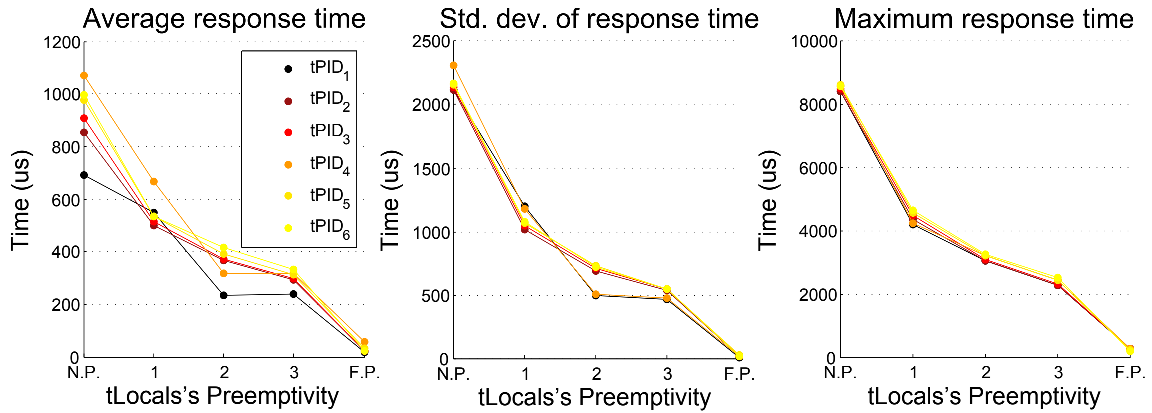



Figure 6.5: Exp. 1, short tasks - Response time of the PID tasks

tChange's maximum response time takes 8000$\mu$s for the non-preemptive case, suggesting that the task is affected by tLocal at least once. The response time decreases as PPs are added. As to tUpdate, the response time is always smaller than 500$\mu$s, suggesting that it was not or only partially blocked by tLocal. These results are shown in Figure 6.6.

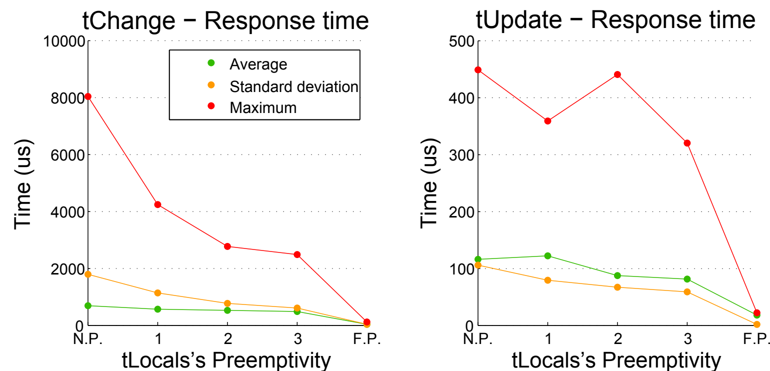



Figure 6.6: Exp. 1, short tasks - Response time of tChange and tUpdate

Figure 6.7 shows tLocal's response time and number of preemptions per instance. As expected, the response time increases as PPs are added to its own code. It reaches a maximum of 9500$\mu$s for 2 and 3PPs and decreases to 9100$\mu$s in the fully-preemptive case. This behaviour is not expected, since, as soon as the last subjob starts, there is no interference of higher priority tasks to expanding tLocal's response time, while in the fully-preemptive case this restriction is not imposed. As shown in the graphic in the right, tLocal suffers an average of 7 preemptions per instance in the fully-preemptive case. The fact that, for the limited-preemptive cases, the maximum number of preemptions per instance is one unit greater than the number of PPs in the code is explained in the note at the end of section 4.2.3. These unintended preemptions do not happen often, as the average number of preemptions per instance is very close to the number of preemption points.

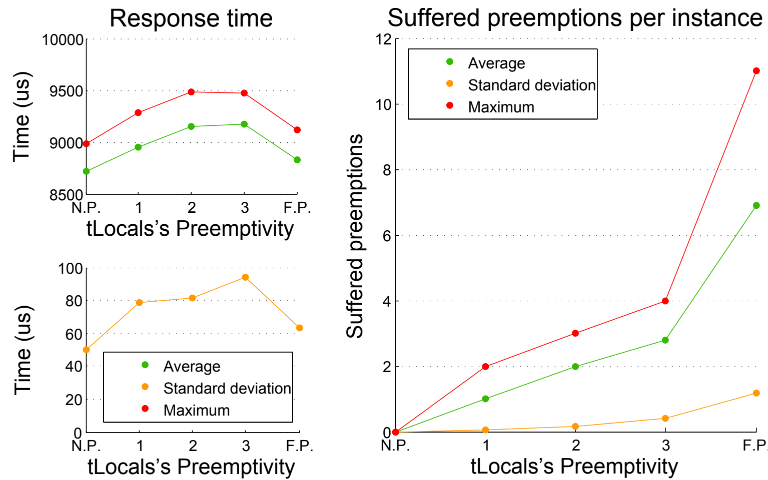



Figure 6.7: Exp. 1, short tasks - Response time and suffered preemptions of tLocal

### Comparison with theoretical response time

Figure 6.8 illustrates all task's theoretical estimations of their worst-case response times for the different cases. The worst-case response times observed for the PIDs and tChange is similar to the ones obtained theoretically. tUpdate has much smaller maximum response times in practice, possibly due to the fact that it is not strongly affected by tLocal's non-preemptive sections. As for tLocal, with the addition of preemption points the maximum response time grows more in practice than the observed in theoretical estimations. Concerning this task, in the fully-preemptive case a longer maximum response time is observed in practice, while in theory the response time is lower than for the non-preemptive situation. The reason for that may be related to the tightness of the estimations: the response time estimation for preemptive systems is likely to provide tighter results than the analysis for fixed preemption points. Also, the theoretical analysis does not consider overheads due to context switching, such as cache interference. In addition to that, if the experiment had lasted longer, eventually all worst-case situations would occur, causing maximum response times to be closer to the theoretical estimations.

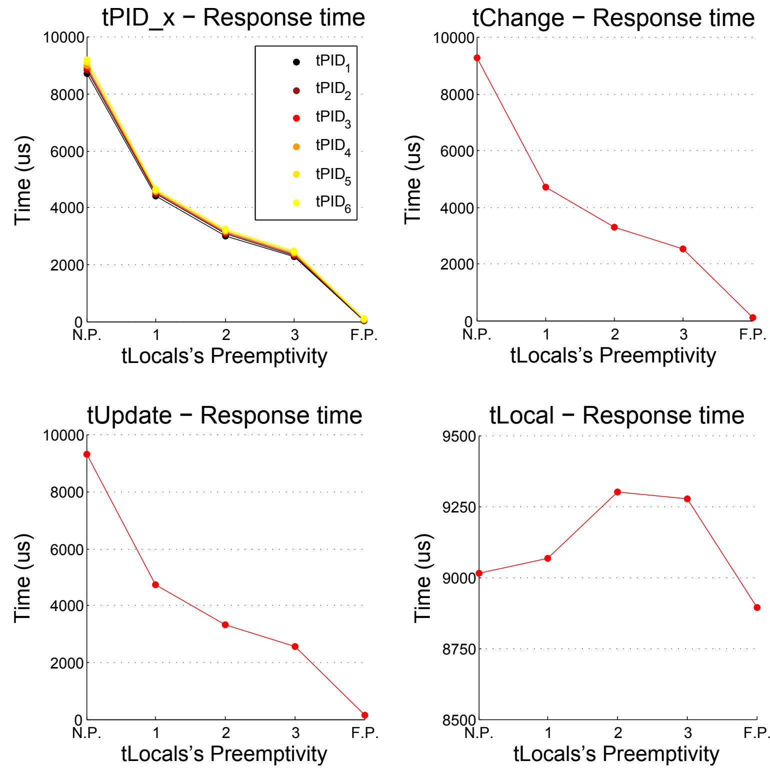



Figure 6.8: Exp. 1, short tasks - Theoretical response times

### 6.1.2 Long tasks

The previous experiment has shown that the overhead of the operations responsible for limiting preemption have a greater impact than the high priority tasks' interference and context switch overheads. In order to increase the interference, the PIDs and tUpdate have been enlarged by means of a finite loop whose only purpose is to force processing.

#### Execution times

The average execution time of the PIDs lies around 485$\mu$s for the non- and limited-preemptive cases. In the fully-preemptive situation it decreases to 440$\mu$s. Maximum values are observed between 580$\mu$s and 670$\mu$s. (Figure 6.9)

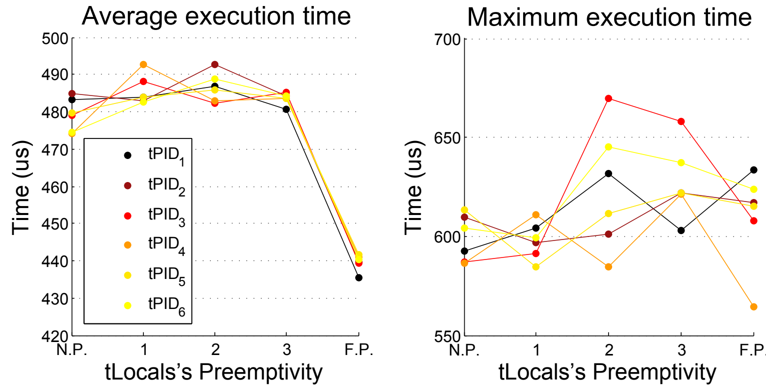



Figure 6.9: Exp. 1, long tasks - Execution times of the PID tasks

tChange has short execution times (average 30$\mu$s and maximum 90$\mu$s) and tUpdate executes for an average of 470$\mu$s and maximum of 590$\mu$s. In the fully-preemptive case the latter task reaches a maximum of almost 700$\mu$s. The execution times of tChange and tUpdate are illustrated in Figure 6.10.

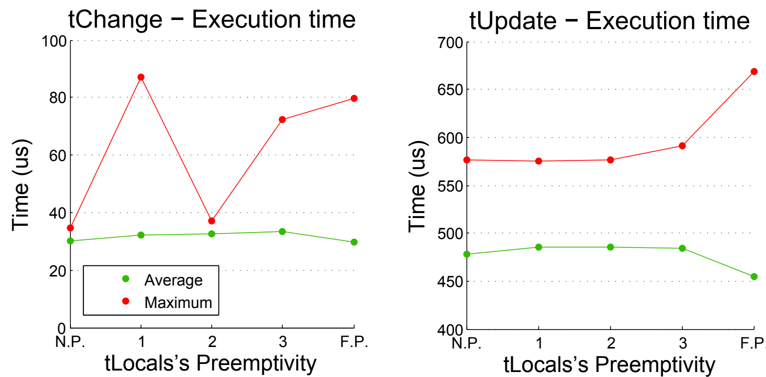



Figure 6.10: Exp. 1, long tasks - Execution times of tChange and tUpdate

Figure 6.11 shows the execution time of the time consuming task tLocal. It indicates that the execution times lie between 7600$\mu$s and 8000$\mu$s and, in the fully-preemptive case, it features a maximum of 8600$\mu$s.

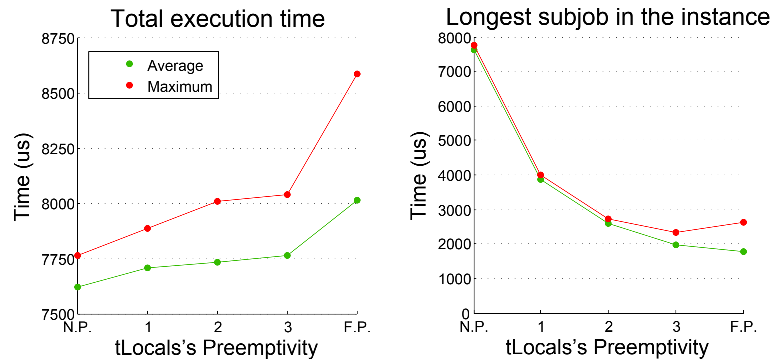



Figure 6.11: Exp. 1, long tasks - Execution times of tLocal (total value and longest subjob)

### Results

Comparing to the short task experiment, the PIDs have higher deadline miss rates and also reach zero misses at 3PPs. Once again tPID_1 stands out with a slightly greater deadline miss rate, due to its short period. As to the remaining PIDs, higher priority ones have a smaller number of misses. This suggests that lower priority PIDs lost their deadlines partially because higher priority PIDs executed instead (interference). The deadline miss rate of the PIDs is represented in Figure 6.12.

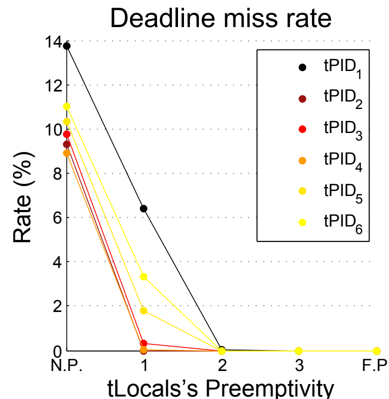



Figure 6.12: Exp. 1, long tasks - Deadline miss rate of the PID tasks

Figure 6.13 shows the average, standard deviation and maximum response time of the PID tasks. Once again, and as expected, the three values decrease as PPs are placed in tLocal. In contrast to the short task case, relevant differences are observed among PIDs: Higher priority tasks have shorter response times than lower priority ones. For the non-preemptive case the PIDs have a worst-case response time between 7900$\mu$s and 12700$\mu$s and in the fully-preemptive case between 1000$\mu$s and 3300$\mu$s, approximately.

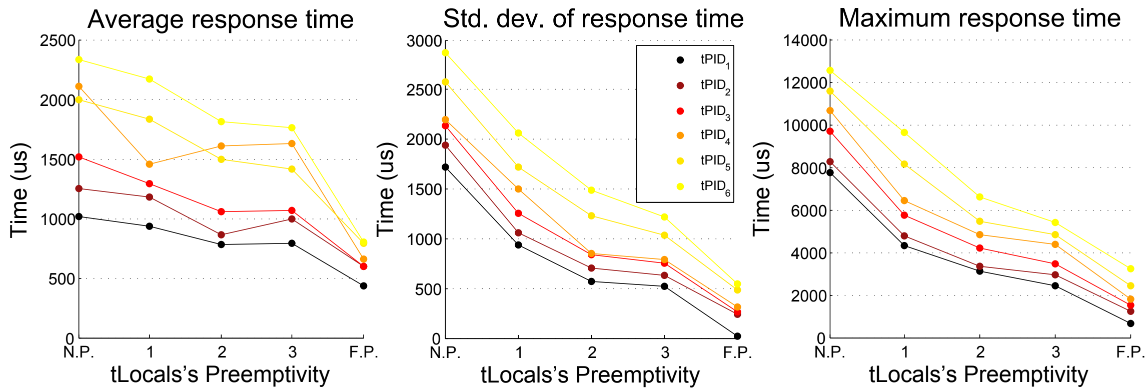



Figure 6.13: Exp. 1, long tasks - Response time of the PID tasks

Judging the maximum response times, both tChange and tUpdate are affected by tLocal at some point of the experiment. tChange has a worst-case response time of 13000$\mu$s. It decreases linearly as PPs are added and reaches 2000$\mu$s in the fully-preemptive case. In the non- and limited-preemptive situations the maximum response time of tUpdate varies in the range between 5000$\mu$s and 7500$\mu$s and, in the fully-preemptive case, it decreases to 3000$\mu$s. (Figure 6.14)

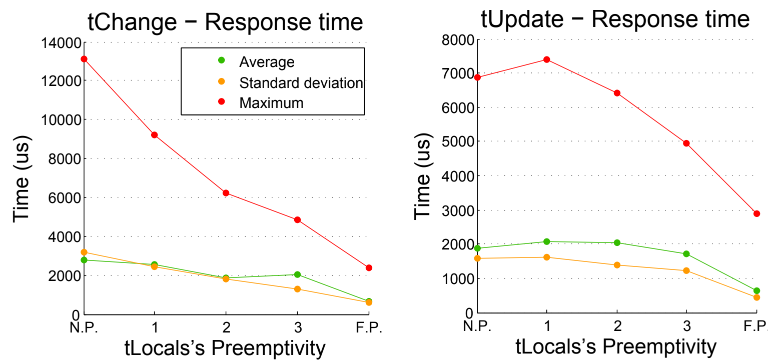



Figure 6.14: Exp. 1, long tasks - Response time of tChange and tUpdate

In contrast to the short task case, having longer PIDs and tUpdate causes the response time of tLocal to have similar values in the cases of fully preemption and with 3PPs (around 21000$\mu$s). For unlimited preemption tLocal suffers an average of 12 preemptions per instance. The response time and preemptions of tLocal are shown in Figure 6.15.

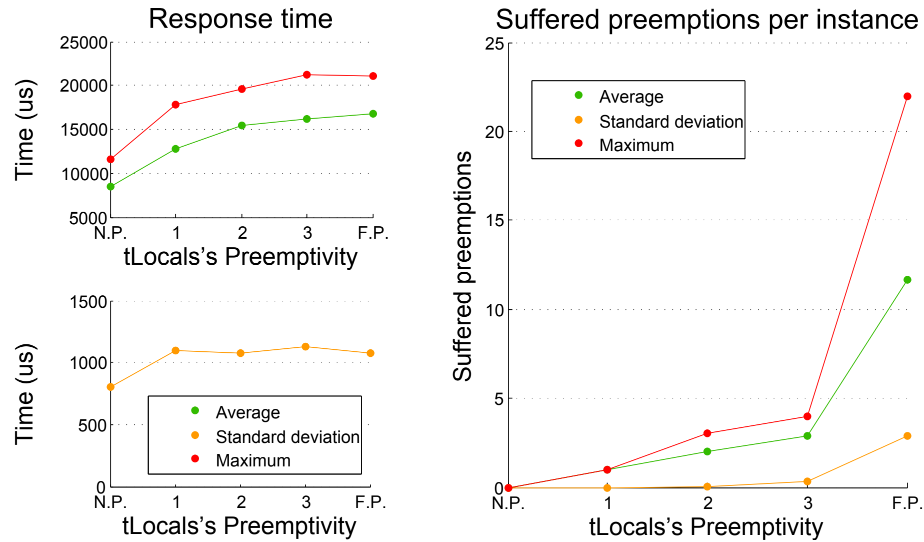



Figure 6.15: Exp. 1, long tasks - Response time and preemptions of tLocal

### Comparison with theoretical response time

The theoretical estimations of the tasks' response time are represented in Figure 6.16. For the non-preemptive case the theoretical response times of the PIDs are pessimistic compared to the practical worst-case results. They become optimistic as preemption points are placed in tLocal, becoming shorter than the observed in practice for high preemption levels of tLocal. The reason for that probably lies on the fact that the theoretical analysis does not consider the overheads due to preemptions. Both tChange and tUpdate present much longer theoretical response times than in practice. A possible reason for that is that the theoretical calculations provide worst-case response times, considering possible situations where the analysed task needs to wait the longest combination of blocking and interference that may happen with the given set. The actual worst-case response time of tLocal is very similar to the estimation, except for the fully-preemptive case, where it is shorter in practice than theoretically.

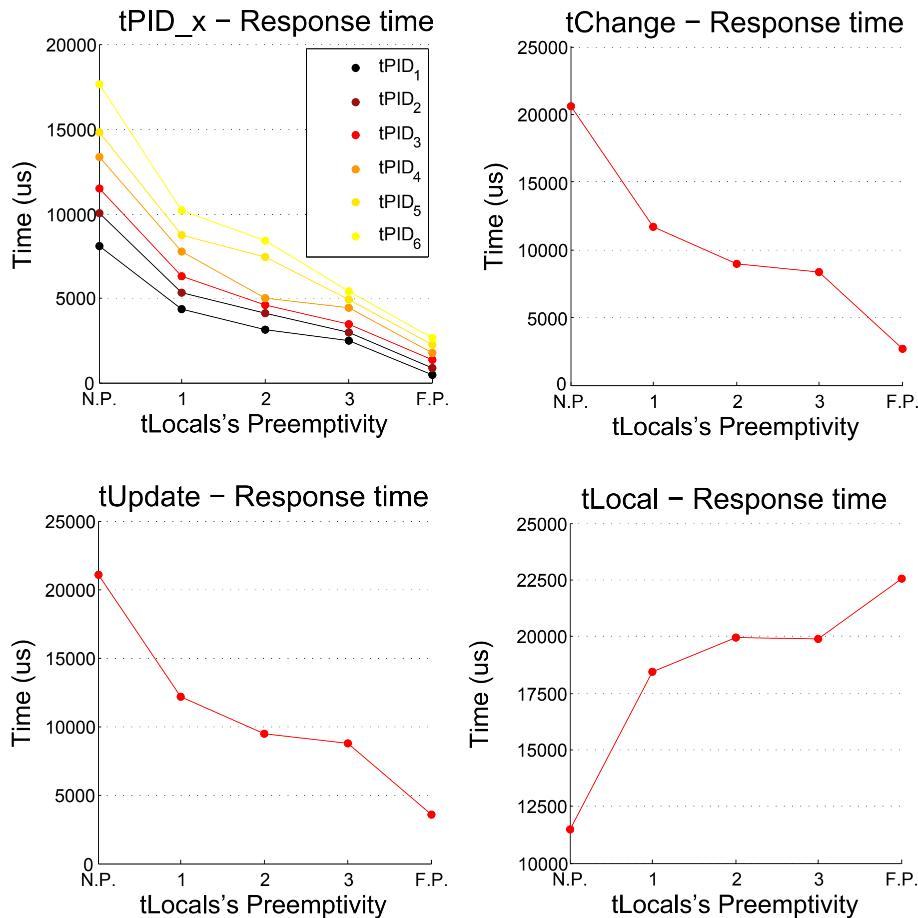



Figure 6.16: Exp. 1, big tasks - Theoretical response times

### 6.1.3 Preliminary discussion

The purpose of limiting preemption is to improve the response time of long, low priority tasks. However it is observed that tLocal has the best, or at least an equally good response time (providing a feasible system) for the fully-preemptive case, compared to the use of preemption points. A possible cause for this difference is that the operations responsible for avoiding preemption (in this case priority manipulation) generate an overhead which is more harmful than the interference of higher priority tasks. The execution and response times of the higher priority tasks have a by far better behaviour in the fully-preemptive case, comparing to the other methods. As such, using the traditional VxWorks scheduler without any extra routines in the tasks is the best option, especially when having a small impact of preemptions (small tasks).

## 6.2 Experiment 2 - Disabling the scheduler

Since using priority manipulation turned out to be a not very efficient solution, another approach was explored, namely to disable the scheduler in order to avoid preemption.

### 6.2.1 Short tasks

Once again, this implementation of the FPP algorithm is tested with original sized and expanded tasks. This section focuses on the short tasks.

#### Execution times

In this experiment the PID tasks execute for an average of 35$\mu$s for the non- and limited-preemptive cases and 12$\mu$s for the fully-preemptive case. Maximum execution times lie between 55$\mu$s and 150$\mu$, except for the fully-preemptive case, where it decreases to 25$\mu$s to 55$\mu$s. (Figure 6.17)

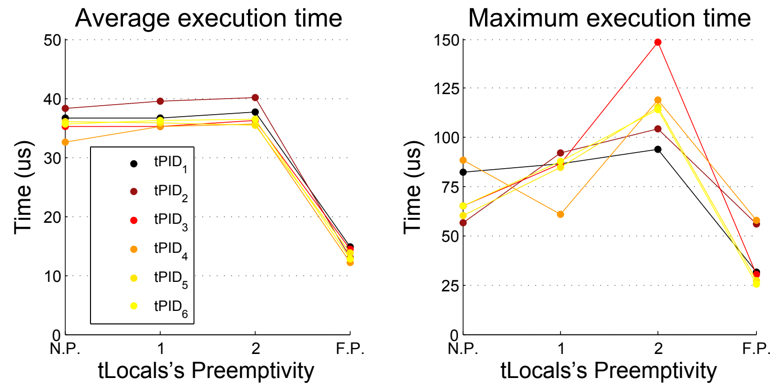



Figure 6.17: Exp. 2, short tasks - Execution time of the PID tasks

Figure 6.18 presents the execution times of tChange and tUpdate. Both tasks execute for an average of 40$\mu$s and maximum of 70$\mu$s for most situations. In the fuly-preemptive case tUpdate's execution time falls to 15$\mu$s to 20$\mu$s.

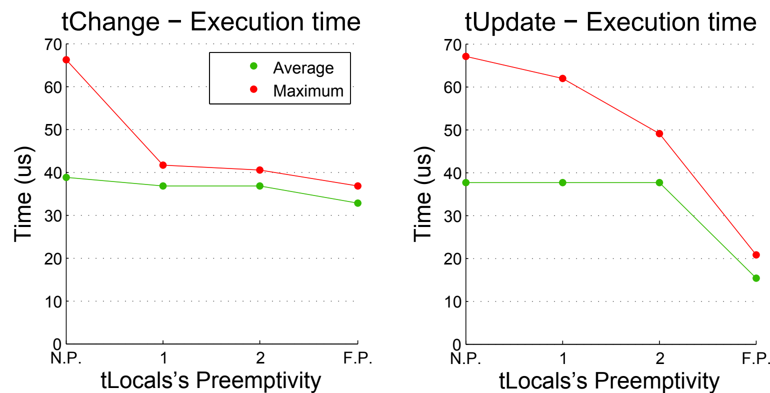



Figure 6.18: Exp. 2, short tasks - Execution time of tChange and tUpdate

Figure 6.19 illustrates the execution time behaviour of tLocal. The maximum values lie around 7260$\mu$s and grow to 7460$\mu$s in the fully-preemptive case. The cause of this increment is likely to lie on the fact that, without any preemption limiting, high priority background tasks interrupt tLocal. Since their execution times are not excluded from the measurements, tLocal's execution time includes the interrupting tasks. On the other hand, in the remaining cases the scheduler is disabled and only tLocal's processing is considered in the measurements.

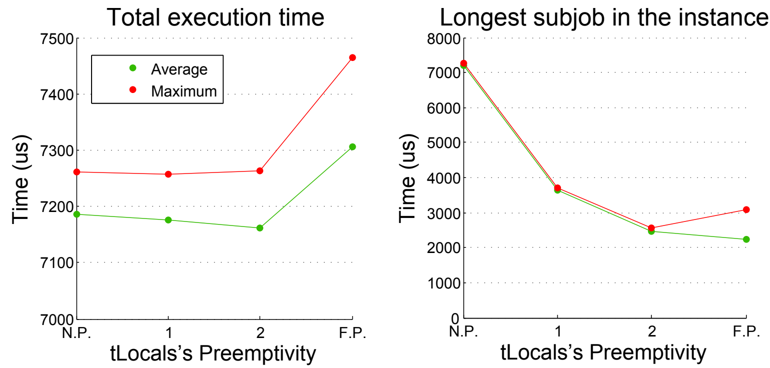



Figure 6.19: Exp. 2, short tasks - Execution times of tLocal (total value and longest subjob)

### Results

In this experiment only tPID_1 and tPID_2 miss deadlines for the non-preemptive case. Although for 1PP they present a very small number of misses, the system is only feasible for 2PPs. (Figure 6.20)

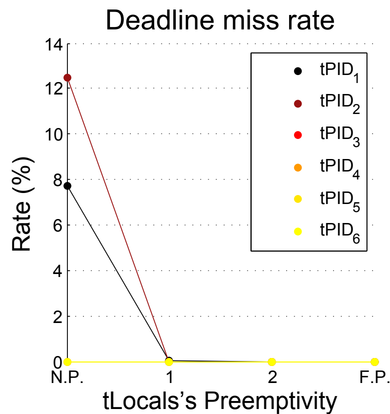



Figure 6.20: Exp. 2, short tasks - Deadline miss rate of the PID tasks

Figure 6.21 illustrates an analysis of the response time of the PIDs. The behaviour is similar as in experiment 1 for short tasks. However, tPID_3 and tPID_5 stand out as their average and maximum response times do not follow the same pattern as the other PIDs. This means that these tasks were never requested at the beginning of a tLocal instance in the non-preemptive case. The best worst-case response times are obtained for the fully-preemptive case and do not exceed 1000$\mu$s for any PID.

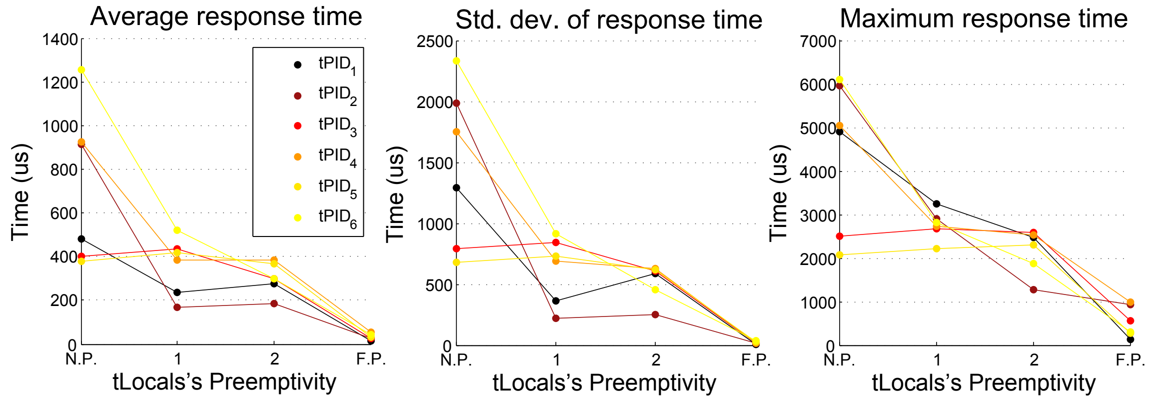



Figure 6.21: Exp. 2, short tasks - Response time of the PID tasks

tChange and tUpdate have similar curves for the response times, as shown in Figure 6.22. Both tasks were likely affected by the non-preemptive sections of tLocal, since their worst-case response times take values similar to the execution time of tLocal's subjobs.

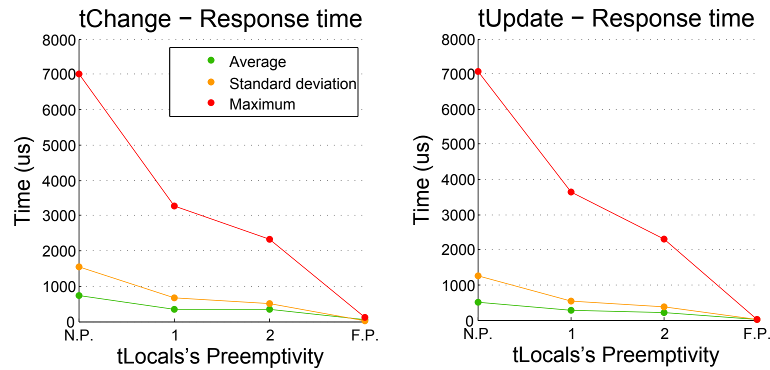



Figure 6.22: Exp. 2, short tasks - Response time of tChange and tUpdate

Figure 6.23 illustrates the response time behaviour and preemptions sufffered by tLocal. The response time increases as PPs are placed in its code, reaching a maximum of 7750$\mu$s for 2PPs. For the fully-preemptive case, where it suffers an average of 5.5 preemptions per instance, the worst-case response time falls to 7700$\mu$s.

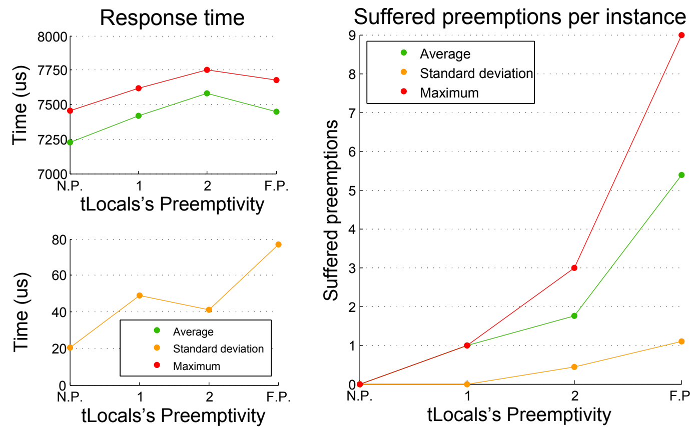



Figure 6.23: Exp. 2, short tasks - Response time and preemptions of tLocal

### Comparison with theoretical response time

The theoretically expected PID response times are mainly pessimistic comparing to the practical maximum response time, except for the fully-preemptive case. The same is observed for tChange and tUpdate. As for tLocal, the theory foresees shorter response times in all cases. A possible cause for that difference is that the estimations are obtained considering average execution times of the tasks. There might be cases, though, where tLocal suffers an interference of high priority tasks with longer execution times. The theoretical estimation of the tasks' response times are shown in Figure 6.24.

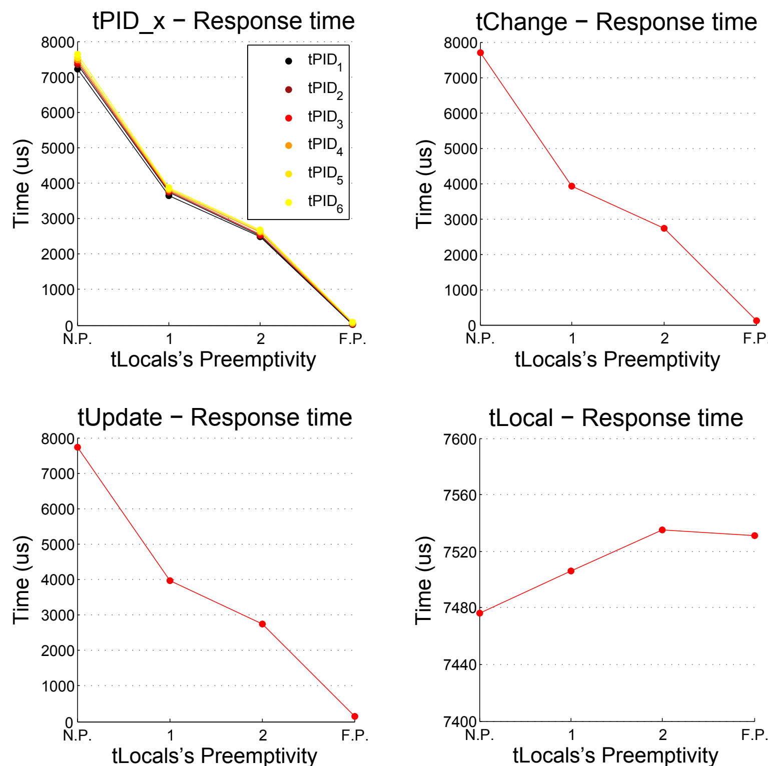



Figure 6.24: Exp. 2, short tasks - Theoretical response times

### 6.2.2 Long tasks

Once again the PIDs and tUpdate are inflated whith loops in order to cause a more significant interference against tLocal.

#### Execution times

As for the inflated PIDs, their execution times take average values between 300$\mu$s and 325$\mu$s for the non- and limited-preemptive cases and between 275$\mu$s and 300$\mu$s for the fully-preemptive case. Maximum values reach almost 450$\mu$s. (Figure 6.25)

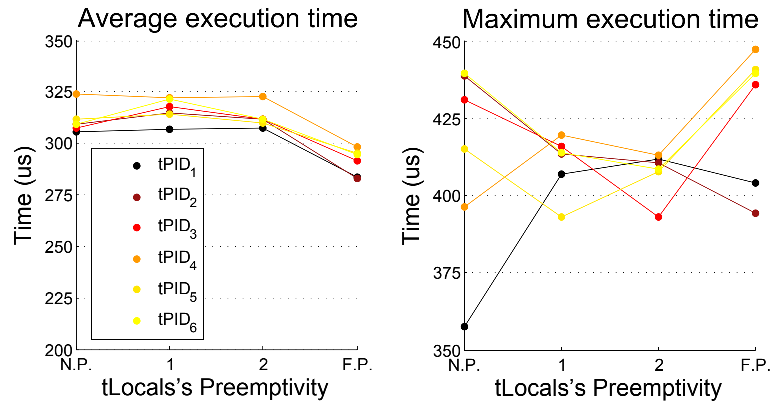



Figure 6.25: Exp. 2, long tasks - Execution time of the PID tasks

Figure 6.26 presents the execution times of tChange and tUpdate. tChange's worst-case execution time lies at 35$\mu$s, except for the fully-preemptive case, where it grows to 55$\mu$s. On the other hand tUpdate executes for a constant maximum of 400$\mu$s.

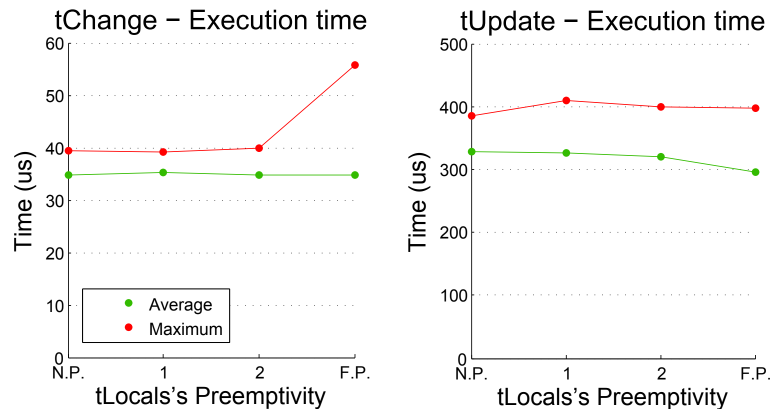



Figure 6.26: Exp. 2, long tasks - Execution time of tChange and tUpdate

tLocal has a maximum execution time around 6400$\mu$s for the non- and limited-preemptive cases and 6900$\mu$s for the fully-preemptive case. tLocal's execution time is represented in Figure 6.27.

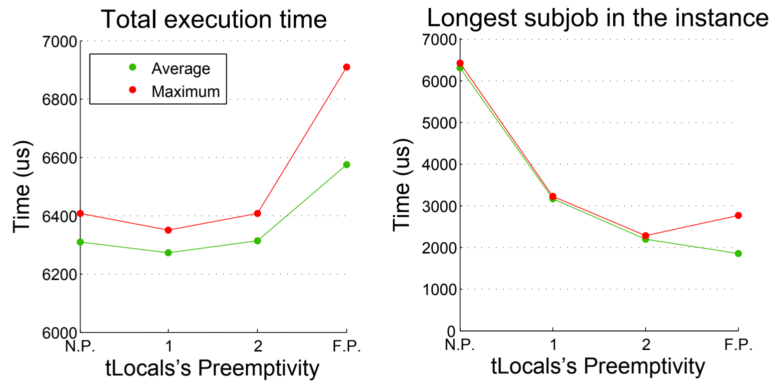



Figure 6.27: Exp. 2, long tasks - Execution times of tLocal (total value and longest subjob)

### Results

The PIDs reach zero percent deadline misses at 2PPs. Except for tPID_1 and tPID_2, they miss a very small number of deadlines along this experiment. The PIDs' deadline miss rate is illustrated in Figure 6.28.

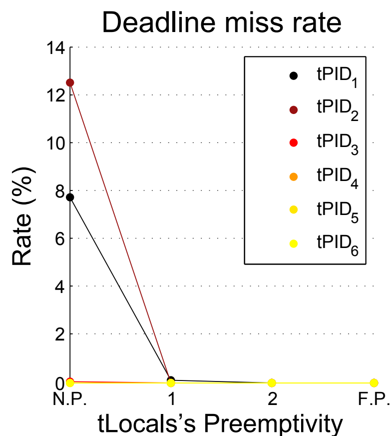



Figure 6.28: Exp. 2, long tasks - Deadline miss rate of the PID tasks

Figure 6.29 shows the average, standard deviation and maximum values of the PIDs' response times. As observed in previous experiments, the response time average, standard deviation and maximum decreases as tLocal's preemptivity level grows. The maximum response time lies, for most PIDs, between 6000$\mu$s and 8000$\mu$s for the non-preemptive case and falls to a range between 300$\mu$s and 2000$\mu$s when preemption is enabled.

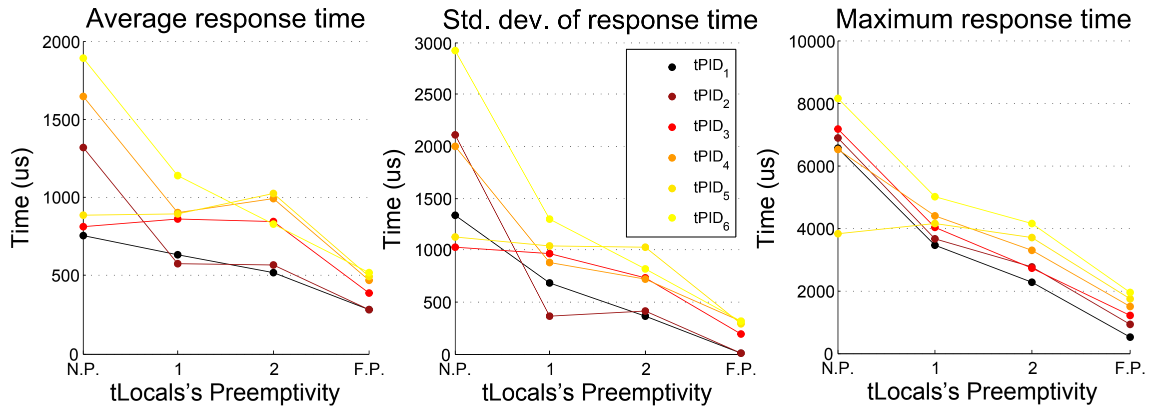



Figure 6.29: Exp. 2, long tasks - Response time of the PID tasks

Like in the short task case, in this experiment tChange and tUpdate present similar response time curves, with both maximums starting at 8000$\mu$s (non-preemptive case) and ending around 1000$\mu$s (fully-preemptive case). Figure 6.30 illustrates both tasks' behaviour.

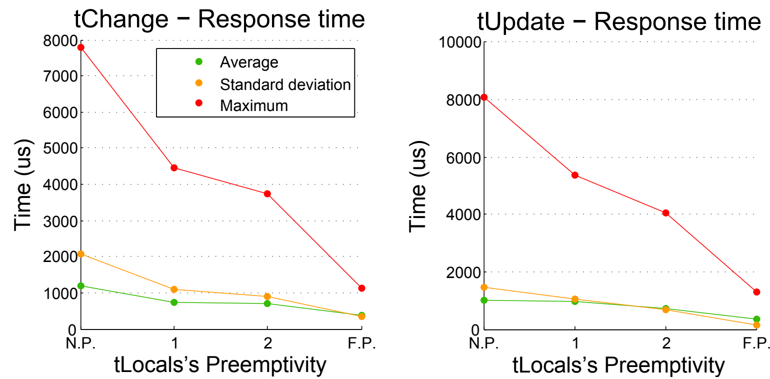



Figure 6.30: Exp. 2, long tasks - Response time of tChange and tUpdate

Figure 6.31 illustrates the response time and suffered preemptions of tLocal. The maximum response time grows with the preemptivity level of tLocal, as expected, varying in a range between 7700$\mu$s (non-preemptive case) and 12500$\mu$s (fully-preemptive case). In the former situation each instance suffers an average of 7.5 preemptions.

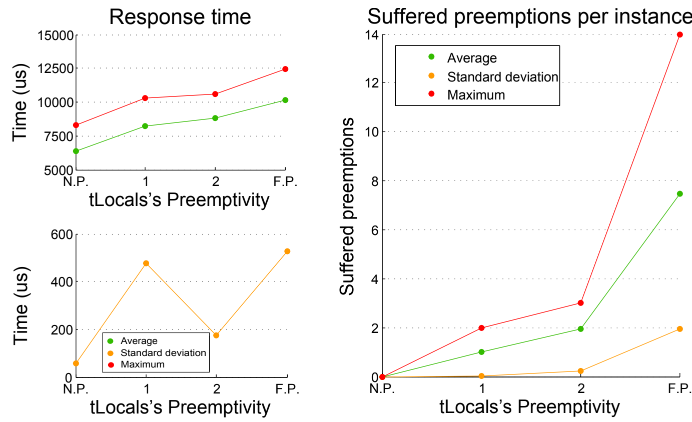



Figure 6.31: Exp. 2, long tasks - Response time and preemptions of tLocal

### Comparison with theoretical response time

The theoretical estimations for the response time of the tasks are illustrated in Figure 6.32. The estimations for the PID tasks are relatively accordingly to the actual worst-case response times. As to tChange and tUpdate, the theoretical response times are overall longer than the ones observed in practice. tLocal's practical maximum response time is shorter than the theoretical estimations except for the fully-preemptive case, where a longer response time is observed.

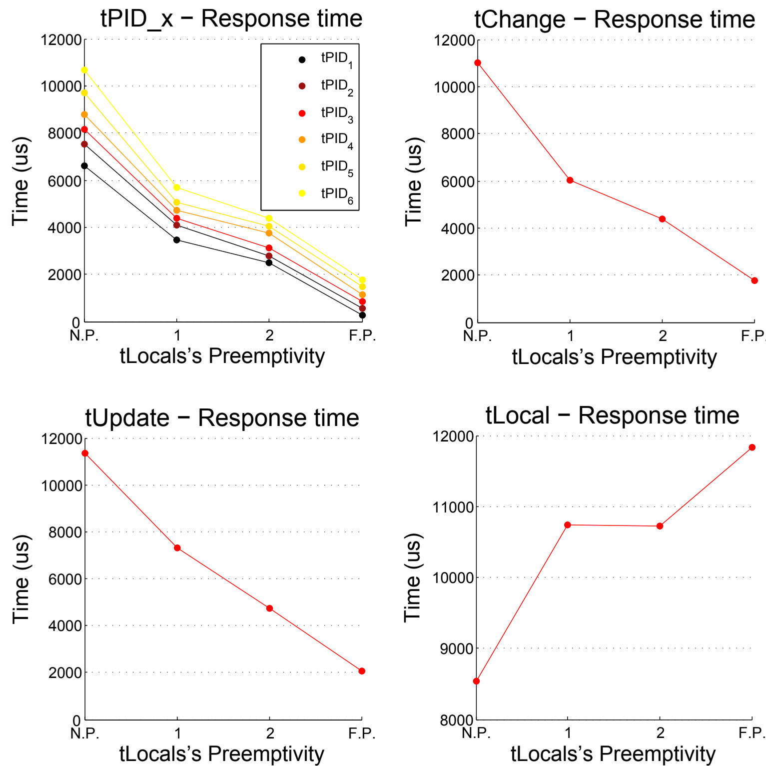



Figure 6.32: Exp. 2, big tasks - Theoretical response times

### 6.2.3 Preliminary discussion

Considering the short task experiment, like in experiment 1, for short tasks, the overhead of the operations responsible for limiting preemption is a drawback to all the tasks, even tLocal. The fully-preemptive scheduling policy is an asset for this task set, using this implementation.

For longer tasks the fixed preemption points provide a better response time to tLocal, comparing to the fully-preemptive scheduling policy, which in turn compensates providing the best response times to the higher priority tasks. However, the reader shall keep in mind that, due to the eventual desynchronizations of the tasks (section 4.2.4), it is possible that they execute less times than they are supposed to, as their interarrival time is always equal or greater than their period. Since this does not happen in the fully-preemptive case, the interference of the higher priority tasks is smaller in the non- and limited-preemptive scheduling policies.

## 6.3 Experiment 3 - Disabling the scheduler using timers

Another experiment using the scheduler disabling method was performed, this time using timers to wake up the application tasks. The reason behind experiment 3 is that, if the waking tasks use taskDelay() to synchronize, as in the previous experiments, they reset their phase every time their wake-up tasks execute after being pended on scheduler locking. This phenomena is explained in section 4.2.4.

In this experiment the application task set is slightly different than the one explained in chapter 5. Instead of having the sporadic task tChange calling tUpdate for a limited number of times, there are only sporadic requests of tChange, which does not call tUpdate and executes only once. In order to compensate the absence of tUpdate, tChange has been expanded with a finite loop, providing a greater impact on the system scheduling.

### 6.3.1 Short tasks

The following section describes the experiment for the task set without any artificial expanding of the PIDs.

#### Execution times

Figure 6.33 illustrates the average and maximum execution time of the PIDs. They execute for an average between 40$\mu$s and 50$\mu$s for the non- and limited-preemptive cases and around 15$\mu$s for the fully-preemptive situation. Maximum execution times lie between 300$\mu$s and 400$\mu$s and tend to be shorter with no preemption limiting.

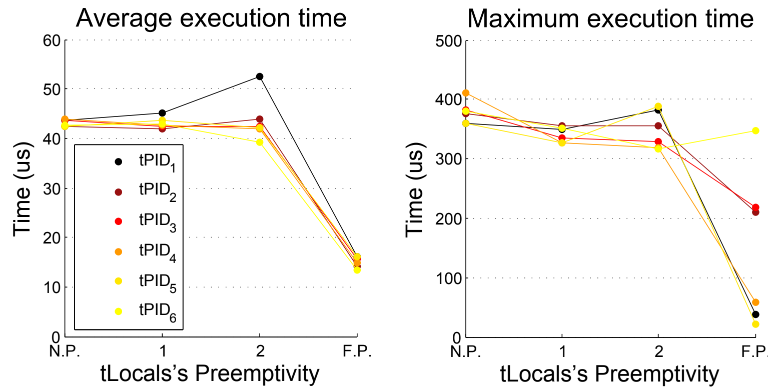



Figure 6.33: Exp. 3, short tasks - Execution time of the PID tasks

tChange executes for an average around 530$\mu$s and takes maximum execution times of 870$\mu$s in the non-preemptive case, decreasing to 610$\mu$s in the fully-preemptive situation. (Figure 6.34)

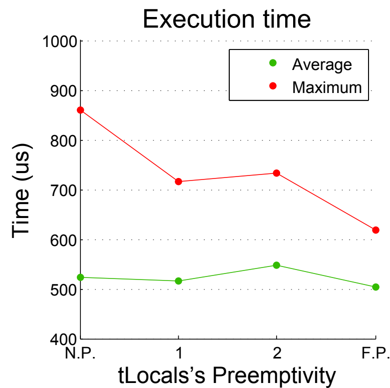



Figure 6.34: Exp. 3, short tasks - Execution time of tChange

tLocal has a maximum execution time of 8000$\mu$s, except for the fully-preemptive case, where it grows to 13000$\mu$s. This can be observed in Figure 6.35. The reason for that raise is explained in Note 2, at the end of section 4.2.4.

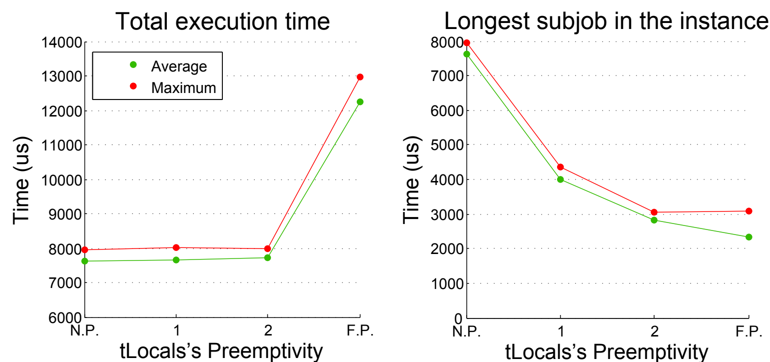



Figure 6.35: Exp. 3, short tasks - Execution times of tLocal (total value and longest subjob)

### Results

Figure 6.37 presents the deadline miss rate of the PIDs. tPID_1 is the task missing more deadlines and, for 2PPs, all PIDs meet their timing requirements.

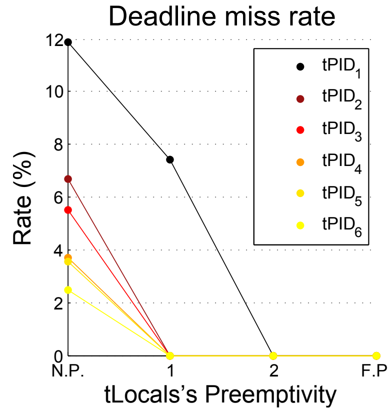



Figure 6.36: Exp. 3, short tasks - Deadline miss rate of the PID tasks

The response time of the PIDs has an average of 900$\mu$s in the non-preemptive case, decreases as preemption points are placed in tLocal and reaches values around 100$\mu$s for full preemption. Maximum values go from about 7500$\mu$s (non-preemptive case) to 1200$\mu$s and bellow (fully-preemptive case). The response time behaviour of the PIDs is shown in Figure 6.37.

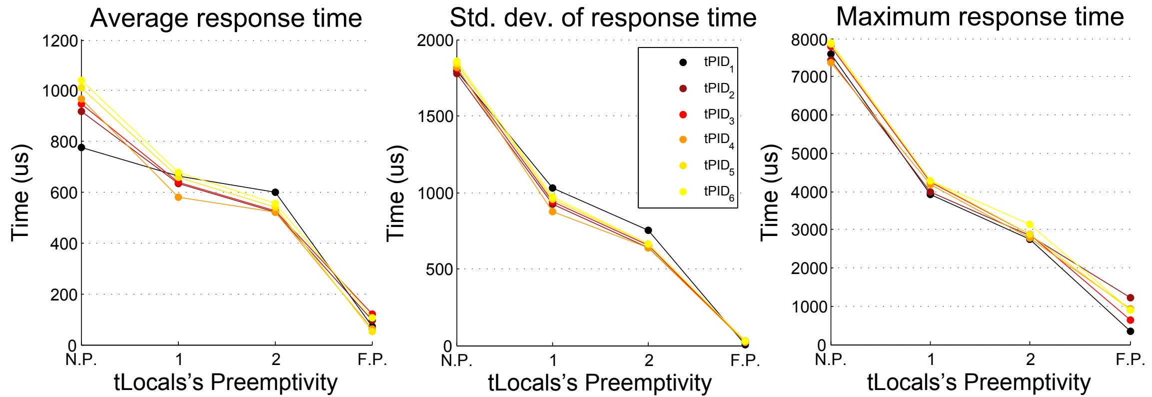



Figure 6.37: Exp. 3, short tasks - Response time of the PID tasks

Figure 6.38 illustrates tChange's response time. The task has relatively short worst-case response times (1600$\mu$s for the non-preemptive case), suggesting that it is not strongly affected by tLocal's non-preemptive chunks during this experiment, i.e., a worst-case situation has not occurred during the observation interval. In the fully-preemptive situation it has the shortest maximum response time, namely less than 700$\mu$s.

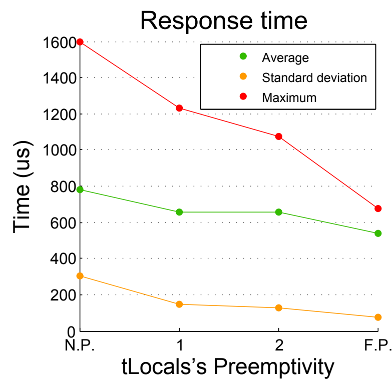



Figure 6.38: Exp. 3, short tasks - Response time of tChange

Figure 6.39 presents the response time behaviour of tLocal and the number of preemptions the task suffers per instance. The average and maximum response times increase as preemption points are accumulated in the code and, for the fully-preemptive case, it grows significantly, reaching a maximum of about 13700$\mu$s. In this case tLocal suffers an average of 12 preemptions per instance.

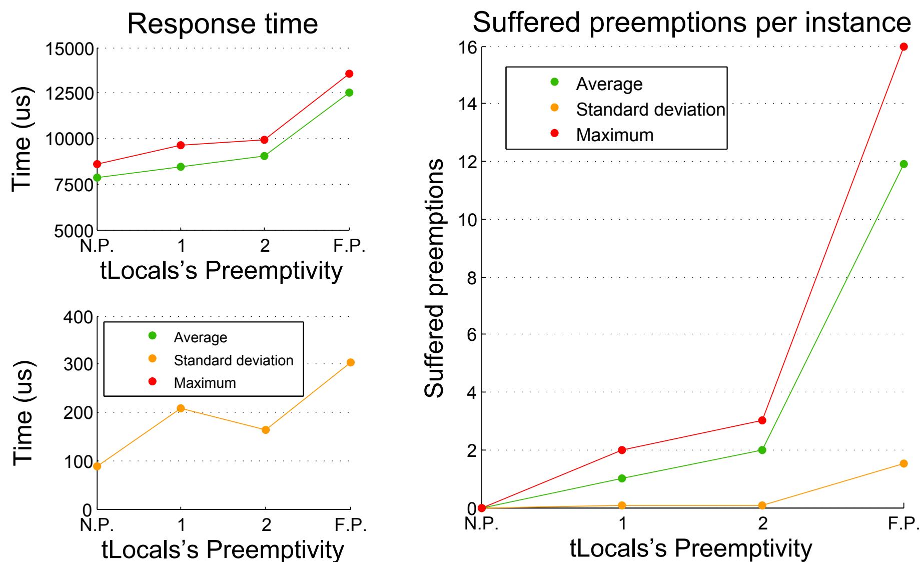



Figure 6.39: Exp. 3, short tasks - Response time and preemptions of tLocal

### Comparison with theoretical response time

The experimental results of the PIDs' maximum response times is very similar to the theoretical calculations, except in the fully-preemptive case, where the response time is significantly longer than estimated. The reason for that may lie on the fact that we used the average execution times of the tasks for the theoretical estimations, instead of the WCET, which may have affected the PIDs more often than expected. As to tChange the maximum response time is pessimistic for the non- and limited-preemptive cases and relatively accurate for the fully-preemptive situation. Finally tLocal's maximum response

time estimation is overall optimistic with respect to the observed results. (Figure 6.40)

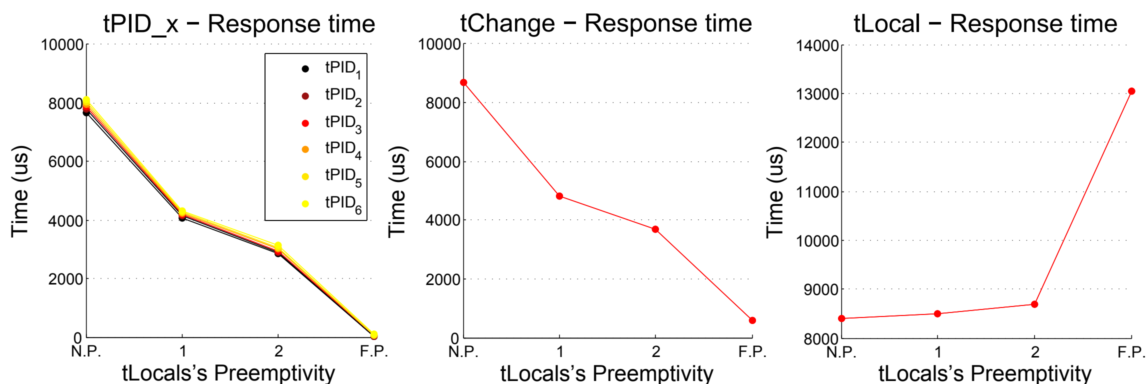



Figure 6.40: Exp. 3, short tasks - Theoretical response times

## 6.3.2 Long tasks

In order to keep up the routine of the previous experiments, an experiment using timers and featuring long tasks was performed.

### Execution times

Figure 6.41 shows the execution time behaviour of the PIDs. They execute for an average of 500$\mu$s, except tPID_1, which lasts slightly longer. In the fully-preemptive case the execution times decay to a range between 450$\mu$s and 500$\mu$s, approximately. Maximum values lie around 900$\mu$s, except in the fully-preemptive case, where they take dispersed values from 730$\mu$s to 980$\mu$s.

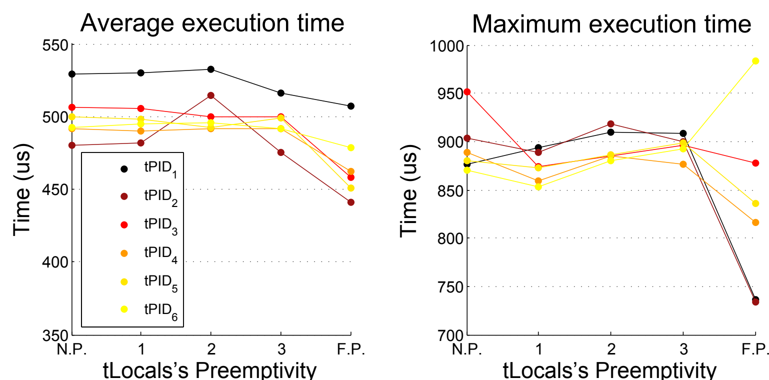



Figure 6.41: Exp. 3, long tasks - Execution time of the PID tasks

tChange has a uniform execution time in all situations, namely an average around 500$\mu$s and maximums between 700$\mu$s and 800$\mu$s. (Figure 6.42)

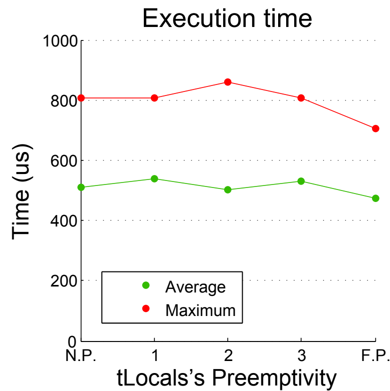



Figure 6.42: Exp. 3, long tasks - Execution time of tChange

As for the short-task experiment, using long tasks allows observing a similar course in the measurements of tLocal's execution times. It executes for around 7000$\mu$s in the non- and limited-preemptive cases and reaches more than 12000$\mu$s when preemption is fully enabled. Figure 6.43 illustrates this behaviour.

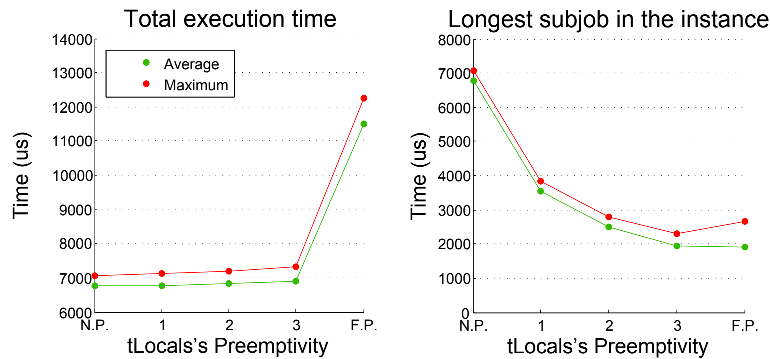



Figure 6.43: Exp. 3, long tasks - Execution times of tLocal (total value and longest subjob)

### Results

As shown in Figure 6.44, the PIDs provide similar curves for the deadline miss rates as in the short-task experiment. However, using long tasks the system only becomes feasible with 3PPs.

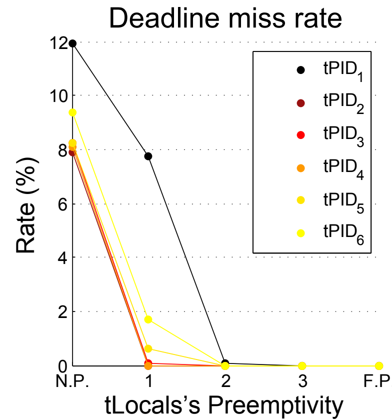



Figure 6.44: Exp. 3, long tasks - Deadline miss rate of the PID tasks

The response time of the PIDs has the expected behaviour. Average values lie between 1250$\mu$s and 2500$\mu$s in the non-preemptive case, fall as preemption points are added to tLocal and reach a range between 500$\mu$s and 1500$\mu$s in the fully-preemptive situation. Maximum values go from a range between 7000$\mu$s and 12000$\mu$s (non-preemptive case) to a range between 800$\mu$s and 4000$\mu$s (fully-preemptive case). This behaviour can be confirmed in Figure 6.45.

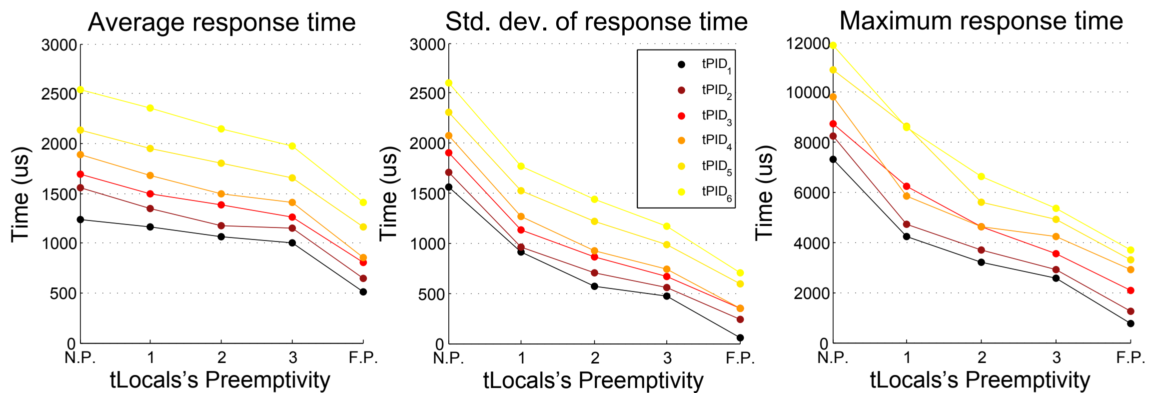



Figure 6.45: Exp. 3, long tasks - Response time of the PID tasks

The response time curves of tChange suggest that the task has been affected by tLocal's non-preemptive sections at some points of the experiment. Its average response time lies around 2000$\mu$s, while the maximum values vary between 5000$\mu$s and 5700$\mu$s. In the fully-preemptive situation the maximum response time falls to 3400$\mu$s. (Figure 6.46)

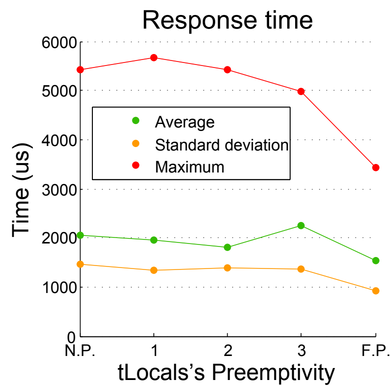



Figure 6.46: Exp. 3, long tasks - Response time of tChange

Figure 6.47 represents the response time and suffered preemptions of tLocal. Once again the task's response time gets longer as its own preemption level is increased and, like in the short-task experiment, takes a particularly big step from the case of 3PPs (maximum of 20000$\mu$s) to the fully-preemptive situation (maximum above 30000$\mu$s). Although the gap in the response times is greater than for the short-task case, the fact that the PIDs are longer causes tLocal's response time to be much larger than its execution time in all situations, and not only in the fully-preemptive case. In this situation tLocal suffers approximately 11.5 preemptions per instance, on average.

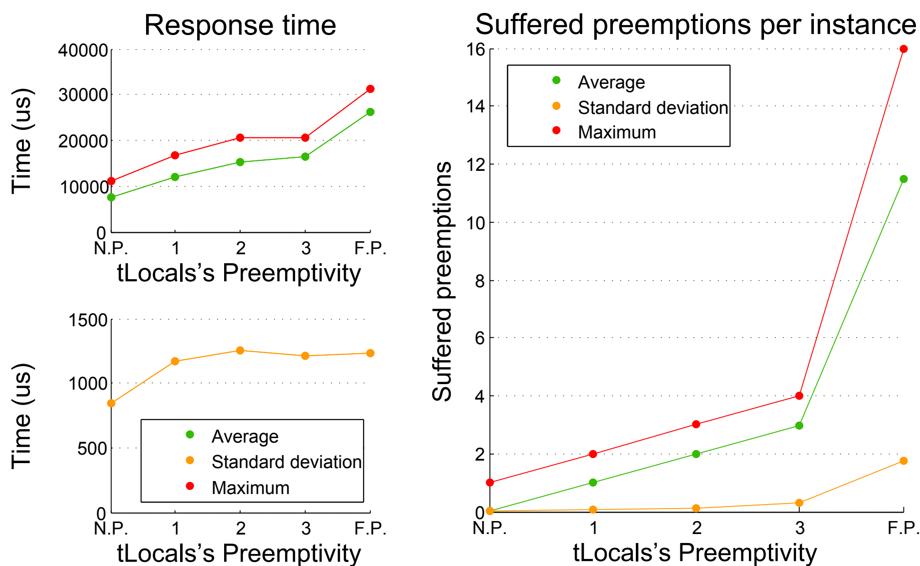



Figure 6.47: Exp. 3, long tasks - Response time and preemptions of tLocal

### Comparison with theoretical response time

Figure 6.48 shows the theoretical response times of the tasks. While for the non- and limited-preemptive cases the PIDs have manifested shorter response times in practice, for

the fully-preemptive situation the theoretical analysis is optimistic with respect to the actual results. As to tChange, the practical behaviour is, in every situation, better than the theoretical estimation. On the other hand, the theoretical response time analysis concerning tLocal presents shorter response times than the ones observed in the practical experiment.

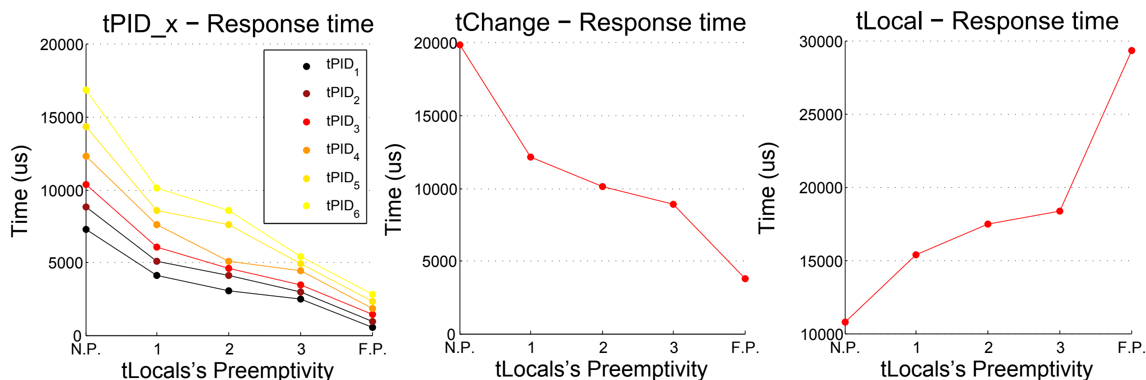



Figure 6.48: Exp. 3, long tasks - Theoretical response times

### 6.3.3 Preliminary conclusion

This implementation of waking tasks with timers avoids the problem of the application tasks increasing their periods due to scheduler disabling. However, due to the way it has been developed, i.e., the wake-up tasks are kept alive by yielding the CPU with a taskDelay(0) command in an infinite loop, it injects a constant overhead in the system. If the application tasks disable the scheduler in order to avoid preemption, this delaying action is not performed. However, in the fully-preemptive case it runs all the time, preempting the application tasks. This results in longer execution (and consequently response) times. As such, this implementation does not provide a good perspective for comparing the use of the FPP algorithm to the traditional fully-preemptive scheduling policy.

## 6.4 General discussion of the results

When focusing on the response time of longer low priority tasks, in this case tLocal, the Fixed Preemption Points algorithm may have a positive influence when we have a large impact of preemptions (long tasks), especially when preemption is inhibited by disabling the scheduler. In this case, neither experiment 2 nor experiment 3 provide completely fair results. In experiment 2 the total number of instances of the tasks is potentially smaller in the non- and limited-preemptive situations than in the fully-preemptive case. This is caused by the phase resetting of the tasks, explained in section 4.2.4. As to experiment 3, the execution time of tLocal is much longer for the fully-preemptive case than for the others. These differences muddle conclusions about the benefits of the FPP algorithm. The suggested method (FPP) does, indeed, present a better response time of the longer tasks. However, if the conditions were equal for all strategies, would the results indicate the same?

Experiment 1 provides a fairer comparison between the different scheduling techniques, since the number of task instances and the execution times are almost equal for all strategies. Because of this fairness, or perhaps because the priority changing routines inject a more significant overhead in the tasks' execution times, the response time of the longer, lower priority tasks does not present any improvements with preemption points relatively to fully-preemptive scheduling. Possibly, for even larger task sets (formed by larger tasks with longer periods), this FPP implementation can have a positive impact in the scheduling.

At the beginning of this document, the related work on limited preemption is discussed (section 1.2). Many of the authors affirm that, in preemptive systems, the WCETs of the tasks are harder to predict, since they suffer an undetermined number of preemptions. Thus, the execution times of the tasks embrace a potentially large number of context switch delays. However, in this project it is observed that both average and maximum values of the tasks' execution times are smaller for fully-preemptive scheduling for all experiments, especially when concerning the PIDs, tChange and tUpdate. This means that VxWorks' preemptive scheduling system is optimized and that the existing context switch delays have a negligible impact when compared to the insertion of extra operations in the tasks' code to inhibit preemption.

Appendix A shows, for each task and for all performed experiments, the average and maximum response times observed in practice, as well as the response times obtained theoretically based on the average and maximum execution times of the tasks.

# 7. Conclusion

This project allowed exploring a few real-time techniques, particularly the Fixed Preemption Points algorithm, which is a method that aims at limiting the preemption of preemptive systems. The purpose of limited preemption scheduling is to provide the longer, low priority tasks a better response time while exploring the slack of higher priority tasks to improve overall schedulability.

Concerning obtained practical skills, a lot was learned about the VxWorks 6.9 RTOS and the Workbench 3.3 software, both developed by Wind River systems. Still, these tools provide a world of useful features and options and many of them were not deepened as deserved along this project.

Facing the performed experiences and their results, many improvements in the implementations can be done and are left for future work. The following two paragraphs discuss some possible approaches.

The excessive overhead of the waking tasks that use timers (timerOffsetPeriodSpawn() and timerOffsetSporadicSpawn()) spoils the execution times of the tasks for fully- preemptive scheduling. In fact, the strategy to keep the waking tasks eternally alive would be approached differently now. For example, instead of having a taskDelay(0) command inside an infinite loop, which is constantly being executed, a larger delaying value can be used. As such the waking task waits a longer time interval between consecutive taskDelay() commands, introducing a less frequent processing in the system. Another idea is for the waking task to suspend itself before ending. If no other task resumes it, than the waking task exists forever in SUSPENDED state. The conditions for the timers' active existence must be studied a priori in order to know if these implementations will work.

We also conclude that the operations that inhibit preemption introduce a significant overhead. These delays are more harmful to the tasks' execution times than the overhead due to a higher number of context switches that may occur in a fully-preemptive system. Therefore we may consider that the approached implementations are far from being optimal. VxWorks' kernel has, by default, a fully-preemptive scheduling. Any attempt to change it is forcing the kernel to behave in a different way than the one it is optimized for. It is though possible to change the kernel in order to support a different type of scheduling, customized by the programmer. This approach is interesting to explore, as it would probably provide the best results, without the overhead of auxiliary functions.

# Bibliography

SENA - EESC - USP. URL http://www3.eesc.usp.br/sena/url/pt/index.php.

Luis Almeida. Escalonamento usando prioridades fixas. Technical report, Faculdade de Engenharia da Universidade do Porto, Departamento de Engenharia Electrotécnica e de Computadores, 2011.

S. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, 2005.

M. Bertogna and S. Baruah. Limited preemption edf scheduling of sporadic task systems. *Industrial Informatics, IEEE Transactions on*, 2010.

M. Bertogna, G. Buttazzo, M. Marinoni, Gang Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, 2010.

M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *2011 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 217–227, July 2011.

Buttazzo, G.C. and Bertogna, M. and Gang Yao. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, Feb 2013.

Yang-Min Seo Sang-Lyul Min Rhan Ha Seongsoo Hong Chang Yun Park Minsuk Lee Chong-Sang Kim Chang-Gun Lee, Joosun Hahn. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *Computers, IEEE Transactions on*, 47 (6):700–713, Jun 1998.

G. Bertogna M. Gang Yao, Buttazzo. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 351–360, Aug 2009.

Sheayun Lee, SangLyul Min, ChongSang Kim, Chang-Gun Lee, and Minsuk Lee. Cache-conscious limited preemptive scheduling. *Real-Time Systems*, 17(2-3):257–282, 1999.

Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, 2007.

Motorola. *MPC8260 PowerQUICC II User's Manual.* Motorola, 1999.

H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 212–224, Dec 2006.

Harini Ramaprasad and Frank Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08, pages 58–67, 2008.

Taisy Silva Weber. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. Technical report, Universidade Federal do Rio Grande do Sul (UFRGS), 2002.

WindRiver. *VxWorks Application API Reference, 6.9.* WindRiver, 500 Wind River Way, Alameda, CA 94501-1153, U.S.A.

WindRiver. *VxWorks Application Programmer's Guide, 6.9.* WindRiver, 500 Wind River Way, Alameda, CA 94501-1153, U.S.A., 4 edition, February 2012a.

WindRiver. *WindRiver System Viewer User's Guide, 3.3.* Wind River, 500 Wind River Way, Alameda, CA 94501-1153, U.S.A., 5 edition, November 2012b.

WindRiver. *Wind River Workbench User Guide 3.3.* WindRiver, 500 Wind River Way, Alameda, CA 94501-1153, U.S.A., 6 edition, February 2013a.

WindRiver. *Wind River VxWorks Simulator User's Guide, 6.9.* WindRiver, 500 Wind River Way, Alameda, CA 94501-1153, U.S.A., 3 edition, May 2013b.

Gang Yao, G. Buttazzo, and M. Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 71–80, Aug 2010.

# **Appendix A -** Summary of the response times (RT) of all tasks

All values are expressed in microseconds.

Legend:

(1) – FPP implementation based on priority manipulation. Wake-up tasks use a delay function.
(2) – FPP implementation based on scheduler disabling. Wake-up tasks use a delay function.
(3) – FPP implementation based on scheduler disabling. Wake-up tasks use periodic timers.

S – **Short** PIDs (low impact of preemptions).
L – **Long** PIDs (high impact of preemptions).

Av. – **Average** response time observed in the experiment.
Max. – **Maximum** response time observed in the experiment.

Av.Ex. – Theoretical worst-case response time obtained from the **average execution times** of the tasks.
Max.Ex. – Theoretical worst-case response time obtained from the **maximum execution times** of the tasks.

n.c – **Not converge**: The theoretical calculus of the response time does not converge to a value.

tPID_1

| | | **Non-preemptive scheduling** | | | | **1 preemption point** | | | | **2 preemption points** | | | | **3 preemption points** | | | | **Fully-preemptive scheduling** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| **(1)** | **S** | 690 | 8410 | 8739 | 8931 | 549 | 4188 | 4431 | 4684 | 234 | 3051 | 3012 | 3204 | 237 | 2291 | 2290 | 2515 | 15 | 250 | 15 | 29 |
| | **L** | 1022 | 7738 | 8103 | 8355 | 939 | 4330 | 4362 | 4611 | 788 | 3112 | 3096 | 3350 | 792 | 2414 | 2470 | 2942 | 436 | 655 | 436 | 634 |
| **(2)** | **S** | 479 | 4914 | 7222 | 73434 | 234 | 3244 | 3662 | 3801 | 279 | 2472 | 2492 | 2641 | x | | x | | 15 | 123 | 15 | 31 |
| | **L** | 757 | 6584 | 6613 | 6764 | 634 | 3466 | 3458 | 3635 | 515 | 2261 | 2493 | 2676 | x | | x | | 284 | 498 | 284 | 404 |
| **(3)** | **S** | 777 | 7578 | 7655 | 8319 | 665 | 3904 | 4059 | 4701 | 601 | 2747 | 2868 | 3440 | x | | x | | 72 | 336 | 16 | 37 |
| | **L** | 1237 | 7310 | 7291 | 7944 | 1165 | 4248 | 4073 | 4727 | 1069 | 3218 | 3029 | 3706 | 1005 | 2584 | 2462 | 3224 | 508 | 736 | 507 | 736 |

tPID_2

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| (1) | S | 855 | 8425 | 8865 | 9304 | 500 | 4355 | 4516 | 4928 | 368 | 3058 | 3098 | 3496 | 294 | 2291 | 2332 | 2677 | 24 | 260 | 29 | 86 |
| | L | 1255 | 8251 | 10038 | 10742 | 1183 | 4790 | 5329 | 5812 | 868 | 3330 | 4075 | 4583 | 999 | 2923 | 2954 | 3563 | 599 | 1251 | 875 | 1251 |
| (2) | S | 912 | 5944 | 7334 | 7566 | 170 | 2897 | 3738 | 3979 | 183 | 1260 | 2532 | 2746 | x | x | x | x | 23 | 16 | 28 | 88 |
| | L | 1315 | 6899 | 7533 | 7917 | 579 | 3681 | 4079 | 4456 | 570 | 2776 | 2805 | 3086 | x | x | x | x | 283 | 938 | 566 | 798 |
| (3) | S | 918 | 7444 | 7784 | 9777 | 633 | 3968 | 4146 | 5406 | 523 | 2840 | 2912 | 4176 | x | x | x | x | 93 | 1233 | 30 | 248 |
| | L | 1557 | 8240 | 8829 | 11478 | 1342 | 4704 | 5085 | 6509 | 1171 | 3706 | 4076 | 5534 | 1151 | 2894 | 2938 | 5033 | 643 | 1272 | 948 | 1470 |

tPID_3

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| (1) | S | 905 | 8486 | 8948 | 9493 | 514 | 4505 | 4556 | 5060 | 369 | 3102 | 3139 | 3625 | 297 | 2323 | 2372 | 2859 | 24 | 256 | 44 | 122 |
| | L | 1523 | 9709 | 11487 | 12548 | 1299 | 5742 | 6300 | 7604 | 1064 | 4185 | 4558 | 5252 | 1071 | 3431 | 3439 | 4824 | 595 | 1499 | 1315 | 1859 |
| (2) | S | 403 | 2488 | 7408 | 7688 | 434 | 2682 | 3773 | 4066 | 300 | 2579 | 2568 | 2894 | x | x | x | x | 23 | 565 | 42 | 118 |
| | L | 811 | 7206 | 8150 | 8787 | 863 | 4036 | 4397 | 4872 | 845 | 2721 | 3116 | 3891 | x | x | x | x | 391 | 1204 | 858 | 1234 |
| (3) | S | 950 | 7797 | 7870 | 10913 | 642 | 4230 | 4188 | 6096 | 528 | 2870 | 2954 | 4505 | x | x | x | x | 122 | 634 | 46 | 466 |
| | L | 1691 | 8763 | 10346 | 15113 | 1489 | 6237 | 6073 | 9166 | 1385 | 4632 | 4576 | 8248 | 1259 | 3568 | 3437 | 6828 | 803 | 2088 | 1407 | 2348 |

tPID_4

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| (1) | S | 1068 | 8571 | 9028 | 9728 | 667 | 4259 | 4595 | 5305 | 316 | 3215 | 3178 | 3737 | 318 | 2425 | 2411 | 2993 | 56 | 250 | 57 | 172 |
| | L | 2111 | 10647 | 13402 | 14901 | 1462 | 6403 | 7764 | 8806 | 1616 | 4840 | 5040 | 7739 | 1638 | 4397 | 4404 | 5445 | 656 | 1786 | 1756 | 2423 |
| (2) | S | 928 | 5041 | 7475 | 7842 | 383 | 2730 | 3808 | 4127 | 384 | 2530 | 2604 | 3013 | x | x | x | x | 54 | 984 | 54 | 176 |
| | L | 1646 | 6538 | 8781 | 9972 | 898 | 4403 | 4719 | 5291 | 991 | 3291 | 3747 | 4304 | x | x | x | x | 472 | 1477 | 1157 | 1681 |
| (3) | S | 965 | 7352 | 7958 | 12088 | 582 | 4168 | 4231 | 7108 | 523 | 2775 | 2996 | 4823 | x | x | x | x | 61 | 912 | 61 | 524 |
| | L | 1889 | 9824 | 12331 | 24244 | 1681 | 5839 | 7598 | 14449 | 1499 | 4643 | 5067 | 11846 | 1409 | 4219 | 4445 | 9508 | 856 | 2930 | 1869 | 3900 |

tPID_5

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| (1) | S | 974 | 8617 | 9149 | 10001 | 533 | 4578 | 4636 | 5442 | 389 | 3231 | 3219 | 3931 | 310 | 2429 | 2451 | 3151 | 30 | 209 | 71 | 200 |
| | L | 2002 | 11544 | 14831 | 18476 | 1834 | 8167 | 8741 | 11202 | 1499 | 5468 | 7470 | 8935 | 1417 | 4843 | 4887 | 8571 | 798 | 2413 | 2197 | 3672 |
| (2) | S | 377 | 2079 | 7544 | 7991 | 416 | 2219 | 3844 | 4212 | 368 | 2295 | 2639 | 3223 | x | x | x | x | 29 | 271 | 68 | 203 |
| | L | 888 | 3851 | 9722 | 11222 | 893 | 4164 | 5033 | 7340 | 1023 | 3708 | 4057 | 4712 | x | x | x | x | 493 | 1743 | 1452 | 2122 |
| (3) | S | 1010 | 7850 | 8044 | 13630 | 659 | 4285 | 4275 | 7762 | 543 | 2890 | 3038 | 5212 | x | x | x | x | 51 | 900 | 77 | 547 |
| | L | 2134 | 10893 | 14343 | 42187 | 1948 | 8633 | 8587 | 24106 | 1805 | 5608 | 7599 | 20805 | 1651 | 4909 | 4944 | 17364 | 1163 | 3322 | 2320 | 4736 |

tPID_6

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| **(1)** | **S** | 997 | 8604 | 9231 | 10357 | 531 | 4634 | 4676 | 5550 | 415 | 3265 | 3260 | 4039 | 334 | 2511 | 2492 | 3453 | 20 | 165 | 85 | 245 |
| | **L** | 2336 | 12530 | 17712 | 23888 | 2181 | 9628 | 10191 | 14777 | 1818 | 6609 | 8445 | 11425 | 1765 | 5405 | 5371 | 10433 | 800 | 3231 | 2637 | 4295 |
| **(2)** | **S** | 1255 | 6104 | 7616 | 8116 | 518 | 2801 | 3880 | 4300 | 297 | 1869 | 2676 | 3337 | x | x | x | x | 42 | 294 | 81 | 228 |
| | **L** | 1889 | 8160 | 10653 | 13262 | 1134 | 5038 | 5669 | 8147 | 830 | 4145 | 4368 | 5121 | x | x | x | x | 519 | 1938 | 1747 | 2562 |
| **(3)** | **S** | 1040 | 7871 | 8129 | 14729 | 680 | 4286 | 4318 | 8440 | 556 | 3137 | 3130 | 6212 | x | x | x | x | 105 | 897 | 90 | 894 |
| | **L** | 2535 | 11874 | 16845 | **n.c.** | 2351 | 8614 | 10109 | **n.c.** | 2150 | 6636 | 8587 | **n.c.** | 1967 | 5385 | 5435 | **n.c.** | 1410 | 3715 | 2799 | 14454 |

tLocal

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| **(1)** | **S** | 8719 | 8982 | 9014 | 9709 | 8954 | 9279 | 9066 | 10126 | 9150 | 9486 | 9299 | 10831 | 9174 | 9471 | 9277 | 11122 | 8832 | 9117 | 8891 | 9457 |
| | **L** | 8506 | 11630 | 11488 | 12559 | 12803 | 17780 | 18418 | 27655 | 15452 | 19644 | 19929 | 37233 | 16130 | 21232 | 19883 | 39797 | 16775 | 21141 | 22579 | 47596 |
| **(2)** | **S** | 7227 | 7456 | 7476 | 7813 | 7415 | 7617 | 7506 | 7947 | 7582 | 7753 | 7535 | 8396 | x | x | x | x | 7445 | 7679 | 7531 | 8010 |
| | **L** | 6350 | 8278 | 8537 | 9311 | 8202 | 10281 | 10732 | 12130 | 8825 | 10565 | 10723 | 13763 | x | x | x | x | 10116 | 12415 | 11833 | 17487 |
| **(3)** | **S** | 7822 | 8617 | 8392 | 11449 | 8479 | 9649 | 8486 | 13166 | 9061 | 9941 | 8686 | 14010 | x | x | x | x | 12500 | 13580 | 13045 | 16794 |
| | **L** | 7594 | 11197 | 10803 | **n.c.** | 12126 | 16841 | 15348 | **n.c.** | 15364 | 20564 | 17491 | **n.c.** | 16522 | 20478 | 18376 | **n.c.** | 26283 | 31123 | 29411 | 349793 |

tChange

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| **(1)** | **S** | 673 | 8035 | 9303 | 10610 | 541 | 4250 | 4706 | 5719 | 526 | 2770 | 3292 | 4095 | 471 | 2472 | 2523 | 3506 | 35 | 98 | 109 | 272 |
| | **L** | 2803 | 13104 | 20613 | 29922 | 2541 | 9207 | 11677 | 17855 | 1892 | 6227 | 8966 | 16482 | 2012 | 4828 | 8306 | 14905 | 647 | 2365 | 2667 | 4375 |
| **(2)** | **S** | 737 | 7021 | 7690 | 8247 | 341 | 3278 | 3917 | 4341 | 340 | 2332 | 2712 | 3377 | x | x | x | x | 39 | 106 | 114 | 265 |
| | **L** | 1177 | 7786 | 10997 | 14596 | 749 | 4455 | 6022 | 8600 | 689 | 3737 | 4403 | 5964 | x | x | x | x | 389 | 1141 | 1781 | 2617 |
| **(3)** | **S** | 781 | 1594 | 8695 | 17086 | 658 | 1234 | 4833 | 9508 | 655 | 1076 | 3677 | 8350 | x | x | x | x | 536 | 678 | 593 | 1512 |
| | **L** | 2058 | 5431 | 19869 | **n.c.** | 1943 | 5678 | 12130 | **n.c.** | 1801 | 5411 | 10120 | **n.c.** | 2240 | 4971 | 8940 | **n.c.** | 1547 | 3425 | 3781 | 47822 |

tUpdate

| | | Non-preemptive scheduling | | | | 1 preemption point | | | | 2 preemption points | | | | 3 preemption points | | | | Fully-preemptive scheduling | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | | Practical RT | | Theoretical RT | |
| | | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. | Av. | Max. | Av.Ex. | Max.Ex. |
| **(1)** | **S** | 116 | 449 | 9343 | 10673 | 121 | 359 | 4746 | 5781 | 86 | 439 | 3333 | 4165 | 80 | 320 | 2563 | 3614 | 17 | 21 | 126 | 292 |
| | **L** | 1871 | 6875 | 21091 | 30498 | 2080 | 7402 | 12162 | 18431 | 2037 | 6422 | 9452 | 17059 | 1711 | 4938 | 8790 | 15497 | 644 | 2893 | 3557 | 8705 |
| **(2)** | **S** | 501 | 7071 | 7728 | 8314 | 274 | 3641 | 3954 | 4403 | 207 | 2288 | 2750 | 3426 | x | x | x | x | 15 | 21 | 130 | 285 |
| | **L** | 1005 | 8066 | 11325 | 14982 | 958 | 5378 | 7290 | 9009 | 731 | 4047 | 4723 | 6363 | x | x | x | x | 338 | 1290 | 2077 | 3419 |