From Theory to Implementation of Embedded Control Applications
- A Case Study

Florian Fize

Supervisor : Amir Aminifar & Rouhollah Mahfouzi
Examiner : Petru Ion Eles
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page [http://www.ep.liu.se/](http://www.ep.liu.se/).

©Florian Fize
Abstract

Control applications are used in almost all scientific domains and are subject to timing constraints. Moreover, different applications can run on the same platform which leads to even more complex timing behaviors. However, some of the timing issues are not always considered in the implementation of such applications, and this can make the system fail.

In this thesis, the timing issues are considered, i.e., the problem of non-constant delay in the control of an inverted pendulum with a real-time kernel running on an ATmega328p micro-controller. The study shows that control performance is affected by this problem.

In addition, the thesis, reports the adaptation of an existing real-time kernel based on an EDF (Earliest Deadline First) scheduling policy, to the architecture of the ATmega328p.

Moreover, the new approach of a server-based kernel is implemented in this thesis, still on the same Atmel micro-controller.

Les applications d’asservissement sont utilisées dans presque tous les domaines scientifiques et subissent des contraintes de temps fortes. De plus, différentes applications peuvent exécuter sur le même support, ce qui mène à des comportements temporels encore plus complexes. Cependant, certains des problèmes temporels ne sont pas toujours pris en compte lors de l’implémentation de telles applications : ceci peut faire échouer le système.

Dans ce projet, les problèmes temporels sont considérés, i.e., le problème du décalage temporel non constant dans le contrôle d’un pendule inversé avec un kernel temps-réel s’exécutant sur microcontrôleur ATmega328p. L’étude montre que la performance de l’asservissement est affectée par ce problème.

En sus, le rapport présente l’adaptation vers l’architecture d’un ATmega328p, d’un kernel temps-réel préexistant, basé sur une police d’ordonnancement EDF (Earliest Deadline First).

Ajouté à cela, la récente approche d’un kernel basé sur un système de serveurs est implantée dans ce projet, toujours sur le même microcontrôleur Atmel.
Acknowledgments

First I would like to acknowledge my supervisors for this thesis, Amir Aminifar and Rouhollah Mahfouzi. Thank you for helping me and advising me all along the thesis. I would also like to thank my examiner Petru Eles for being patient for all the administrative issues.

Likewise, I acknowledge all the people in the staff of Linköping University who helped me to make possible this year of study and thesis in Sweden.

Finally, I would like to thank my home university, Grenoble Institute of Technology in France, for giving me the opportunity to participate to this double degree with the University of Linköping.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vii
List of Tables 1
Glossary 2
Acronyms 3

1 Introduction 4
1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.2 Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Research questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.4 Delimitation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.5 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.6 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Theory 7
2.1 Real-Time Systems and Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Control System Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Control Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3 Method 13
3.1 Study and Modification of a Real-time Kernel . . . . . . . . . . . . . . . . . . . . 13
3.2 Real-Time Servers for Control Applications . . . . . . . . . . . . . . . . . . . . 18
3.3 Worst-Case Execution Time: Static Timing Analysis . . . . . . . . . . . . . . . . 20
3.4 Inverted Pendulum Control and Performance Analysis . . . . . . . . . . . . . . 23

4 Results 26
4.1 Modification and Adaptation of the TRTKernel to a New Architecture . . . . . . 26
4.2 Implementation of The Server-Based Real-Time Kernel . . . . . . . . . . . . . . 27
4.3 Timing Analysis of the Control Application . . . . . . . . . . . . . . . . . . . . . 30
4.4 Simulation of the Control Application and its Performance . . . . . . . . . . . . 30

5 Discussion 34
5.1 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.3 The work in a wider context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

Acronyms 3
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Open-loop plant of the modeled system [michigan]</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Inverted pendulum set-up schematic [michigan]</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of periodic control application with constant delay (top) and non-</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>constant delay (bottom)</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Memomoy Layout of the ATmega328p</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>SRAM layout details</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Timer/Counter 1 working details</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of the Two-level Scheduling principle</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of scheduling with the (Q, P) pair</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>The Two-level EDF Scheduling</td>
<td>20</td>
</tr>
<tr>
<td>3.7</td>
<td>Execution Time Distribution</td>
<td>21</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of an Assertion File</td>
<td>23</td>
</tr>
<tr>
<td>3.9</td>
<td>Inverted Pendulum Model Specifications on Matlab</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Structure of the Server-based Kernel</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Structure of the server in the Server-based Kernel</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Structure of the local scheduler used in the servers of the Server-based Kernel</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Structure of a task in the Server-based Kernel</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>State-graph of how servers are managed</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Simulation 1, Angle Response Curves</td>
<td>32</td>
</tr>
<tr>
<td>4.7</td>
<td>Simulation 2, Angle Response Curves</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Simulation 3, Angle Response Curves</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>Simulation 4, Angle Response Curves</td>
<td>33</td>
</tr>
<tr>
<td>D.1</td>
<td>Simulation 1, Position Response Curves</td>
<td>48</td>
</tr>
<tr>
<td>D.2</td>
<td>Simulation 2, Position Response Curves</td>
<td>49</td>
</tr>
<tr>
<td>D.3</td>
<td>Simulation 3, Position Response Curves</td>
<td>49</td>
</tr>
<tr>
<td>D.4</td>
<td>Simulation 4, Position Response Curves</td>
<td>49</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Initial State of the Inverted Pendulum for the simulation</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Properties of the first set of tasks running on the Arduino. The control task is the task 2. Execution times are minimum, average and maximum observed, respectively.</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Properties of the second set of tasks running on the Arduino. The control task is the task 2. Execution times are minimum, average and maximum observed, respectively.</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Main Architectural Differences between the ATmega8L and ATmega328p devices.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>WCET Analysis Results with Bound-T Tool.</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Simulation 1, Parameters and Control Costs.</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Simulation 2, Parameters and Control Costs.</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Simulation 3, Parameters and Control Costs.</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Simulation 4, Parameters and Control Costs.</td>
<td>31</td>
</tr>
</tbody>
</table>
*composability* an architecture is said to be composable with respect to a specified property if the system integration will not invalidate this property, once the property has been established at the subsystem level. Examples of such properties are timeliness or testability. In a composable architecture, the system properties follow from the subsystem properties [23], [18].

*jitter* in our context, is the variation in delay between the invocation (or arrival) of a task, and its termination. [10], [12], [30].

*kernel* can be described as the part of an operating system interfacing between the applications and the system resources. For simplicity, kernel and operating system can be taken together. [7], [13], [15], [17].

*preemption* occurs in a preemptive system which allows a task to have its execution stopped in order to let another task with higher priority execute. [12].
Acronyms

API  Application Programming Interface. 29  34
EDF  Earliest Deadline First. 8  17  20  28  34  36
EEPROM  Electrically Erasable Programmable Read-Only Memory. 14
IDE  integrated Development Environment. 35
ISR  Interrupt Service Routine. 15
LQR  Linear-Quadratic Regulator (state-feedback controller). 10  23
MIPS  Million Instructions Per Second. 14
PC  Program Counter. 18
RISC  Reduced Instruction Set Computing. 13  14
RMS  Rate Monotonic Scheduling. 8
SP  Stack Pointer. 18
SPI  Serial Peripheral Interface. 14
SRAM  Static Random Access Memory. 14  17  26
SREG  Status Register. 16  18
TRTKernel  TinyRealTime Kernel. 9  17  19  20  24  34  37
VM  Virtual Machine. 18
WCET  Worst Case Execution Time. 13  21  30  34
1 Introduction

The thesis reports how timing can impact a control application performance, such as the control of an inverted pendulum, in an embedded real-time context during its implementation. This chapter describes with more details what is the aim of the thesis, as well as justifies the work and specifies the questions researched all along. Finally, it gives the delimitation and the background of the work, as well as a brief overview of the report.

1.1 Motivation

Control applications are meant to regulate the behavior of a system following certain rules in order to obtain the expected service. Moreover, these applications are always subject to timing constraints because time is always a parameter in a physical system. Nevertheless, this parameter is more critical for real-time systems and will, most of the time, shape the theory and the implementation of the control application.

However, some assumptions and approximations are used to develop the theories. This way, the impact of some timing phenomena is ignored or drastically simplified which can have severe consequences on the control performance.

1.2 Aim

The purpose of the thesis is to understand the timing issues that may occur during the implementation of control applications, such as the control of an inverted pendulum. Therefore, the idea is to explain the timing problems and to measure how much they impact the control performance.

An additional goal of the thesis is the implementation of a server-based system in order to have an isolation between different applications.

1.3 Research questions

To fulfill the objectives of the thesis, the report answers to the following questions:

1. What are the timing issues observed during the implementation of a control task?
2. How much do these issues impact the control performance of the control task?
3. How can a served-based system be implemented in a way to provide isolation between different applications?

Answering to these questions gives good insights to the problems pointed out at the beginning of the thesis.

1.4 Delimitation

The work is limited to the problem of an inverted pendulum, since the timing issues mentioned before are not related to the application itself and this is a classic case of study in control theory.

Moreover, the control application has to be executed by a real-time kernel, the TinyReal-time kernel [20], that runs on an Arduino Uno board [6]. Using a real-time kernel with given sources and a micro-controller with limited resources helps to have a better understanding of the execution environment and more stringent timing constraints.

1.5 Background

Here is the list of the tools used:

- Atmel Studio for Arduino code simulation [10]
- Arduino IDE for Arduino code [5]
- Bound-t tool for computing execution time [11]
- MATLAB for control simulation [26]

The following list contains references to related works:

- Computer-Controlled Systems, Åström and Wittenmark [7]
- “Integrated control and real-time scheduling”, Cervin [16]
- “Resource partitioning among real-time applications”, Lipari and Bini [24]
- “Control-scheduling codesign of real-time systems: The control server approach”, Cervin and Eker [17]
1.6 Overview

The thesis is separated into four chapters: theory, method, results and discussion. The next chapter presents what is considered as a real-time system in the thesis, gives the theoretical elements relevant to the control performance and describes the problem of the inverted pendulum.

The method chapter shows what work has been done to solve the problems pointed out earlier in the thesis. The study and the modification of the real-time system used are described, the timing analysis method is detailed, the server-based system is explained and, finally, the inverted pendulum model is given.

In chapter four, the final specification of the real-time system is presented, and the timing analysis results are summarized. Moreover, the implementation of the server-based system is reviewed and the simulation and test of the control application are explained.

Before the conclusion, results and methods are put into perspective and the work is put into a wider context.
This chapter aims to give theoretical basis that are useful for the understanding of this thesis. Indeed, the system is defined with a closer look at scheduling theory, and the control model as well as the inverted pendulum are described.

2.1 Real-Time Systems and Scheduling

The main concern of this thesis, besides control theory, is real-time systems. It seems then obvious to explain what can be described as a real-time system, particularly in the context of this thesis. An important feature of a real-time system is the task scheduling policy. Nevertheless, there are various policies to schedule tasks and the second part of this section is about the mechanism considered for this work.

Real-Time Systems

It is complicated to give a definition of real-time computing, but it is for sure that this kind of system is subject to real-time constraints. These constraints are strict\(^1\) and in the order of milliseconds or microseconds. Timing requirements for tasks are called deadlines, and the other parameters to consider are the execution time\(^2\) of the task and its period. Such characteristics have led to different theories to provide the most suitable schedule of tasks in order to meet the deadlines of every job. Moreover, executing tasks is using resources which need to be managed, thus, there are also hardware issues that must be handled. All of this is done in the kernel\(^3\) which manages the resources and schedules the tasks according to a chosen policy. For this reason, the way the kernel\(^4\) is implemented can have a huge impact on the performance of the system, it is explained in the section 3.1. Likewise, even if deadlines are met and the real-time paradigm is considered, control applications are a special case of real-time systems because control performance can be badly impacted by timing problems which are not taken into account by real-time computing theory. This is developed in the section 2.3.

To sum up, a real-time system is about timing requirements, but the thesis goes even more deeper into timing because of the consequences on control performance. The timing

---

1. Time constraints for a real-time system can be soft, firm or hard\(^5\) constraints. The latter is considered here.
2. The worst-case execution time, cf. section 3.3.
issues which can be noticed are linked to the way tasks are scheduled, so the next paragraphs explain how tasks can be scheduled in such a system.

Scheduling

In order to solve the problem of task scheduling several theories have been established [14]. Proposed techniques can be classified into two categories: fixed priority and dynamic priority algorithms. The idea is to give priorities to tasks so they will have an explicit order of execution which will ensure the correct functionality of the overall system. The probably most well-known and used algorithms are Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) scheduling [15]. They respectively belong to the first and second category mentioned before. These two algorithms are described in the following paragraphs, because they are used and/or mentioned all along the thesis.

Rate Monotonic Scheduling

The RM scheduling is a fixed priority algorithm, which means that the tasks have a priority assigned in advance that cannot be changed while running. Notwithstanding, the system is preemptive. Indeed, if a task with a higher priority is released when another one with lower priority is executing, the scheduler will interrupt the running task for the higher priority task. In addition, the algorithm is on-line, since the “next task to run” is chosen while running, based on predefined rules, even if the priorities don’t change at running time. In brief, this algorithm is based on those three principles:

- On-line
- Preemptive
- Priority-based with fixed priorities.

Each task is described with a quadruplet \((r, C, D, T)\), respectively the release time of a task, its worst case execution time (see section 3.3), its deadline, and its period. The priorities assignment follows the following rule: the shortest period gives the highest priority. The simplest version of the RMS analysis assumes that tasks don’t share resources, deadlines and periods are equal, and finally, that context switch times and other thread operations are ignored. Therefore, a sufficient schedulability test exists for this analysis [14]:

\[
\sum_{n=i}^{N} C_i / T_i < n(2^{1/n} - 1)
\]

with \(i\), the \(i^{th}\) task among a set of \(N\) tasks.

This compares the utilization of the tasks with an upper bound (which tends to \(\sim 0.69\) for \(n \to \infty\)) that guarantees the schedulability of the tasks with the RMS algorithm. No more details are given for this approach of the scheduling, because only EDF scheduling is used in this thesis, but it is still good to have insights about the RMS algorithm against EDF scheduling to understand better the latter. More details are provided in the Liu and Layland work [25].

Earliest Deadline First

As opposed to RMS, the Earliest Deadline First scheduling is based on dynamic priorities and its policy is the following: the task that is closest to its deadline runs first. The idea is to keep a priority queue for tasks and then whenever a scheduling event happens (a task is released, finished . . .), the task with the highest priority is set to run and the queue is updated considering the closest deadline among the set of tasks. The simplest EDF analysis assumes
that each task has a known period, a maximum computation time, a relative deadline less
than or equal to the period and the processes are independent. This way, EDF can schedule
the set of tasks so they all meet their deadlines. Similar to RMS, this scheduling algorithm is
also on-line and preemptive. In the case of periodic tasks, if their deadlines are equal to their
periods, the following necessary and sufficient schedulability condition applies [14]:

\[ \sum_{i=1}^{N} C_i / T_i \leq 1, \]  

(2.2)

with \( i \), the \( i^{th} \) task among a set of \( N \) tasks.

Equation 2.2 allows deciding on the schedulability of a set of tasks. It shows that the
algorithm can reach an upper bound of 100% of utilization, and the tasks are still schedulable.
First, it means that EDF scheduling is still usable at higher processor loading, but it assumes
also that context switching between tasks is negligible. A consequence of this is that when an
overload occurs, it is unpredictable to know which deadlines will be missed. So to sum up
the characteristics of the EDF scheduling:

- On-line
- Preemptive
- Priority-based with dynamic priorities.
- Closest deadline run first
- Deadlines are met even at higher utilization

This scheduling algorithm is implemented in the TinyRealTime Kernel (TRTKernel) which
is described in the section 3.1.

2.2 Control System Model

The application of the thesis is the control of an inverted pendulum. In order to establish this
control, the system needs to be modeled. An overview of the system model is given first, then
the controller model is described and finally the model is clarified in the case of an inverted
pendulum.

Plant Description

The plant is modeled by a continuous-time system of equations:

\[
\begin{align*}
\dot{x} &= Ax + Bu + v \\
y &= Cx + e,
\end{align*}
\]

with \( x \) the \textit{plant state}, \( u \) the \textit{control signal} and \( v \) the \textit{plant disturbance} as a continuous white-
noise process with zero mean and known co-variance matrix \( R_1 \). Also, \( y \) is the \textit{plant output}
which is periodically sampled and \( e \) is the \textit{measurement noise}, as a discrete-time Gaussian
white-noise with zero mean and co-variance \( R_2 \). \( A, B \) and \( C \) are the \textit{state}, \textit{input} and \textit{output}
matrix, respectively. These parameters are system dependent and are specified in the inverted
pendulum description part. To have another representation of the model, here is a schematic
of the plant:
Controller Description

For the controller synthesis, the system is simplified to the case of continuous-time plant and controller, ignoring delay, measurement noise and disturbance. The controller considered is a Linear-Quadratic state-feedback controller (LQR). The idea of LQR design is to find the gain matrix $K$ for a linear state feedback control law $u = -Kx$, by minimizing a quadratic cost function of the form:

$$\int_0^\infty x(t)^T Q x(t) + u(t)^T R u(t) \, dt,$$

where $Q$ and $R$ are weighting parameters that penalize certain states or control inputs.

Here, only a quick overview of the controller description without justifications is given, however details and proofs can be found in [1] and [7]. The model has been described from a general point of view. Below, the particular case of the inverted pendulum is discussed.

Inverted Pendulum Description

The case of the inverted pendulum is first a problem of mechanics. The figure 2.2 shows how the pendulum is set up.

Parameters of figure 2.2 are $M$ mass of the cart, $m$ mass of the pendulum, $b$ coefficient of friction for the cart, $l$ length of the pendulum, $I$ mass moment of inertia, $F$ force applied to the cart, $x$ position of the cart and $\theta$ pendulum angle.

The schematic gives all the parameters that are needed to apply the law of dynamics on the inverted pendulum. After computations and approximations, the two linearized equations of motion can be established:

$$\begin{align*}
(I + m l^2) \ddot{\phi} - m g l \dot{\phi} &= m l \ddot{x} \\
(M + m) \ddot{x} + b \ddot{x} - m l \ddot{\phi} &= u,
\end{align*}$$

(2.5)
2.3 Control Performance

with the same parameters that are described for Figure 2.2 plus \( \phi \) as a small angle deviation from equilibrium. The details of the analysis are found at [22].

It is worth to mention that the control law described before is based on this linearized system, which means that the controller is effective only when the pendulum is near the upright position.

With these equations, it is possible to represent the inverted pendulum in state-space form:

\[
\begin{bmatrix}
\dot{x} \\
\ddot{x} \\
\dot{\phi} \\
\ddot{\phi}
\end{bmatrix} =
\begin{bmatrix}
0 & -1 & 0 & 0 \\
0 & \frac{m^2g^2}{I(M+m)+Mml^2} & 0 & 1 \\
0 & 0 & \frac{mgl(M+m)}{I(M+m)+Mml^2} & 0 \\
0 & 0 & 0 & \frac{mgl(M+m)}{I(M+m)+Mml^2}
\end{bmatrix}
\begin{bmatrix}
x \\
\dot{x} \\
\phi \\
\dot{\phi}
\end{bmatrix} +
\begin{bmatrix}
0 \\
I+m^2l^2 \\
0 \\
0
\end{bmatrix} u
\]

\( y = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} u \)

(2.6)

So, matrices described in the plant overview are found as:

\[
A =
\begin{bmatrix}
0 & -1 & 0 & 0 \\
0 & \frac{m^2g^2}{I(M+m)+Mml^2} & 0 & 1 \\
0 & 0 & \frac{mgl(M+m)}{I(M+m)+Mml^2} & 0 \\
0 & 0 & 0 & \frac{mgl(M+m)}{I(M+m)+Mml^2}
\end{bmatrix},
B =
\begin{bmatrix}
0 \\
I+m^2l^2 \\
0 \\
0
\end{bmatrix},
C =
\begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

(2.7)

Now, with all these equations and parameters, it is possible to model all the system including the inverted pendulum itself as well as the LQR controller associated. Nevertheless, pendulum specifications have to be given in order to have a model corresponding to the plant analyzed in the thesis. These parameters are specified in Section 3.4.

2.3 Control Performance

In the thesis the performance of the control application is evaluated in order to understand the impact of timing issues. Here it is explained how the performance is quantified and also the main timing problem that can be observed is described.

Control cost

The performance of the control application is quantified by a control cost. The control law in Section 2.2 is given by minimizing the quadratic cost shown in equation 2.4.

Nevertheless, the system can be simplified and adapted to the discrete-time representation. In fact, in the method chapter it is explained that the inverted pendulum will be considered as a discrete-time system (see Section 3.4). Finally, the control cost is evaluated as follows:

\[
\sum_{k=0}^{N} x(k)^T Q x(k) + u(k)^T R u(k),
\]

(2.8)

with \( k \) the sample index, \( N \) the total number of samples and \( x, u, R \) and \( Q \) as defined before in this chapter.
Non-Constant Delay Problem

The main timing issue considered for the thesis is pointed out in this subsection.

The main idea behind a control application is to sample periodically the state of the plant, to compute the control value to apply following a predefined control law and then to apply this value to the plant. So the control application can be divided into three parts: sampling, computing and actuating. Figure 2.3 (upper part) exemplifies this scenario with \( k \) the sampling period, \( L \) the constant time between beginning and the end of the control task, \( u \) the control input value applied and \( t \) the time.

One problem is that during the time between sampling and actuating the state of the plant the previous control value is still applied. Therefore, if this time is too long, the system can completely fail. In fact, if the \( L \) value as considered earlier is found to be suitable for a correct control performance, a problem can remain with a delay caused by an event exterior to the control application itself. Actually, other tasks can run in the system and may delay the execution of the control part. Again, if this delay is known and constant it can be compensated in the controller synthesis [1]. Yet, the problem is really different if this delay is not constant and unpredictable. For example the control task can be delayed because another task executes before, but also, preemption can occur and add delay between sampling and actuating. This variation in delays is called jitter. Such a scenario is illustrated in figure 2.3 (lower part) where non constant delays \( a, a' \) and \( b \) are added, due to the execution of other tasks. With several tasks running on the same processor, delay can become non constant and yield into consequences for the control performance that cannot be compensated with the controller synthesis. This is in this case that control performance is evaluated for the purpose of the thesis.

More details on the non constant delay problem are available at [1].

Figure 2.3: Example of periodic control application with constant delay (top) and non-constant delay (bottom)
3 Method

In this chapter, first, it is explained how the real-time kernel is studied and modified. Then the WCET (Worst-Case Execution Time) parameter is described, as well as how to evaluate it. After that, the server-based real-time Kernel principle is detailed and, finally, it is presented how the inverted pendulum is used for the purpose of the thesis.

3.1 Study and Modification of a Real-time Kernel

The idea of the thesis is to run control applications on an embedded real-time device. The first thing to consider is to have a real-time kernel operating and managing the applications. The kernel used, is based on an already existing implementation, developed by Anton Cervin & Dan Henriksson [20]. This operating system[1] has been adapted to a fairly popular micro-controller product: the Arduino Uno [6]. In order to give a relevant description of the used and modified kernel, the micro-controller architecture is reviewed.

Atmel ATmega328p architecture overview

To understand the details of the implementation of the kernel, it is needed to give an overview of the hardware that runs this kernel. Firstly, the device is introduced by a description of its specifications and performances. Then, more details are given, such as the memory map and the main features considered, for a deeper understanding of the kernel implementation.

The micro-controller

All along the thesis, an Arduino Uno board is used for the experiments and analysis of a real-time system. Only the micro-controller is of interest, even if the other components of the board can be of use and can simplify access to some features, the serial communication for instance.

The board contains an Atmel ATmega328p 8-bit AVR [RISC] based micro-controller, which presents sufficient performance needed for this thesis. A complete description of the product and the details of its parameters, can be found on the Atmel website [8].

[1] Kernel and Operating System are taken together for simplification.
Here come the main specifications that are relevant to consider.

- 131 instructions
- 32 General 8bit-registers
- 32 KBytes Flash memory
- 20 MHz clock frequency
- 1 KBytes EEPROM
- 2 KBytes SRAM
- Two 8-bit Timer/Counters and one 16-bit Timer/Counter
- SPI interface

The micro-controller follows the philosophy of a RISC processor and, thus, offers a rather simple architecture and a large set of low complexity instructions, which provides an efficient execution unit. In fact, most of the instructions execute in one clock cycle and the performance can be evaluated at nearly 1 MIPS (Million Instructions Per Second) per MHz [8]. In order to, again, maximize the efficiency of the processor, a Harvard architecture, i.e. separating program and data memories (and their access), is used. The only limitation, that is discussed further in subsection 3.1, is about the memory and mainly the SRAM. The timer/counters are the core of the kernel, as it is described in the Timers and Interrupts part, in subsection 3.1. The SPI interface may be used for interacting with the pendulum and more details are found in subsection 3.4.

The list shown above, gives an overview of the system, which, despite its rather simple resources, fits totally to the purpose of the thesis. Moreover, a closer look to the memory layout is needed for a good understanding of the kernel.

### Memory Map

One important part of a kernel functionality is to manage the memory for the running applications in the system. It is even more critical for a real-time resources-limited system, as it is considered here. This layout from the Atmel documentation [8], helps to understand how the memory is organized in the device:

![Memory Layout of the ATmega328p](image)

In fact, the memory space that matters the most is the SRAM, which will be the main memory concern for the kernel. Figure 3.2 explains a bit more how this part of the memory is arranged:

In addition to a better view of the different allocated spaces of the SRAM, addresses of free memory available are shown, so this figure can also help to figure out what the addresses used further stand for.
Another important specification of the micro-controller that is a key feature for the kernel is the interrupts and timers available in the device.

Timers and Interrupts

In addition to managing the memory, the core functionality of the real-time kernel studied is based on the use of timers and their associated interrupt routines. Therefore, it is important to take the time to explain how this functionality works, before explaining why and how it is used for the Kernel.

In brief, an interrupt is a call to a subroutine that is triggered by a hardware event and, if enabled, will execute without any other conditions, even if another program is running. Actually, there is some priorities between the different interrupts, so one can be executed before another because of a higher priority; this is the only case that can delay the execution of an interrupt routine. To be more clear, the "interrupt" is the interruption of the normal flow of execution, and the "interrupt routine" is the code associated that is executed to handle the event that triggered the interrupt. This routine is also named ISR (Interrupt Service Routine) and the different interrupts are referenced as vector interrupts. In the Kernel, the interrupts used are generated by counting tasks in background, that use dedicated timers.

Several timers are available in the device, but only one is used for the first version of the Kernel. Not all the timers are exactly the same, slight differences exist in the way of configuring them, as well as their resolution and functionality. Nevertheless, describing only how the Timer/Counter 1 works will be sufficient to understand the purpose of it. In fact, the Timer/Counter 2 is also used for the second implementation of the Kernel, but details are provided in the related section 3.2.

The Timer/Counter 1 is a 16-bit counter which has different features, and is characterized by different registers. The timer can be used to generate interrupts when:

- The timer value matches a value in a specific register (two different registers available)
• An overflow of the timer occurs
• An event is captured (not used here, more information in Atmel’s datasheet \[9\])

To enable a specific interrupt, related registers have to be set, and flags are raised in associated registers when the interrupt occurs. There is also a status register \(\text{SREG}\) which allows globally enabling or disabling the interrupts.

The schematic below, gives details on how these registers are set up and how an interrupt can be generated, in the case of a compare/match with the \(\text{Timer/Counter 1}\):

![Schematic](image_url)

**Figure 3.3: Timer/Counter 1 working details**

As it can be seen in the schematic:

- \(\text{TIFRx}\)\(^2\) shows flag corresponding to the kind of interrupt that occurred
- \(\text{TIMSKx}\) is a mask to whether consider or ignore the flags
- \(\text{SREG}\) allows or not all the interrupts in the system
- \(\text{TCCRx}_A/\text{B}\)\(^3\) are used to configure the \(\text{Timer/Counter}\) and interrupt, e.g. change the clock rate, kind of interrupt to generate …
- \(\text{OCR}_{x_A/\text{B}}\) contain the value used for a compare match with the corresponding \(\text{Timer/Counter}\)

For the kernel, the \(\text{Timer/Counter}\) is configured in order to generate an interrupt when the \(\text{Timer/Counter}\) matches the value in the \(\text{Compare/Match}\) register \(A\). There is no reset of

\(^2\) \(x\) can be either 0, 1 or 2, corresponding to which timer it refers to.

\(^3\) \(A\) and \(B\) refer to the two \(\text{Compare/Match}\) registers available for each timer
the Timer/Counter or the Compare/Match register A when a match occurs, but an overflow resets the Timer/Counter.

This functionality fits perfectly with the EDF (Earliest Deadline First) scheduling policy that is used in the kernel. Furthermore, the kernel should use a minimum amount of SRAM memory, because of the small size provided in the ATmega328p. This is why the TinyRealTimeKernel (TRTKernel) is described in details in the next part, so it is possible to understand how the previous specifications are used to create a real-time EDF-based kernel.

TinyRealTimeKernel, an EDF-based real-time kernel

The kernel is a real-time kernel for Atmel AVR ATmega8L 8-bit micro-controller, which has been adapted for the Atmel AVR ATmega328p. Only few details are different between the two architectures which follow the same philosophy. Details on the ATmega328p’s architecture are described in the previous section 3.1, which can help to understand how the kernel is implemented and how it has been modified. Since the two Atmel devices are close, the way of working is similar and can be explained regardless of the specific architectures.

The first point is an overview of the kernel, and then a zoom on the internal workings gives a good understanding. Finally, the adaptation of the kernel to the new architecture is presented, where the main architectural differences are pointed out and modifications made are highlighted.

Overview of the kernel and its working routine

Details on the implementation of the Kernel are given in the technical report written by Anton Cervin & Dan Henriksson [20]. The Kernel is designed in order to follow an EDF scheduling policy, which provides a dynamic schedule based on the idea that the task with the earliest deadline gets the highest priority in the priority queue. The EDF scheduling is specified in the theory part 2.1. First, an overview of the kernel is given, next the interrupt routine, which is the core of the kernel, is detailed.

Overview of the TRTKernel

Although the kernel is EDF-based, it has the particularity to not have a queue for the tasks; instead each task has its associated state. Thus, the scheduler searches among the tasks to find which one is ready to execute and has the earliest deadline. Mainly, this implementation is done to reduce the memory requirement of the kernel. Schematic 3.2 shows how the SRAM is organized for the kernel.

The idea is that most of the specifications of a task are part of its own structure and each task has its own stack. Thereby, each task holds its own context, so the kernel only chooses the next task to run, and with the stack pointer known in the tasks structure, all the context of a task can be saved or restored. In addition, the kernel manages the tasks regarding timing events which generate clock interrupts. This is at this point that the Timer/Counter, available on both ATmega8L and ATmega328p and described in the subsection 3.1 is used.

The EDF scheduling is done by creating timing events, which will trigger an "interrupt" which is basically the kernel routine. Timing events are matches between the Timer/Counter 1 and the time value written in the OCR1A (Output Compare Register 1 A). Then, the interrupt handler executes the routine associated in order to schedule the next task to run. This routine is explained a bit later in the report.

The timer is a 16-bit register which allows to count to a value of $2^{16}$, that is extended with another 16-bit register. This latter is increased whenever the timer reaches its maximum value, which is called a new cycle for the timer. To extend the life time of the system, a prescaler is used, as it is shown in the figure 3.3 in order to divide the system clock. Thereby, the "time" for the timer is a fraction of the system clock, and is counted as a number of ticks.
However, using a pre-scaler reduces the resolution of the timer, there is obviously less ticks than clock cycles. This is not really a problem since most of the time values are way much higher than the clock period, so there is no loss of accuracy for the timer.

In brief, the kernel operates with timing events to schedule tasks and manage their contexts by accessing the internal SRAM. All these operations are done in the interrupt handler routine, which is what the part below is about.

**ISR: Interrupt Service Routine, the core of the TRTKernel**

As it is explained in the previous paragraph, the kernel internal workings mainly occur in the interrupt routine related to the match between the *Timer/Counter 1* and the Output Compare Register 1 A. (The code of this routine is given in the listing in appendix A.)

This routine has three important features, which schedule the tasks in the system:

- Save / restore the context of a task
- Choose the next task to run
- Set the timer for the next interrupt

For the first one, it can be done automatically by the compiler, since it takes care of how handling an interrupt call or the code can be explicitly written. In both cases, it consists in pushing into the stack the PC (Program Counter) value, the 32 working register values and the SREG for saving and restoring; the same thing is done by a pop operation of the same data from the stack. As said above, each task has its own stack, and the SP (Stack Pointer) is also part of the task structure. Thereby, the push for saving the context is done into the stack of the running task, based on its SP value. Then, the current SP value is saved in the previous running task structure and SP is changed to the stack pointer of the next task to run. In the end, data in the stack of the next task to run, are popped.

Choosing the next task to run is following the EDF-based scheduling idea, excluding the priority queue. The set of tasks in the system, is gone through in order to check the state of each task, to know if a task is ready to run or will be released soon, and among the ready tasks, the one with the earliest deadline is the next task to run. A task goes from a “waiting” state to a “ready” state when its release time is reached, according to the current timer value. When a new task is chosen, the kernel updates the index of the running task (to know which one is currently running) and switches context by changing the current SP value to the SP of the next task to run, as it is detailed in the above paragraph.

Since the main components of the TRTKernel have been described, it is possible to explain what has been changed, in order to adapt this kernel to the architecture of the Atmel ATmega328p, used during the experiments of the thesis. This is the purpose of section 4.1 in the results chapter.

### 3.2 Real-Time Servers for Control Applications

A modern approach to the implementation of control applications is the abstraction of server/VM (Virtual Machine) system [17]. In order to guarantee the **composability** of applications and the fulfillment of their timing requirements, each application can get a dedicated server that will use an assigned fraction of the processor which runs the system. Thus, using a two-level scheduler architecture one can implement such a system. This section explains more in depth the principle of this architecture, before providing details on how to implement it with a real-time kernel running on the ATmega328p micro-controller.
The Two-level Architecture

Nowadays, it is possible to execute different real-time applications in the same system, since computers are getting more and more powerful. It is of interest to ensure the composability of these applications and it can be done by implementing a system that runs these applications following a chosen scheduling policy. However, it is better if you can use already existing applications and add them to the new system. Nevertheless, these applications can have their own scheduling policy. Here comes the advantage of a two-level scheduler. First, there is the **global scheduler** which manages the different applications and a sub-level scheduler: the **local scheduler**. By this isolation, each application uses its own scheduling algorithm without any impact on either the other applications or the global system. Figure 3.4 illustrates this principle.

![Figure 3.4: Illustration of the Two-level Scheduling principle](image)

The idea for this architecture is to have the global scheduler, with its own scheduling mechanism, that decides which server will run. Furthermore, each application has a pre-defined budget and period denoted by the pair \((Q, P)\). This means that a server has \(Q\) units of time to execute every \(P\) units of time. For instance, figure 3.5 shows how applications can be scheduled with this mechanism.

![Figure 3.5: Example of scheduling with the \((Q, P)\) pair](image)

To sum up, with the server-based approach and the two-level scheduler architecture, it is possible to compose different applications with their own scheduling mechanism in the same system. Assigning a bandwidth to each application, in agreement with its timing requirements, it is also possible to guarantee the good working of the application without any interference with the others. Nonetheless, part of this thesis is to implement such a system. To do so, the work is based on an algorithm from the Lipari and Bini’s paper [24] and the **TRTKernel** described in the section 3.1. The next part of the thesis aims to explain how this implementation is done.
Server-based Real-Time Kernel

As said in the section above, the Lipari and Bini’s paper [24] provides an algorithm which describes how to manage resources among the different servers regarding their budget and period. Then, the idea is to implement this algorithm in the context of a real-time Kernel for the ATmega328p micro-controller. It is still possible to choose the global scheduling policy, because the Lipari and Bini’s paper [24] provides the rules for servers with an abstraction of a particular global scheduler. Furthermore, the paper said that EDF is totally suitable for the algorithm. So, it is possible to use and adapt the TRTKernel described in the section 3.1 for the server-based kernel.

For more details about the algorithm for managing the servers, refer to the chapter 4 and the appendix [P].

The first part is to schedule the different servers with an EDF scheduler based on the TRTKernel, and then each application might have its own local scheduling policy. In this thesis, only one control application is implemented and is scheduled by an EDF policy. Thereby, the local scheduling of the applications will first focus on an EDF mechanism, although the final idea is to have a total abstraction of the local scheduler.

To sum up, the server-based kernel is implemented on the roots of the TRTKernel, and so far, the kind of server to run with this kernel, is also inspired by the control application running on the TRTKernel. See the figure 3.6 for an illustration of this mechanism.

The result of the implementation of the server-based real-time kernel is described in the results chapter, in section 4.2.

3.3 Worst-Case Execution Time: Static Timing Analysis

In most systems, timing is important, since every task should perform within a given time to ensure the proper working of the system. In real-time systems, timing is the main concern and deadlines for the tasks are often strict, it means no violation on the deadline is allowed. In addition, there are several jobs running at the same time, so the system decides which one must run and when, in order to guarantee a good working of the overall system. This is why one of the main features of a real-time operating system is the scheduling of tasks. Yet, to build a relevant schedule (whatever algorithm) for the tasks, it is needed to know the time taken by a task to execute. For this reason a timing analysis is needed for the implementation of the control application in this thesis. In this section, firstly it is explained why the analysis is static and why the worst case is needed. After that, the different tools studied are briefly reviewed before describing the one which has been chosen and how the static analysis is done.
3.3. Worst-Case Execution Time: Static Timing Analysis

Static Analysis and choice of the tool

In order to provide a good scheduling of the tasks, their execution time have to be analyzed. This analysis can either be dynamic or static, so it is first explained why the analysis is static. Then, different tools for WCET (Worst Case Execution Time) static analysis are reviewed and it is explained why a tool has been chosen rather than another.

Static Analysis

To ensure that the schedule will always make the tasks meeting their deadlines, the execution time evaluated must be the longest, the worst case. There are different possibilities to analyze the timing for the execution of tasks; it can be dynamic (running simulation) or static analysis. For the first one, there are several problems. Firstly, the simulation can take hours, or even more for a complicated system, to be able to simulate all the running scenarios. The second one is that, simulation provides maybe an average or a most probable running time but it is not possible to know if this time is the worst case. This is what is illustrated in figure 3.7.

![Figure 3.7: Execution Time Distribution](image)

This is why the static analysis is preferred instead of simulation. The idea behind this kind of analysis is to determine how a program will execute, and, with specifications of the hardware on which the code will run, it is possible to know the time taken for the execution, without simulating this execution. In addition, the analysis is based mainly on the code of the program, thus, it is possible to determine the worst path that can be taken by the program, so the upper bound of the timing execution can be found.

Different tools exist in order to find the WCET of a task by a static analysis; they are briefly presented in the next part.

Static analysis and WCET estimation tools

Finding the WCET of a task is possible using static analysis tools which will almost automatically find the upper bound of the execution time of a task. Several tools are available, some are commercial, others are totally free or only free for academic purpose. The first feature needed for choosing a tool (in the case of this thesis), is that it has to be free, so here comes a first list of these tools:

- Bound-T Tool of Tidorum
- Research prototypes from Florida State University / North Carolina State University / Furman University
- OTAWA from IRIT, Toulouse
- Research prototypes from TU Vienna
3.3. Worst-Case Execution Time: Static Timing Analysis

- The Chronos research prototype from National University of Singapore
- The Heptane tool of IRISA Rennes
- SWEET (SWEdish Execution Time Tool)
- Research prototype from Chalmers University Of Technology
- SymTA/P Tool of TU Braunschweig, Germany
- aiT WCET Analyzers from AbsInt (commercial but with a 30-days free trial)

A detailed comparison of all these tools (and more) is given in the paper “The worst-case execution-time problem—overview of methods and survey of tools” [29].

Since the tool should take into account the architecture for the execution of the program, the ATmega 328p micro-controller needs to be supported by the tool. This eliminates most of the tools cited above. Nevertheless, the second one, OTAWA [27] and Chronos [18] could have been adapted for the hardware used, but it takes time to do so. The first tool doesn’t support the ATmega328p, but it supports other ATmega devices which have the exact same hardware architecture for the part it is needed to consider for the static timing analysis. Thus, Bound-T [12] is the tool that fits the most the static analysis which has to be done for the thesis.

To understand a bit more how the tool works, an overview of the static analysis principle for computing the WCET, as well as how the tool is used, are given in the next subsection.

Outlines of the WCET static timing analysis and use of Bound-T

The static timing analysis of a program is able to give an upper bound of the WCET of a task, with the help of tools, such as Bound-T. The aim of this subsection is to explain the main idea of the WCET static timing analysis, particularly in the case of the Bound-T tool.

Counter to a simulation, static timing analysis does not try to run the program in different scenarios which tend to be the worst possible; the idea is to analyze the machine code of a program to determine the worst path and to bound every part of the program. The notion of bound is important, because the analysis is not about finding the unique worst case possible, but to compute an upper bound which can be used to ensure that a task will not execute in more time than this limit [21]. Sometimes it can be obvious to find such a limit, because the path of the program is certain, so knowing the hardware specifications, it is only about counting a number of clock cycles. However, tasks can have an execution path which is data dependent. For example a loop can be executed an unknown number of times because it has a condition which is based on an input external of the device itself, e.g. values from a sensor. Moreover, there can be logical links between different loops or “if” statement, so it is not really obvious to find how the program is going to behave and it is even more difficult to find out the worst case.

To help finding the WCET of a task, several tools and especially Bound-T, use indications about the behavior of the code. These indications are written by the user in a file, and are called assertions. Figure 3.8 is an example of how an assertions file can look like. Mainly, this file is used to explicit the scenario of the execution, for instance: number of iterations of an unbounded loop, inputs load, execution count of a call... With such details the tool is able to create more easily the flow graph of the program.

In addition, the tool creates a model of computations based on the instructions of the hardware target and their effects. Then the data-flow can be analyzed. In particular, the Presburger Arithmetic is used to model the relations between data-cell values, even when the actual values are not statically known [12]. Also this analysis allows finding bounds for each loop, which represents a bottleneck of the timing analysis. This analysis added to the model of execution time of instructions gives all the elements needed for the final step: calculating the overall bounds.
3.4 Inverted Pendulum Control and Performance Analysis

In this section, it is explained how the control of the pendulum is realized and with which means performance is evaluated.

Control of the Inverted Pendulum

At first, the idea was to run on a computer the simulation of the control of the inverted pendulum and then the simulation results would be compared to the control performance of the same control application but running on the Arduino board controlling the inverted pendulum. However, it turned out that it was impossible to get the inverted pendulum within the duration of the thesis. So it was not possible to use a physical inverted pendulum. The next paragraph explains what is done instead.

As it is mentioned in the title of the thesis, there is a study between what occurs in theory against what happens after the implementation in a real-time embedded device. The first part is done by running a simulation of the control of the inverted pendulum on MATLAB [26]. As explained before, since there is not a physical pendulum, a model is used. The specifications of the model are from a MATLAB project from Michigan University already mentioned before [22]. They are given in figure 3.9.

With this model, the theory described in subsection 2.2 and the MATLAB functions an LQR controller is synthesized and modeled. Finally, these elements lead to a discrete-time
3.4. Inverted Pendulum Control and Performance Analysis

\[ M = 0.5; \quad \% \text{Mass Of the cart (kg)} \]
\[ m = 0.2; \quad \% \text{Mass of the pendulum (kg)} \]
\[ b = 0.1; \quad \% \text{Coefficient of friction for cart (0.1 N/m/sec)} \]
\[ I = 0.006; \quad \% \text{Mass moment of inertia of the pendulum (kg.m}^2) \]
\[ g = 9.8; \quad \% \text{Standard gravity (N/kg)} \]
\[ l = 0.3; \quad \% \text{Length to pendulum center of mass (m)} \]

Figure 3.9: Inverted Pendulum Model Specifications on Matlab

description of all the plant. Therefore, a control loop which controls the inverted pendulum model with a chosen period, is simulated. The detail of the simulation code is presented in appendix B. In addition, the pendulum has an initial state that is shown in table 3.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>0</td>
</tr>
<tr>
<td>Speed</td>
<td>0</td>
</tr>
<tr>
<td>Angle (deviation)</td>
<td>19.6°</td>
</tr>
<tr>
<td>Angular speed</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1: Initial State of the Inverted Pendulum for the simulation

This initial state corresponds to a simple deviation of 19.6° of the pendulum from its upright position. It is almost a step response simulated here during approximately 10 ms.

For the simulation the sampling period chosen is 5 ms. It means that every 5 ms, the state of the pendulum is sampled and then follows the computation-actuating process which holds the control input value until the next period of sampling.

Conversely, there is a simulation with the Arduino and a simulated scheduler with non-constant delays. The idea is to simulate the control of the inverted pendulum with a control task run by the [RTKernel] on the Arduino board and also with a simulation of the scheduler on MATLAB. However, there are other tasks running on the micro-controller and the simulated scheduler at the same time, thus, the control task can be delayed and preempted. Consequently, the time between applying two different control values becomes non constant. On the computer, with a program that simulates the EDF-scheduler for the tasks there is also a variation of delays but this time the tasks are not really running. Actually, the execution time of the tasks is fixed as either the average execution time or the maximum execution time. Thus, non-constant delays from the Arduino simulation are compared with the same simulation but on the computer with a simulated scheduler. The table 3.2 shows the different tasks running with their deadlines, period, release time and execution time.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Release (ms)</th>
<th>Period (ms)</th>
<th>Deadline (ms)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>15.4</td>
<td>5</td>
<td>16</td>
<td>1.02</td>
</tr>
<tr>
<td>Task 2</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>1.09</td>
</tr>
<tr>
<td>Task 3</td>
<td>1.3</td>
<td>5</td>
<td>2</td>
<td>0.77</td>
</tr>
<tr>
<td>Task 4</td>
<td>11</td>
<td>5</td>
<td>14</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table 3.2: Properties of the first set of tasks running on the Arduino. The control task is the task 2. Execution times are minimum, average and maximum observed, respectively.

The table 3.3 shows another set of tasks used for simulation.
Table 3.3: Properties of the second set of tasks running on the Arduino. The control task is the task 2. Execution times are minimum, average and maximum observed, respectively.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Release (ms)</th>
<th>Period (ms)</th>
<th>Deadline (ms)</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0.06</td>
</tr>
<tr>
<td>Task 2</td>
<td>1.1</td>
<td>5</td>
<td>3</td>
<td>0.58</td>
</tr>
<tr>
<td>Task 3</td>
<td>6.2</td>
<td>5</td>
<td>7</td>
<td>0.32</td>
</tr>
<tr>
<td>Task 4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>0.37</td>
</tr>
</tbody>
</table>

So with the schedule, it is possible to see how the pendulum is controlled with a non-constant delay for the control task. It remains to evaluate the control performance of the control in both case.

**Control Performance Analysis**

The control performance of the application is evaluated with the value of the control cost described in the theory chapter. The equation is recalled here:

\[
\sum_{k=0}^{N} x(k)^T Q x(k) + u(k)^T R u(k),
\]

In conclusion, with the control cost, it is possible to determine the control performance of the control application running either on the Matlab simulation or with the Arduino.
4 Results

This chapter presents the results of the implementation of the methods described in the previous chapter. It is divided into four parts corresponding to the adaptation of the TRTKernel. The implementation of the server-based Kernel, the results of the static timing analysis with the Bound-T tool and, finally, the resulting simulation of the control application and its performance are presented.

The interpretation of the results is given in the next chapter (5).

4.1 Modification and Adaptation of the TRTKernel to a New Architecture

So far, in the theory chapter, the architecture of the device used has been explained, as well as the main specifications of how the TRTKernel works. Since this kernel was aimed for an ATmega8L and an ATmega328p was used, modifications have been made to adapt the kernel. In first place, architectural differences are pointed out and then main code changes are explained.

Architectural Differences

There are several differences between the two micro-controllers but not all are of concern for the thesis. First thing is the clock frequency which was 14.7456 MHz for the ATmega8L, and is 16 MHz for the ATmega328p. The clock frequency will change the clock resolution and the life of the system which is reduced with the latter. On the other hand, it speeds up the execution of the instructions.

The other important change for the architecture is the size of the SRAM which is twice the size of the ATmega8L. Likewise, the flash memory of the ATmega328p is four times bigger. All of this means that with the new architecture, there is more memory available for the program itself, for the tasks, as well as for the local variables. Nevertheless, the program itself does not need a lot of memory, so it is not a major change, whereas the SRAM is a rather limited resource, so it is a good improvement for the kernel, and it is even better in the new implementation described in the section 4.2. Table 4.1 sums up the differences mentioned above.

In addition to the changes in the architecture, modifications have been made in the code itself, and are presented in the next paragraph.
4.2 Implementation of The Server-Based Real-Time Kernel

Table 4.1: Main Architectural Differences between the ATmega8L and ATmega328p devices.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>ATmega8L</th>
<th>ATmega328p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (MHz)</td>
<td>14.7456</td>
<td>16</td>
</tr>
<tr>
<td>SRAM (bytes)</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>Flash Memory (bytes)</td>
<td>8K</td>
<td>32K</td>
</tr>
</tbody>
</table>

Modifications of the code

Since the architectures of the two devices are quite close, not a lot of changes are needed to make the kernel work on the ATmega328p. Nevertheless, it requires to study in depth these architectures in order to understand which part of the code needs to be changed and how it has to be changed. It results that the main modification is only to suppress one byte in the initialization of a task’s stack. However, other modifications have been made, although they do not really change the main functionality of the kernel, they are shown in the list below:

- Time scale changed from “seconds” to “milliseconds”
- Addition of new macro for the conversion between “TICKS” and “milliseconds”
- Addition of a structure for timers to save their state
- Addition of freeze and resume timer functions to stop the count of time with timers
- Possibility to communicate with Matlab via an implemented Arduino library added (only data relevant for the thesis)
- Access and modification of tasks’ timing values in “seconds” and “milliseconds” added
- Addition of a function to start the kernel, instead of a kernel starting when created
- Use of a custom “Push/Pop” function for the interrupt (done by user and not by compiler)

The explicit list of the added functions is given in appendix C.

4.2 Implementation of The Server-Based Real-Time Kernel

In addition to using the TRTKernel for controlling the inverted pendulum, part of the thesis is to implement a server-based real-time kernel. The principle is explained in the method chapter (see section 3.2).

The implementation follows the same idea as in the TRTKernel. The main functionality lies in the interrupt routine. However, this time, there are two routines, one used for the global scheduler and the other one for the local scheduler. Since there should be an isolation between server, only the global scheduler is aware of every servers and only by means of their respective local scheduler. It results that the top level schedule does only four things, as described in the following list:

- Make the local scheduler of the running server save its context
- Release servers, choose the next one to run (following the algorithm mentioned in the method chapter and summed up in figure 4.5)
- Set the next time for the routine to be executed

The “clock” of the timer, which corresponds to the system clock frequency divided by the prescaler
4.2. Implementation of The Server-Based Real-Time Kernel

- Make the local scheduler of the next running server restore its context

Likewise, the local scheduler manages the following functions:

- Save context of the running task
- Schedule the next task to run
- Set the next time for the routine to be executed
- Restore the context of the next task to run

As explained in the method part, both the local and global scheduler use an EDF scheduling policy already detailed before (see section 2.1).

Using two different routines means also using two different timers in order to generate interrupts. In the ATmega328p there are three different timers which are not exactly the same. Firstly, except the 16-bit Timer 1, timers are 8-bit. It results that the Timer 2 used for the local scheduler has a shorter life time. However, the two are extended with a 32-bit register, then, the same maximum value is reachable for the two. Moreover, the way of setting the different timers is the same for each timer, even if there is some slight differences that can easily be found in Atmel’s data-sheet [9].

In order to understand a bit better the implementation of the Kernel, figure 4.1 shows its structure:

```c
struct kernel {
    uint8_t nbrOfTasks;
    uint8_t nbrOfServers;
    uint8_t running_srv;
    server_t servers[MAXNBRSERVERS+1];
    srvParam_t rsr[MA
XNBRSERVERS+1];
    uint8_t *memptr;
    uint32_t cycles;
    uint32_t nextHit;
    BOOL glbl_isr;
};
```

Figure 4.1: Structure of the Server-based Kernel

The kernel mainly keeps track of servers and tasks, corresponding resources, state of the free memory and timing information for using the timer of the global scheduler.

Then, the structure of the server is given in figure 4.2.

The structure of the server is composed of its current budget, its deadline and its state used for the algorithm of the server-based approach. Part of its structure is the local scheduler for the server, and the corresponding structure is shown in figure 4.3.

```c
struct server{
    uint32_t q;
    uint32_t d;
    s_state state;
    scheduler_t local_schdlr;
};
```

Figure 4.2: Structure of the server in the Server-based Kernel
4.2. Implementation of The Server-Based Real-Time Kernel

```c
struct scheduler {
    uint8_t nbrOfTasks;
    uint8_t running_task;
    task_t tasks[MAXNBRTASKS+1];
    uint32_t cycles;
    uint32_t nextHit;
};
```

Figure 4.3: Structure of the local scheduler used in the servers of the Server-based Kernel

```c
struct task{
    uint8_t spl ;
    uint8_t splh ;
    uint32_t release ;
    uint32_t deadline ;
    uint8_t state ;
};
```

Figure 4.4: Structure of a task in the Server-based Kernel

The local scheduler has a structure quite similar to the kernel, since they are both schedulers and because they are based on an EDF scheduling policy. Thus, the lower level scheduler keeps track of the tasks in the server and timing values for the use of the local timer.

Finally, in figure 4.4, the structure of a task is presented. It is the same as for the TRTKernel.

In addition, the API of the kernel is given in Appendix E.

To follow the algorithm described in the Lipari and Bini’s paper a state graph of how the servers are managed by the kernel is shown figure 4.5:

![State-graph of how servers are managed](image-url)

Figure 4.5: State-graph of how servers are managed
4.3 Timing Analysis of the Control Application

A server is characterized by the pair \((Q, P)\), with \(Q\) the maximum budget and \(P\) the period. It means that a server gets \(Q\) budget every period \(P\). There are also two internal variables \(q\) and \(d\). \(q\) represents the current budget remaining of a server. \(d\) is the “deadline” which is the next period, relative to the current time. In addition to the figure 4.5, the algorithm deals with three more rules:

- At any time \(t\), the global scheduler chooses a server among the active ones
- Servers can be preempted, then, their budget is not decremented anymore
- When a task is activated in an active server, \(q\) and \(d\) variables remain unchanged

The source code corresponding to this implementation is given in the Appendix F.

### 4.3 Timing Analysis of the Control Application

As explained before, to evaluate the execution time of a set of tasks, it is needed to compute the WCET (Worst Case Execution Time) of each task. This value is computed by a static timing analysis with the Bound-T tool [11]. The table 4.2 shows the results of the WCET calculation with the tool for several operations. These operations are a simple addition / multiplication of four integers or floating point values vectors in a loop. Finally, the execution time should have been analyzed for the control task in the context of the real-time kernel. However, it has not been done, as it is explained in chapter 5.

<table>
<thead>
<tr>
<th>Operation</th>
<th>WCET (clock Cycle)</th>
<th>WCET (µs)</th>
<th>WCET (TICKS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer +</td>
<td>147</td>
<td>9.1875</td>
<td>0.143554688</td>
</tr>
<tr>
<td>Integer ×</td>
<td>183</td>
<td>11.4375</td>
<td>0.178710938</td>
</tr>
<tr>
<td>Floating Point +</td>
<td>1561</td>
<td>97.5625</td>
<td>1.524414063</td>
</tr>
<tr>
<td>Floating Point ×</td>
<td>3399</td>
<td>212.4375</td>
<td>3.319335938</td>
</tr>
</tbody>
</table>

Table 4.2: WCET Analysis Results with Bound-T Tool.

The idea is to test the tool with different operations close to the control task and finally evaluate the control task itself.

### 4.4 Simulation of the Control Application and its Performance

This section presents the results of the simulation run on MATLAB with a modeled inverted pendulum. As mentioned in the method chapter, there are different simulations: one running with constant delay for the control task and others with non-constant delay. For the non-constant delay there is a comparison between the set of tasks running on the Arduino board and the same tasks running on the computer with a simulated scheduler. First, the simulated scheduler considers average execution time and then, worst-case execution time observed. This way, the importance of jitter can be observed because context switching and preemption will not have effect on the computer. For an easier comparison in the discussion chapter, results are organized as tables featuring the parameters of the simulation, and the control costs followed by the resulting curves for both computer simulation and Arduino timings.

The source code of the MATLAB simulation is given in the appendix B. Different tests are run to see how the system responses. The initial state of the pendulum is described in the Method chapter and corresponds to an angle deviation from the upright position. The period for the constant delay computer-based simulation is 5ms. A Gaussian noise is added to the control input value for each simulations and its variance is 0.2.

The different parameters which are changed for the simulations are:
4.4. Simulation of the Control Application and its Performance

• If it is compared to constant delay. Simulations 1 and 2.

• The execution time of the tasks for the simulated scheduler (average or maximum observed). Simulations 3 and 4 only.

• The importance of the variation of the non-constant delay for the Arduino

<table>
<thead>
<tr>
<th>Constant Delay</th>
<th>0.5ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-constant Delay Standard Deviation</td>
<td>1.2ms</td>
</tr>
<tr>
<td>Control Cost (Constant Delay)</td>
<td>85.1749</td>
</tr>
<tr>
<td>Control Cost (Non Constant Delay)</td>
<td>91.3457</td>
</tr>
</tbody>
</table>

Table 4.3: Simulation 1, Parameters and Control Costs.

<table>
<thead>
<tr>
<th>Constant Delay</th>
<th>0.5ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-constant Delay Standard Deviation</td>
<td>0.5ms</td>
</tr>
<tr>
<td>Control Cost (Constant Delay)</td>
<td>84.7425</td>
</tr>
<tr>
<td>Control Cost (Non Constant Delay)</td>
<td>87.8156</td>
</tr>
</tbody>
</table>

Table 4.4: Simulation 2, Parameters and Control Costs.

<table>
<thead>
<tr>
<th>Executions Time for Simulation</th>
<th>Average 1.2ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-constant Delay Standard Deviation</td>
<td>1.2ms</td>
</tr>
<tr>
<td>Control Cost (Simulation Non Constant Delay)</td>
<td>85.2646</td>
</tr>
<tr>
<td>Control Cost (Arduino Non Constant Delay)</td>
<td>90.8986</td>
</tr>
</tbody>
</table>

Table 4.5: Simulation 3, Parameters and Control Costs.

<table>
<thead>
<tr>
<th>Execution Times for Simulation</th>
<th>Maximum 1.2ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-constant Delay Standard Deviation</td>
<td>1.2ms</td>
</tr>
<tr>
<td>Control Cost (Simulation Non Constant Delay)</td>
<td>89.6994</td>
</tr>
<tr>
<td>Control Cost (Arduino Non Constant Delay)</td>
<td>91.8825</td>
</tr>
</tbody>
</table>

Table 4.6: Simulation 4, Parameters and Control Costs.

Here, only the angle response curves are shown, but the response of the inverted pendulum for the position is given in the appendix

For all the simulations, the curves are quite close, although the control costs are different.

---

2how much it is delayed from the 5ms period → standard deviation.
4.4. Simulation of the Control Application and its Performance

Figure 4.6: Simulation 1, Angle Response Curves

Figure 4.7: Simulation 2, Angle Response Curves
4.4. Simulation of the Control Application and its Performance

Figure 4.8: Simulation 3, Angle Response Curves

Figure 4.9: Simulation 4, Angle Response Curves
This chapter analyzes the results, as well as the method, before putting the thesis in a wider context.

5.1 Results

In this section, results are discussed. Firstly, the modification of the TRTKernel, then the server-based kernel and the static timing analysis are presented. Finally, a closer look at the control simulation results is taken.

Modifications of the TRTKernel

The TRTKernel has been successfully adapted to the new architecture of the ATmega328p. Since there is not a lot of differences between the previous architecture and the Arduino’s one, most of the details for the implementation are already given in the previous TRTKernel technical report [20].

Nevertheless, it is useful to have adapted the kernel to this device, since the Arduino Uno is a quite popular board. In addition, the implementation is done as an Arduino library, so it is easy to use it, and even more easier with the new API described in the Appendix E.

The Server-based Kernel

The kernel is implemented, but not completed. In fact, more tests are needed to make it fully work. Moreover, it doesn’t totally do the server abstraction. Indeed, only an EDF scheduler is available for server application. However, it is possible to adapt it to this principle without so much modifications because it has been thought for this.

The Static Timing Analysis

It is good and important to spend some time understanding the problem of static timing analysis and WCET. Moreover, it is needed in a real-time system to know the execution upper bound for every task, in order to satisfy a real-time constraints.

However, results of the timing analysis have not really been used for the control application. First because it is not possible to evaluate the real execution time of the control task,
since the Arduino is communicating with the computer and this communication time should not be taken in account for the analysis but does exist when using the application (even if timers are frozen when communicating). Also, the tool is not easy to use and was not first aimed for the ATmega328p architecture so it is not sure that the results are 100% accurate. Finally, the tool does not take into account the timing effect of the scheduling, e.g. preemption, context switching, etc.

The Control Application

Results show that there is an impact on the control performance when the application is subject to a non-constant delay. Figures do not seem to be very different between constant and non-constant delay simulations, but the control cost indicates a difference between the two. Indeed, control cost is increased by around 3 – 8% with a non-constant delay. Likewise, the increase is higher with a higher standard deviation of the non-constant delay. For instance, it goes from 3.69% to 7.24%, between the simulation 3 and 1.

Moreover, the simulations between the simulated scheduler and the Arduino are again quite close, but still control cost is different. The difference is bigger when the schedule is simulated with average execution time. It shows that effects from scheduling of tasks running on the same processor, which are ignored in theory, have an important impact since it increases the control cost.

Nevertheless, more simulations should be done, with different schedules and different sampling periods to see a bit more the effect of the non-constant delay.

5.2 Method

Here the method is discussed and follows the same arrangement that the previous section.

The TRTKernel

There are not really other means for modifying the kernel, because it is only an adaptation. In addition, added functions are just here to ease the use of the kernel.

Nevertheless, some criticisms can be made on the tools used. In fact, no debug tool has been used so it is not easy to understand some behaviors of the kernel, without such a tool. In addition, the Arduino IDE is not the best tool to work with for the development of something such as a Kernel. However, Atmel Studio software has been also used, with the advantage of the availability of a simulation tool integrated to it. The drawback is that it can only be used on a Windows Operating System. Moreover, a simulation tool is still not a debug tool, which is necessary since the Kernel interfaces with the hardware directly. It means that some hardware issues cannot be seen with a simulation.

Also, with more literature about the TRTKernel, it would have been easier to adapt it.

The Server-based Kernel

Same criticism as for the TRTKernel, the lack of debug tool is a real problem when developing a Kernel.

Moreover, this type of implementation is quite new and more theoretical than really applied. It results that there is not an example to compare with. Also, there are different ways of implementing this kind of Kernel and the algorithm found in the Lipari and Bini’s paper is not excessively detailed. Different interpretations can be made, so it leads to some difficulties on how to implement this kernel.

Again, the memory of the ATmega328p is a bit limiting for the server-based kernel, since each application takes lot of space of the SRAM to run.
5.3. The work in a wider context

The Static Timing Analysis
As mentioned before, the tool is not really easy to use, since some indications have to be given to the tool, by checking the assembly code of the program. Also, it was aimed for another Atmel device, even if the architectures are close enough to consider that the tool is still usable for the ATmega328p. However, it could be worth to spend some time on this question to be totally sure that the tool can be used in the case of this device.

In addition, it could have been good to use other tools in order to compare the results to be sure that the upper bound found is relevant.

The Control Application
Mainly, using a physical inverted pendulum would be more relevant for giving results on the question of control performance with a non-constant delay for the control application.

Linked to this, the imperfections of the model can give results different than in the real case.

Finally, more tests might give better insights on the effect on control performance of the non-constant delay problem.

5.3 The work in a wider context

This work is a good mean to understand the complexity of real-time systems and embedded applications. Even if the thesis is focused only on one control problem such as the inverted pendulum, results are relevant for all control applications.

In addition, using an Arduino board (although the criticisms made in the previous section), allows a wide range of people to experiment the same problems as in the thesis. Moreover, an EDF kernel is not fairly common, so the adaptation to the Arduino gives an easier access to this kind of kernel.

Likewise, the server-based kernel implementation is a new approach that may be good to consider for control applications. So there is the basis for developing this kind of kernel for other works.

Finally, to expand the work it could be good to use different kernels, different scheduling policies or to run other control applications.
Conclusion

The thesis aims to study the impact of timing issues during the implementation of embedded control applications. The focus is on the control of an inverted pendulum and the problem of the non-constant delay. The results show that there is a real impact on control performance when an application is subject to a non-constant delay due to the different tasks running on the same processor. The effect is evaluated by the computation of the control cost of the application which is increased with the increase of the standard deviation of the non-constant delay and the effects of scheduling.

Also, since the application should run on an Arduino Uno, and the kernel used was not made for this architecture, the TRTKernel has been adapted successfully to the architecture of the ATmega328p.

In addition, on the same device, a server-based kernel has been developed. Although it needs more tests and modifications to fully work, the basis of this approach are made and can be taken in order to have a working kernel.

As mentioned in section 5.3, future work might be to expand to a wider context the thesis, with experiments of different control application, different kernel, on different scheduling policies. Moreover, the server-based kernel needs to be tested more and to get some modifications.

In brief, answers to the questions stated at the beginning of the thesis have been given in this thesis, even if some more work is required to fix some remaining issues, in particular for the server-based kernel.
Bibliography


ISR(TIMER1_COMPA_vect, ISR_NAKED) {

    PUSHREGS();
    uint8_t running, oldrunning;
    struct task *t;
    uint8_t i;
    uint32_t now;
    uint32_t nextHit;
    int32_t timeleft;

    TIMSK1 = 0; // turn off output compare 1A ISR
    nextHit = 0x7FFFFFFF;
    oldrunning = kernel.running;
    running = 0;

    if (TIFR1 & (1<<TOV1)) {
        ++kernel.cycles;
        TIFR1 |= (1<<TOV1);
    }

    // Read clock
    now = (kernel.cycles << 16) + TCNT1;

    // Release tasks from TimeQ and determine new running task
    for (i=1; i <= kernel.nbrOfTasks; i++) {
        t = &kernel.tasks[i];
        if (t->state == TIMEQ) {
            if (t->release <= now) {
                t->state = READYQ;
            }
        }
    }
}
else if (t->release < nextHit) {
    nextHit = t->release;
}

if (t->state == READYQ) {
    if (t->deadline < kernel.tasks[running].deadline) {
        running = i;
    }
}

if (running != oldrunning) {
    // perform context switch?

    // store old context
    t = &kernel.tasks[oldrunning];
    t->spl = SPL;
    t->sph = SPH;

    // load new context
    t = &kernel.tasks[running];
    SPH = t->sph;
    SPL = t->spl;

    kernel.running = running;
}
kernel.nextHit = nextHit;

now = (kernel.cycles << 16) + TCNT1;
timeleft = (int32_t)nextHit - (int32_t)now;
if (timeleft < ISR_EXE_RESP) {
    timeleft = ISR_EXE_RESP;
}

if ((unsigned long)TCNT1 + timeleft < TCNT1_MAX) {
    OCR1A = TCNT1 + timeleft;
} else if (TCNT1 < TCNT1_MAX - ISR_EXE_RESP) {
    OCR1A = 0x0000;
} else {
    OCR1A = ISR_EXE_RESP;
}

TIMSK1 |= (1<<OCIE1A);

POPREGS();
RETI();
NOP();
Matlab Simulation Code

%% Pendulum specs
M = 0.5; % Mass of the cart (kg)
m = 0.2; % Mass of the pendulum (kg)
b = 0.1; % Coefficient of friction for cart (0.1 N/m/sec)
I = 0.006; % Mass moment of inertia of the pendulum (kg.m^2)
g = 9.8; % Standard gravity (N/kg)
l = 0.3; % Length to pendulum center of mass (m)

p = I*(M+m)+M*m*l^2; %denominator for the A and B matrices

%% State-space matrices
A = [0 1 0 0; 
     0 -(I+m*l^2)*b/p (m^2*g*l^2)/p 0; 
     0 0 0 1; 
     0 -(m*l*b)/p m*g*l*(M+m)/p 0];
B = [0; 
     (I+m*l^2)/p; 
     0; 
     m*l/p];
C = [1 0 0 0; 
     0 0 1 0];
D = [0; 
     0];

nb_system_states = 4;
nb_system_inputs = 1;
nb_system_outputs = 2;

states = {'x' 'x_dot' 'phi' 'phi_dot'}; % states names
inputs = {'u'}; % control input name
outputs = {'x'; 'phi'}; % system output names

%% Construct controller
sys_ss = ss(A,B,C,D,'statename',states,'inputname',inputs,'outputname',outputs);;

Tcs = 5/1000; % Controller sample rate (5 ms)
sys_d = c2d(sys_ss,Tcs,'zoh') % discrete system for controller

%% Discrete state-space matrices
A = sys_d.a;
B = sys_d.b;
C = sys_d.c;
D = sys_d.d;
Q = C'*C ; % state-cost matrix
R = 1; % performance index matrix
[K] = dlqr(A,B,Q,R) % Controller

Ac = [(A-B*K)];
Bc = [B];
Cc = [C];
Dc = [D];
Tss = 1/1000;
sys_cl = ss(Ac,Bc,Cc,Dc,Tss,'statename',states,'inputname',inputs,'outputname',outputs);
%sys_d = c2d(sys_cl,'zoh') % discrete system for pendulum

%% Init Values

t = 0:Tss:0.0012; % Time vector (0.0012+first actuating delay)
t_ar = 0:Tss:0.0012;
cc = 0 ; % control cost
cc_ar = 0 ;
now = 0 ; % Current time
now_ar = 0 ;
simu_time = 10.7 ; % simu time (10 s)
h = 1 ; % Control task samples index/nbr
k = 1 ; % Control task samples index/nbr

43
u_val = 0; % Initial control value
u_val_ar = 0;

noiseVariance = var_value; % Control value noise

t_negAgle = simu_time/2; % Time when pendulum pulled left (if so)

% (position, speed, derivation angle, derivation angle speed)
% Initial State
x = x_zero; xdot = xdot_zero; theta = theta_zero;
thetadot = thetadot_zero;
x_ar = x_zero; xdot_ar = xdot_zero; theta_ar = theta_zero;
thetadot_ar = thetadot_zero;
X0=[x xdot theta thetadot]'; % pendulum state matrix
X0_ar=[x_ar xdot_ar theta_ar thetadot_ar]'; % pendulum state matrix

% Buffer to keep track of output, time and control value
y_buff = [x theta];
t_buff = 0;
u_buff = u_val;

y_buff_ar = [x_ar theta_ar];
t_buff_ar = 0;
u_buff_ar = u_val_ar;

%% Control Loop
while (now < simu_time)
    nb_samples = length(t); % Update, depending on t
    nb_samples_ar = length(t_ar);
    noise = normrnd(0,noiseVariance);
    u_val = u_val + noise;
    u_val_ar = u_val_ar + noise;

    % Discrete input control value (for each samples of simu)
    u = u_val*ones(nb_samples, nb_system_inputs);
    u_ar = u_val_ar*ones(nb_samples_ar, nb_system_inputs);

    % Simulate pendulum behaviour
    % X0 -> Initial / previous state matrix
    % t -> time vector
    % u -> control input
    % sys_cl -> pendulum discrete model
    [y,t,X]=lsim(sys_cl,u,t,X0);
    [y_ar,t_ar,X_ar]=lsim(sys_cl,u_ar,t_ar,X0_ar);
    % y -> output matrix (x theta)
    % t -> time vector
    % X -> state matrix

    X0 = X(nb_samples,:); % Last state sample
    X0_ar = X_ar(nb_samples_ar,:);

y_buff = [y_buff ; y(end,1) y(end,2)]; % Store last output sample
y_buff_ar = [y_buff_ar ; y(end,1) y(end,2)];

u_buff (end + 1) = u_val ; % Store actuating value
u_buff_ar (end + 1) = u_val_ar ;

u_val = -K*X0' ; % Control operation
u_val_ar = -K*X0_ar' ;

cc=cc+X0*(Q*X0') + u_val*(R*u_val') ; % Control cost
cc_ar=cc_ar+X0_ar*(Q*X0_ar') + u_val_ar*(R*u_val_ar') ;

now = t(end) ; % Update current time value
now_ar = t_ar(end) ;

k = k + 1 ; % Increase control sample value
h = h + 1 ;

t_buff(end + 1) = now ; % Overall time
t_buff_ar(end + 1) = now_ar ;

t = now:Tss:(simu_jitt.dt(h)+now) ;
t_ar = now_ar:Tss:(arduino.dt(k)+now_ar) ;
if k >= numel(arduino.dt)-1 || h >= numel(simu_jitt.dt)-1
    break ;
end
end
control_cost_sim = cc
control_cost_arduino = cc_ar
```c
#define MILLISECONDS2TICKS(T) ( (uint32_t)( (((float) T)*TICKSPERSECOND/1000.0)))
#define TICKS2SECONDS(T) ( (float)( (float)(T)/(float)(TICKSPERSECOND) ) )
#define TICKS2MILLISECONDS(T) ( (float)( (float)(T*1000)/(float)(TICKSPERSECOND)))

/************************************************************************/
// Added functions
void trtStartKernel(void) ;
float trtCurrentTime_ms(void) ;
float trtGetRelease_ms(void) ;
float trtGetDeadline_ms(void) ;
float trtGetRelease_s(void) ;
float trtGetDeadline_s(void) ;
uint32_t trtSetRelativeRelease(float add_release) ;
uint32_t trtSetRelativeDeadline(float add_deadline) ;
uint32_t trtSetAbsoluteRelease(float abs_release) ;
uint32_t trtSetAbsoluteDeadline(float abs_deadline) ;
void trtFreezeTimers(timer_status_t* t) ;
void trtResumeTimers(timer_status_t* t) ;

/************************************************************************/
void trtInitKernel(int idlestack) ; // !\ Modif
void trtCreateTask(void (*fun)(void*), uint16_t stacksize, float release, float deadline, void *args) ; // !\ modif : release and deadline in ms, conversion in the function

/************************************************************************/
struct timer_status{ 
    uint8_t tccr0b ;
    uint8_t tccr1b ;
    uint8_t tccr2b ;
} ;
```
typedef struct timer_status timer_status_t ;

/************************************************************************/
// MatlabSerial
void establishContact(void) ;
void matlabSerialSend(uint8_t t_idx, float t_time) ;
void matlabSerialControl(volatile float arg[4], unsigned char cmd,\
uint8_t t_idx, float t_time) ;
This appendix shows the results of the simulation for the control application of the inverted pendulum described in this thesis. The conditions for each simulations, as well as the results for angle response are given in the results chapter (see 4.4).

Figure D.1: Simulation 1, Position Response Curves
Figure D.2: Simulation 2, Position Response Curves

Figure D.3: Simulation 3, Position Response Curves

Figure D.4: Simulation 4, Position Response Curves
API of the Server-based Kernel

/* Local scheduler / Tasks */
void srvInitLocalScheduler(uint8_t s_idx) ;
void srvCreateLocalTask(void (*fun)(void*), uint16_t stacksize, uint8_t s_idx,
uint32_t release, uint32_t deadline, void *args) ;
void srvLocalSleepUntil(uint32_t release, uint32_t deadline) ;
uint32_t srvGetTaskRelease(void) ;
uint32_t srvGetTaskDeadline(void) ;
uint8_t srvGetTaskState(void) ;
void srvTaskTerminate(void) ;
void srvSetTaskRelease(uint32_t release) ;
void srvSetTaskDeadline(uint32_t deadline) ;
void srvSetTaskState(uint8_t state) ;
uint32_t srvLocalCurrentTime(void) ;
uint8_t srvGetLocalTaskIdx(void) ;

/* Server */
void srvCreateServer(void (*idle_func)(void), uint16_t idlestack, uint8_t s_idx,
uint32_t budget, uint32_t period) ;
void idle(void) ;
void srvSleepUntil(uint32_t budget, uint32_t deadline) ;
uint32_t srvGetQ(void) ;
uint32_t srvGetBudget(void) ;
uint32_t srvGetPeriod(void) ;
s_state srvGetState(void) ;
void srvTerminate(void) ;
void srvSetQ(uint32_t maxbudget) ;
void srvSetBudget(uint32_t budget) ;
void srvSetPeriod(uint32_t period) ;
void srvSetDeadline(uint32_t deadline) ;
void srvSetState(s_state state) ;
/* Kernel */
void srvInitKernel(int idlestack) ;
void srvStartKernel(void) ;
uint32_t srvCurrentTime(void) ;
void srvFreezeTimers(timers_p* t) ;
void srvResumeTimers(timers_p* t) ;
unsigned int TIM16_ReadTCNT1(void) ;
unsigned int TIM8_ReadTCNT2(void) ;
```c
//...
for (i = 1; i <= kernel.nbrOfServers; i++) { // For each server
    s = &kernel.servers[i];
    // Pointer to current server local scheduler
    local_scheduler_p = &(s->local_schdlr);
    if (s->state == S_SUSPENDED) {
        if (OCR1A >= s->d) { // if interrupt has been generated because deadline(next period)
            // - = at time d - of suspended server
            s->state = S_READY;
        } else if (s->d < nextHit) // next "release" for the suspended server is its deadline
            // (as defined when activated)
            nextHit = s->d;
    }
    if (s->state == S_INACTIVE) { // Among all tasks of the server
        for (j = 1; j <= local_scheduler_p->nbrOfTasks; j++) {
            t = &(local_scheduler_p->tasks[j]);
            // If there is at least one task which needs to be activated
            if (t->state == TIMEQ) {
                // If a task is ready to be activated, then server is ready to run
                s->state = S_READY;
                break;
                // We now that this server is ready now,
                // so we don’t need to find another active task or its next release
            }
        }
    }
}
```

// If server is ready to run (has been released),
// It can be activated in order to be scheduled
if (s->state == S_READY) {
    s->q = (kernel.rsrc[i]).Q; // Allocate max budget respectively
    s->d = (kernel.rsrc[i]).P + now; // Given period at time t activated
    s->state = S_ACTIVE;
}
if (s->state == S_ACTIVE) {
    if (i == oldrunning) { // Decrement budget, if it was the running server
        int32_t tmp = (int32_t) s->q;
        tmp -= (int32_t) now;
        if (tmp < 0) { // job not finished yet
            s->q = 0;
        } else {
            s->q = tmp;
        }
        if (s->q <= 0) { // If no more budget
            for (j = 1; j <= local_scheduler_p->nbrOfTasks; j++) {
                // If there is at least one task which is not terminated
                if ((local_scheduler_p->tasks[j]).state != TERMINATED) {
                    s->state = S_SUSPENDED; // Suspend server
                    if (s->d < nextHit) {
                        // Next "release" for the suspended server is its deadline
                        nextHit = s->d;
                    }
                    break;
                }
            }
            // No more budget and all tasks are terminated
            if (s->state != S_SUSPENDED) {
                s->state = S_TERMINATED;
            }
        }
    } else if (s->d < (kernel.servers[running]).d) {
        running = i; // Still budget + EDF
    }
}
if (s->state == S_TERMINATED) {
    // If zero => division by zero (the case of idle task of idle server)
    if ((kernel.rsrc[i]).Q != 0) {
        if (now >= (s->d - (s->q)*((kernel.rsrc[i]).P) /
                ((kernel.rsrc[i]).Q))) {
            s->state = S_INACTIVE;
        }
        else if ((s->d - (s->q)*((kernel.rsrc[i]).P) /
                  ((kernel.rsrc[i]).Q)) < nextHit) {
            nextHit = (s->d - (s->q)*((kernel.rsrc[i]).P) /
                      ((kernel.rsrc[i]).Q));
            s->state = S_ACTIVE;
        }
    }
}
else if (s->d < (kernel.servers[running]).d) {
    running = i;
}
Next running server and its local scheduler

```c
int s = kernel.servers[running];
int local_scheduler_p = (s->local_schdlr);
```

Except the case of idle server

```c
if ((s->q + now) < nextHit && (kernel.rs[running]).Q != 0)
```

```c
    nextHit = s->q + now;  // when the next event is "budget is zero"
```

Update kernel values

```c
kernel.running_srv = running;
kernel.nextHit = nextHit;
```

//...