Implementation and analysis of a virtual platform based on an embedded system

Examensarbete utfört i Datorteknik vid Tekniska högskolan vid Linköpings universitet

av

Adam Sandstedt

LiTH-ISY-EX--14/4774--SE

Linköping 2014
Implementation and analysis of a virtual platform based on an embedded system

Examensarbete utfört i Datorteknik vid Tekniska högskolan vid Linköpings universitet av

Adam Sandstedt

LiTH-ISY-EX--14/4774--SE

Handledare: Andreas Ehliar
           Mathias Bergvall

Examinator: Mattias Krysander

Linköping, 11 juni 2014
The complexity among embedded systems has increased dramatically in recent years. During the same time has the capacity of the hardware grown to astonishing levels. These factors have contributed to that software has taken a leading role and time-consuming role in embedded system development.

Compared with regular software development, embedded development is often more restrained by factors such as hardware performance and testing capability.

A solution to some of these problem has been proposed and that is a concept called virtual platforms. By emulating the hardware in a software environment, it is possible to avoid some of the problems associated with embedded software development. For example is it possible to execute a system faster than in reality and to provide a more controllable testing environment.

This thesis presents a case study of an application specific virtual platform. The platform is based on already existing embedded system that is located in an industrial control system. The virtual platform is able to execute unmodified application code at a speed twice of the real system, without causing any software faults. The simulation can also be simulated at even higher speed if some accuracy losses are regarded as acceptable.

The thesis presents some tools and methods that can be used to model hardware on a functional level in an software environment.

The thesis also investigates the accuracy of the virtual platform by comparing it with measurements from the physical system. In this case are the measurements mainly focused of the data transactions in a controller area network bus (CAN).
Sammanfattning

Komplexiteten hos inbyggda system har ökat dramatiskt de senaste åren. På samma gång har kapaciteten hos hårdvara ökat till enastående nivåer.

Dessa faktorer har bidragit till att mjukvara har tagit en ledande och tidskrävan- de roll vid utvecklingen av inbyggda system.

Jämfört med normal mjukvaruutveckling är utveckling för inbyggda system mer begränsade av faktorer så som prestandan hos hårdvara och möjligheter till test av systemet.

En lösning på några av dessa problem har föreslagits och det är ett konceptet kallas för virtuella plattformar. Genom att emulera hårdvaran i en mjukvarumiljö är det möjligt att undvika några av de problem som är associerade med utveckling av mjukvara för inbyggda system. Till exempel är det möjligt att exekvera ett system snabbare än i verkligheten och det är också möjligt att tillhandahålla en mer kontrollerbar testmiljö för systemet.


Examensarbetet presenterar några användbara verktyg och metoder som kan användas för att modellera hårdvara på en funktionell nivå i en mjukvarumiljö.

Examensarbetet undersöker även noggrannheten hos den virtuella plattformen genom att jämföra den med mätningar gjorda på det fysiska systemet. Mätningarna är främst fokuserade på datatrafiken i en controller area network bus (CAN)
Abstract

The complexity among embedded systems has increased dramatically in recent years. During the same time has the capacity of the hardware grown to astonishing levels.

These factors have contributed to that software has taken a leading role and time-consuming role in embedded system development.

Compared with regular software development, embedded development is often more restrained by factors such as hardware performance and testing capability.

A solution to some of these problem has been proposed and that is a concept called virtual platforms. By emulating the hardware in a software environment, it is possible to avoid some of the problems associated with embedded software development. For example is it possible to execute a system faster than in reality and to provide a more controllable testing environment.

This thesis presents a case study of an application specific virtual platform. The platform is based on already existing embedded system that is located in an industrial control system. The virtual platform is able to execute unmodified application code at a speed twice of the real system, without causing any software faults. The simulation can also be simulated at even higher speed if some accuracy losses are regarded as acceptable.

The thesis presents some tools and methods that can be used to model hardware on a functional level in an software environment.

The thesis also investigates the accuracy of the virtual platform by comparing it with measurements from the physical system. In this case are the measurements mainly focused of the data transactions in a controller area network bus (CAN).
Acknowledgments

I would like to thank all persons who have helped me during this thesis project. Special thanks to my supervisor at Alten, Mathias Bergvall, to Björn Stenborg and to Mikael Ahkoila for your help and answers regarding hardware, software, measurements and many other problems. Additional thanks to Charles Hughes for your proofreading of the thesis.

Linköping, May 28 2014
Adam Sandstedt
# Contents

List of Figures xiv

List of Tables xv

Notation xvii

1 Introduction 1

1.1 Background ................................................. 1
1.2 Purpose ....................................................... 2
1.3 Related work ............................................... 2
1.4 Outline ....................................................... 3
1.5 Terminology ............................................... 4

2 Embedded systems 5

2.1 What is an embedded system? ............................. 5
2.2 The parts of an embedded system ....................... 6
2.2.1 CPU ....................................................... 6
2.2.2 Memory .................................................... 7
2.2.3 Peripherals ............................................... 7
2.3 Communication ............................................. 8
2.4 Embedded application code ............................... 8

3 Virtual platforms 9

3.1 What is a virtual platform? ............................... 9
3.2 Reasons for usage of virtual platforms ................. 10
3.3 The purpose of a virtual platform ....................... 10
3.3.1 Enable early and effective software development ... 10
3.3.2 Minimize the hardware dependency .................. 13
3.3.3 Increase testing efficiency ............................ 14
3.4 Problems and limitations with virtual platforms ....... 14
3.5 Development tools for virtual platforms .............. 15
3.5.1 Development applications ............................. 15
3.5.2 Development languages ............................... 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6 Open virtual platforms</td>
<td>16</td>
</tr>
<tr>
<td>4 Introduction to the physical platform</td>
<td>19</td>
</tr>
<tr>
<td>4.1 Platform overview</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Platform components</td>
<td>19</td>
</tr>
<tr>
<td>4.2.1 CPU</td>
<td>20</td>
</tr>
<tr>
<td>4.2.2 Memories</td>
<td>20</td>
</tr>
<tr>
<td>4.2.3 Peripherals</td>
<td>20</td>
</tr>
<tr>
<td>4.3 Application</td>
<td>21</td>
</tr>
<tr>
<td>4.4 External parts</td>
<td>21</td>
</tr>
<tr>
<td>5 Virtual platform modelling using OVP</td>
<td>23</td>
</tr>
<tr>
<td>5.1 Generation of a virtual platform</td>
<td>23</td>
</tr>
<tr>
<td>5.1.1 Platform file</td>
<td>23</td>
</tr>
<tr>
<td>5.1.2 Application code</td>
<td>25</td>
</tr>
<tr>
<td>5.1.3 Peripheral files</td>
<td>25</td>
</tr>
<tr>
<td>5.2 Simulation</td>
<td>26</td>
</tr>
<tr>
<td>5.2.1 The concept of time</td>
<td>27</td>
</tr>
<tr>
<td>5.3 Debugging</td>
<td>29</td>
</tr>
<tr>
<td>6 Implementation of the virtual platform</td>
<td>31</td>
</tr>
<tr>
<td>6.1 Overview of the virtual platform</td>
<td>31</td>
</tr>
<tr>
<td>6.1.1 Toolchain</td>
<td>33</td>
</tr>
<tr>
<td>6.1.2 Memory map</td>
<td>33</td>
</tr>
<tr>
<td>6.1.3 Terminology</td>
<td>34</td>
</tr>
<tr>
<td>6.2 Modelling methodology</td>
<td>34</td>
</tr>
<tr>
<td>6.3 Modelling tools</td>
<td>35</td>
</tr>
<tr>
<td>6.3.1 Registers</td>
<td>35</td>
</tr>
<tr>
<td>6.3.2 Nets</td>
<td>36</td>
</tr>
<tr>
<td>6.3.3 Callbacks</td>
<td>36</td>
</tr>
<tr>
<td>6.3.4 State machines</td>
<td>37</td>
</tr>
<tr>
<td>6.3.5 Timers</td>
<td>37</td>
</tr>
<tr>
<td>6.4 Processor core</td>
<td>38</td>
</tr>
<tr>
<td>6.5 CPU-peripherals</td>
<td>38</td>
</tr>
<tr>
<td>6.5.1 Endianness correction</td>
<td>38</td>
</tr>
<tr>
<td>6.5.2 GEN</td>
<td>39</td>
</tr>
<tr>
<td>6.5.3 SER</td>
<td>41</td>
</tr>
<tr>
<td>6.5.4 MEM</td>
<td>41</td>
</tr>
<tr>
<td>6.5.5 EFE</td>
<td>41</td>
</tr>
<tr>
<td>6.5.6 DMA</td>
<td>42</td>
</tr>
<tr>
<td>6.6 Board-peripherals</td>
<td>42</td>
</tr>
<tr>
<td>6.6.1 RAM</td>
<td>42</td>
</tr>
<tr>
<td>6.6.2 Flash</td>
<td>42</td>
</tr>
<tr>
<td>6.6.3 LED</td>
<td>43</td>
</tr>
<tr>
<td>6.6.4 RS232</td>
<td>44</td>
</tr>
<tr>
<td>6.6.5 SPI</td>
<td>45</td>
</tr>
</tbody>
</table>
6.6.6 EEPROM ........................................ 45
6.6.7 RTC ........................................... 47
6.6.8 CAN ........................................... 50
6.6.9 Response generation .......................... 52
6.7 The complete system ............................. 54

7 Measurements 55
7.1 Overview ........................................ 55
7.2 Measurement tools ............................... 56
7.2.1 Physical platform ............................. 56
7.2.2 Virtual platform .............................. 56
7.3 Metrics .......................................... 56
7.3.1 CPU load ..................................... 56
7.3.2 Duration ...................................... 56
7.3.3 Timing errors ................................. 56
7.3.4 Relative error ................................. 57
7.4 Parameters ...................................... 57
7.4.1 Theoretical bus transmission delays ...... 58
7.4.2 Estimation of MIPS value .................. 61
7.4.3 Estimation of response time on external boards 61
7.4.4 Initial simulation parameter values .......... 62
7.5 Measurement scenarios .......................... 63
7.5.1 Scenario 1 ..................................... 64
7.5.2 Scenario 2 ..................................... 64
7.5.3 Scenario 3 ..................................... 65
7.6 Performed measurements ......................... 65
7.6.1 CPU load ..................................... 65
7.6.2 Transmission delay ........................... 65
7.6.3 Timing error measurements ................ 66
7.6.4 Simulation speed .............................. 71
7.6.5 SPI accesses .................................. 71

8 Results and comparisons 75
8.1 Comparisons between the virtual and physical platform 75
8.1.1 Reliability .................................... 75
8.1.2 Simulation speed ............................. 75
8.1.3 Timing ........................................ 76
8.2 Modelling ....................................... 78
8.3 The developed virtual platform compared with other tools 79

9 Conclusions 81

10 Future work 83
10.1 Dynamic response generation .................. 83
10.2 Ethernet/DMA implementation ................ 83
10.3 RAM timing .................................... 84
A  Example code  87
   A.1  platform.c .................................................. 87
   A.2  rtc.h .......................................................... 93
   A.3  rtc.c .......................................................... 94

B  Measurement data  103
   B.1  Start-up ....................................................... 103
   B.2  Shutdown ...................................................... 103

Bibliography  107
List of Figures

2.1 A simple embedded system. ........................................... 6
3.1 A comparison between consecutive and concurrent development. . 11
3.2 Development with a physical platform. ........................... 12
3.3 Development with a virtual platform. .............................. 13
4.1 Simplified model of the physical platform. ......................... 20
4.2 The application stack for the system. ............................. 22
5.1 Overview of a simulation environment. ........................... 28
5.2 Simulation time flow. ............................................... 29
6.1 Overview of the simulation environment. .......................... 32
6.2 The virtual platform and its environment. ....................... 32
6.3 The virtual platform toolchain. ................................... 33
6.4 Development methodology. ....................................... 36
6.5 Illustration of a callback function. ............................... 37
6.6 An example of the internal structure of a CPU-peripheral. ...... 38
6.7 Solution of the endian problem. .................................. 39
6.8 The structure of the flash model. ................................ 43
6.9 State machine for flash model. .................................. 44
6.10 Illustration of the LED model. ................................... 44
6.11 Illustration of the RS232 model. ................................ 45
6.12 Illustration of the SPI model. .................................... 46
6.13 Illustration of the EEPROM model. ............................... 47
6.14 State machine for the EEPROM model. ......................... 47
6.15 Illustration of the RTC model. ................................... 48
6.16 State machine for RTC model. .................................. 49
6.17 State machine for the RTC time registers. ...................... 49
6.18 State machine for the RTC RAM registers. ...................... 50
6.19 Illustration of the CAN model. .................................. 51
6.20 State machine for the CAN device. .............................. 52
6.21 Illustration of the response generation. ......................... 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Transmission chain for CAN messages.</td>
<td>58</td>
</tr>
<tr>
<td>7.2</td>
<td>CPU load vs MIPS.</td>
<td>61</td>
</tr>
<tr>
<td>7.3</td>
<td>Measurement scenarios.</td>
<td>64</td>
</tr>
<tr>
<td>7.4</td>
<td>The error values during start-up</td>
<td>68</td>
</tr>
<tr>
<td>7.5</td>
<td>The error values during start-up</td>
<td>68</td>
</tr>
<tr>
<td>7.6</td>
<td>Measurements of SPI accesses on the virtual platform.</td>
<td>73</td>
</tr>
</tbody>
</table>
List of tables

2.1 An example of a 16-bit virtual memory. ........................................... 8
4.1 CPU modules. .................................................................................. 20
5.1 Peripheral file structure. ................................................................. 25
6.1 Hardware models. ............................................................................. 33
6.2 Virtual memory map. ....................................................................... 34
6.3 Big endian versus little endian. ......................................................... 39
6.4 Messages sent over the CAN bus. ...................................................... 53
7.1 Response times for messages sent from the external boards. ............ 62
7.2 Nominal simulation parameters. ......................................................... 63
7.3 CPU load on different measurement scenarios. ................................. 65
7.4 Delay over the whole transmission chain. ......................................... 66
7.5 Simulation results from the start up, using a time slice of 1\(\mu\)s. .... 67
7.6 Simulation results from the shutdown, using a time slice of 1\(\mu\)s. .... 67
7.7 Relative errors from both start-up and shutdown, using a time slice of 1\(\mu\)s. ................................................................. 69
7.8 Simulation results from the start up, using a time slice of 1ms. ......... 70
7.9 Required host time for the different PSE units. .................................. 71
7.10 Accesses over the SPI bus. ............................................................. 74
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviations</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CFI</td>
<td>Common Flash Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read Only Memory</td>
</tr>
<tr>
<td>ELF</td>
<td>Executable and Linkable Format</td>
</tr>
<tr>
<td>ESL</td>
<td>Electric and System Level</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GDB</td>
<td>GNU Project Debugger</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Input Output</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Descriptive Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ISS</td>
<td>Instruction Set Simulator</td>
</tr>
<tr>
<td>JIT</td>
<td>Just In Time</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access controller</td>
</tr>
<tr>
<td>MII</td>
<td>Media Independent interface</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multiprocessor System on a Chip</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OVP</td>
<td>Open Virtual Platform</td>
</tr>
<tr>
<td>PSE</td>
<td>Peripheral Simulation Engine</td>
</tr>
<tr>
<td>QEMU</td>
<td>Quick Emulator</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>RSP</td>
<td>Remote Serial Protocol</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RSP</td>
<td>Remote Serial Protocol</td>
</tr>
<tr>
<td>RTC</td>
<td>Real Time Clock</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transaction Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SOC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modelling</td>
</tr>
<tr>
<td>TTM</td>
<td>Time to Market</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
</tbody>
</table>
1

Introduction

1.1 Background

The development and testing of software to embedded systems can usually prove to be an challenging and time-consuming task. Compared with ordinary software development aimed towards general purpose processors (GPP), embedded software tend to rely much more on the available hardware. This means that the development can suffer and be delayed if a proper hardware model not is available in an early stage of the development. Embedded software development also often lacks the excellent debugging capabilities that an integrated development environment (IDE) can offer in the case of regular software development.

With the advent of multi-core processors in embedded systems the requests for extended debugging capabilities have increased. Also, to be able to keep the time to market (TTM) demands, embedded software developers are in need of a reliable reference model before the actual hardware is available.

Virtual platforms have been proposed as a solution to these problems. A virtual platform means that the whole system, hardware and all, is emulated in software. If this is done early in the development it is possible for the embedded software developers to have an early reference model. Another benefit is that a system simulated in software is much easier to control and monitor than real hardware.

The only problem is that when hardware is modelled in software it must behave as intended and ideally it should behave exactly as it would do in real hardware. That is something that is hard to guarantee and the only possibility to find out is when the real hardware is available. If the virtual platform is modelled badly it just might mean that the extra time spent modelling was of no benefit.
1.2 Purpose

The purpose of this thesis is to implement and investigate the capabilities of a virtual platform. The virtual platform is based on an already existing embedded system in order to enable comparisons between simulation and reality.

To help with this a development environment called Open virtual platform (OVP) will be used. OVP offers an Application Programming Interface (API) for construction of virtual platforms and also processor-core models for several common brands [2]. Additional information about OVP will be presented in Section 3.6.

The work has been divided into two parts. The first part consists of the implementation of the virtual platform and the second part consists of measurements on the real hardware. The data from the measurements is then compared with the virtual platform to investigate the reliability of the simulation.

The main goals of this master thesis are the following:

- Investigate how a virtual platform can be modelled to allow unmodified application code to be executed on it without faults.
- Analyse which part of the system that must be modelled with good timing precision and which parts that can be modelled less accurately. In other words, where must the delays be simulated? Is it necessary to simulate all delays? Which delay have the largest influence on the system?
- Perform comparisons between the real and simulated systems in order to conclude if events that happen in the real system can be recreated on the virtual one. Does these events share the same timing on the virtual and physical platforms?
- Analyse if the chosen development tool is fitting for the requirements of this platform or if a more/less accurate tool could be used instead.

1.3 Related work

The area of virtual platforms is still relatively young and it is not yet a common subject. This means that there are rather few books regarding this subject. One book that however does this is for example [27].

In the research field there is more information that can be found about virtual platforms. The main interest in the research seems aimed at the modelling and simulation of multi-core architectures. See for example [9] and [17].

Since the simulation of full systems can be placed somewhere between Functional level modelling and Register transfer level modelling (RTL) many papers investigates the performance of virtual platforms compared with RTL and how different simulation methods effect the simulation capability. See for example [24] and [28]. In this research field is mostly different simulation mechanisms addressed. This will not be the main focus in this thesis, but it is of some interest to see
how simulation of virtual platforms is performed and how accuracy effect the simulation speed.

Since the use of virtual platforms enables better ability to control and monitor the system some research is focused on something that is called fault injection. Some examples of this are [8] and [18].

At last there are also some research that focus on testing and verification aspects of virtual platforms. See for example [12] and [14]. These papers discuss some modelling concepts that are of interest in thesis. These papers also discusses the modelling of full systems which exactly is what will be done in this thesis.

In general it can be said that much research is focused at simulation and modelling of virtual platforms on a level a little bit higher than RTL. Most of these platforms are cycle accurate which is by far too accurate for the intended platform in this thesis. This platform is instead intended to achieve instruction accuracy to provide faster simulation. This platform will simulate at the same level as the full system simulator Simics [21]. The difference is that this platform will be an application specific simulator with much lower functionality than Simics.

1.4 Outline

This section presents a chapterwise overview of the thesis.

• **Chapter 1** presents the background and purpose of the thesis.
• **Chapter 2** gives an introduction to embedded systems in general.
• **Chapter 3** gives an introduction and explanation to the concept of virtual platforms.
• **Chapter 4** gives a description of the available embedded system on which the virtual platform will be based on.
• **Chapter 5** describes how Open virtual platform can be used to model virtual platforms.
• **Chapter 6** describes how the virtual platform was implemented.
• **Chapter 7** describes how measurements on the physical and virtual platform were performed.
• **Chapter 8** presents results and comparisons between the physical and virtual platform.
• **Chapter 9** presents the conclusions of the thesis work.
• **Chapter 10** presents suggestions to possible future work and improvements to the thesis.
1.5 Terminology

Here is an explanation of frequently occurring terms in the thesis.

**Physical platform**: A system existing in physical hardware.
**Virtual platform**: A system where the hardware is modelled in software.
**Host**: A computer/system that the virtual platform is executed on.
This chapter aims to give an introduction of the basics in embedded systems. Readers that feel that they already are accustomed to embedded systems can skip this chapter if they like.

2.1 What is an embedded system?

There are many definitions of what an embedded system is. The definitions tend to vary and are also often somewhat unclear. Noergaard gives a short and in my opinion good definition of an embedded system is:

"An embedded system is an applied computer system" [25]

This means that an embedded system, in opposition to a GPP, is dedicated to do one or a few specific tasks.

An embedded system is a combination of hardware and software and can be found in a large number of products such as pacemakers, cars, telephones, washing machines and much more. In 2009 it was estimated that 98 % of all commercially sold microprocessors could be found in embedded systems. [11]

One of the first embedded systems was the Apollo Guidance Computer. This system was produced for the Apollo lunar missions and served as the control and navigation system for the spacecraft. It was one of the first systems built using integrated circuits (IC) and had a Random Access Memory (RAM) of 2000 words and a fixed memory of 36 000 words. [1]

Luckily the capacity of integrated circuits has increased in the last 50 years, but embedded systems still remains a challenging area today as it was then.
Due to the tight dependency between software and hardware in embedded systems, development is often constricted by a number of factors. This can be things such as limitations of power consumption in handheld devices, real time applications that have a hard deadline on computations and so on.

2.2 The parts of an embedded system

To put it crudely, an embedded system can be divided into three parts: CPU, memory, and peripherals. A simple example of an embedded system is shown in Figure 2.1.

2.2.1 CPU

The CPU, often called a microcontroller, is the core of the embedded system. It is capable to execute a specific set of instructions that it receives from the application code. The CPU often also contains additional modules that control things such as memory accesses and bus communication. [7]

Since embedded systems often are supposed to do very specific tasks, CPUs in embedded systems can be quite different from ordinary GPPs. The CPU used in an embedded system can for example be an Application Specific Integrated Circuit (ASIC) that is designed for a special purpose, for example low power consumption or specific computational tasks. One common ASIC type is Digital Signal Processing processors (DSP) that is designed to handle signal processing in an efficient way. [20]
2.2 The parts of an embedded system

2.2.2 Memory

In general two main types of memories are used in embedded systems: program memory and storage memory.

Program memory

A program memory is used to store the application program during its execution. The memories used in most cases are RAM memories. They are fast, but volatile. Volatile means that they lose their information when the power supply is removed. Therefore the application code must be stored in another memory and be fetched to the program memory at the boot of the system. [25]

Storage memory

The storage memory carries the application code of the system and is a non-volatile memory which allows the data to be stored even when the system is unpowered. The memories used are often a ROM or flash memory. These memories are slower than RAM memories, but they are usually cheaper and have larger storage capacity. [25]

There can sometimes be more than one storage memory in an embedded system. An Electrically Erasable Read Only Memory (EEPROM) is a kind of flash memory that is often used in embedded systems to store smaller amounts of data. [25]

Virtual memory

To give the programmer easy access to different parts of the system a virtual memory is created. The virtual memory lists the addresses where memories and also peripherals can be accessed. An example structure can be seen in Table 2.1. The addresses on the virtual memory does not need to correspond with the physical addresses in the RAM. The translation from virtual to physical address is performed with a memory control unit (MMU) in the CPU. This unit sets the right control signals to accesses the correct part of the system. [25]

The size of the virtual memory is often set to the size of the address bus. This means that the virtual memory can address far more addresses than the RAM has physically place for. [7]

2.2.3 Peripherals

Peripherals are hardware modules that are located around a processor core. The purpose of the modules is to perform auxiliary tasks such as handling I/O, communication or backing storage. [20] In the case of embedded systems a broader definition has been chosen. Peripherals are in addition to the previous definition also hardware that is located around the CPU. This means any kind of circuitry that adds functionality to the system. It can be things such as analog/digital-converters, sensors, power regulators or Light Emitting Diodes (LED). Here are hardware used to control bus-traffic also accounted.
<table>
<thead>
<tr>
<th>Module</th>
<th>Address range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused</td>
<td>0x5300-0xFFFF</td>
</tr>
<tr>
<td>EEPROM</td>
<td>0x5200-52FF</td>
</tr>
<tr>
<td>SPI</td>
<td>0x5100-0x51FF</td>
</tr>
<tr>
<td>Sensor</td>
<td>0x5000-0x50FF</td>
</tr>
<tr>
<td>Unused</td>
<td>0x4101-0x4FFF</td>
</tr>
<tr>
<td>LED</td>
<td>0x4100</td>
</tr>
<tr>
<td>Serial Port</td>
<td>0x4000-0x40FF</td>
</tr>
<tr>
<td>FLASH</td>
<td>0x2000-0x3FFF</td>
</tr>
<tr>
<td>RAM</td>
<td>0x0000-0x1FFF</td>
</tr>
</tbody>
</table>

Table 2.1: An example of a 16-bit virtual memory. Compare with Figure 2.1.

2.3 Communication

In order for the system to work it must not only be able to communicate with other parts of the system, but also with external components. To do this buses are used. There is a wide variety of buses. Internal buses can be used without a specific bus protocol. For external communication more complex protocols such as SPI, UART or CAN can be used.

2.4 Embedded application code

A common programming language used to write embedded application code is the C-language. In order to make it able to run on a CPU it must be cross-compiled to machine code and downloaded to the physical platform using for example a JTAG. [25]

At start up the program counter starts at a predefined place in the memory called the reset vector. This is also the place where the program counter would go to if a reset button would be pushed. [16] The reset vector usually points at a boot loader program in the memory which sets up the initial state or a operating system if such is used. When this is done the actual application program can start.

Since many events in an embedded system are asynchronous, such as bus transactions and I/O, embedded systems have a frequent use of interrupts. Often so frequent that many processors have a special interrupt unit to generate and control different interrupts. [20]

Debugging and testing of embedded application code is different from regular GPPs. It is hard to monitor internal states on an embedded system and the debugging capabilities are more limited than those that an IDE for regular development offers. A Joint Test Action Group tool (JTAG) can offer simple debugging features [25], but debugging and testing are often complicated and time-consuming parts in embedded system development.
This chapter aims to give a general overview of virtual platforms. This includes explanations of what they are and why they are used. With this are also common problems and benefits that are associated with virtual platforms mentioned. In the end of the chapter are also a number of development tools for virtual platforms presented.

3.1 What is a virtual platform?

To realize what a virtual platform is, one just need to consult a dictionary. The Oxford dictionary defines the word virtual as "not physically existing as such but made by software to appear to do so". [3] This is a rather precise description of what a virtual platform is. It is a platform that is implemented in software and a platform in this case means a combination between hardware and software.

In other words, a virtual platform is a complete simulation of a computer system. All hardware is abstracted to a software model that can interact with the compiled application code. Optimally, the platform should be modelled so that the binary application can see no difference between running on the virtual platform or on the physical.

In some way a virtual platform can be seen as an extended Instruction Set Simulator (ISS) which not only simulates the CPU, but in addition the complete system. The modelling of virtual platforms is by some defined as electrical system level modelling (ESL). This level can be placed somewhere between RTL and a functional level. The use of this level is motivated when the cycle true nature of RTL is regarded as too detailed, but when still some hardware accuracy is required.
3.2 Reasons for usage of virtual platforms

The hardware industry has under a long time used simulation tools to verify the functionality of complex hardware circuits before they enter production. In the field of embedded systems the common practice has been different, where a hardware prototype is usually developed first. When this is done, it is handed over to the software engineers as a reference model. They start to develop the application code for the system and use the prototype for testing. When the hardware design is completed, the software and hardware are finally integrated together and verification testing of the functionality starts. This method can be called consecutive development, since the software development waits for the completion of the hardware development and the testing waits for the completion of both hardware and software development. An illustration of this development method can be seen in Figure 3.1a.

With the increased complexity among embedded systems this development method has become very fragile. Since all parts rely on each other it is possible that even a tiny delay early in the chain can become a large bottleneck for the others. It can even go so far that software engineers are forced to start the development without any hardware prototype, which in turn puts extra demands on the testing.

Virtual platforms have been prosed as a solution to these problems. The problem with dependency on an early hardware model would disappear if the embedded industry started to simulate their products just as the hardware industry has done for years.

3.3 The purpose of a virtual platform

In general the goals of virtual platforms are to achieve the following.

- Enable early and effective software development.
- Minimize the hardware dependency.
- Increase the testing efficiency.

3.3.1 Enable early and effective software development

Concurrent development

As mentioned earlier, embedded software developers often have to wait for a hardware prototype. This results in a bottlenecked development where steps have to be finished in a consecutive manner like in Figure 3.1a.
If a virtual platform is implemented early in the development software developers can start working even before a hardware prototype is available. The developers could then use the virtual platform as reference instead of the real hardware. The virtual platform is also useful for the hardware designers since they can make and test small changes in the hardware without having to do that on the physical platform.

The availability of a virtual platform would loosen the dependencies between hardware design, software development and test/verification. It is also possible that some parts of the different design steps could merge with each other. For example would software development with good testing in a virtual platform potentially result in less problems in the end phase of the development. This design methodology can be called concurrent design and is described in Figure 3.1b. The main goal of concurrent design is to decrease the total development time and work effort.

Figure 3.1: A comparison between consecutive and concurrent development of embedded systems.
Development on a single platform

One problem with regular embedded software development is that the application is developed on a different platform than it is intended to be used on. This can be a problem in many ways. Figure 3.2 gives a simple description of how regular embedded systems are developed.

The development process in Figure 3.2 could be described with the following steps:

1. A specification of the wanted behaviour is made.
2. The application is implemented/modified in, for example, the C-language.
3. The source code is linked and compiled into an executable file.
4. The executable file is downloaded on the physical platform.
5. The executable file is executed on the physical platform, which results in a behaviour.
6. The behaviour can be observed through the I/O, physical measurements or some kind of debugging tool like a JTAG debugger.
7. The behaviour is compared with the specification and either the result is satisfying or the iteration is repeated from step 2.
With a virtual platform the physical system instead is modelled on the host computer, resulting in the development structure seen in Figure 3.3.

![Figure 3.3: Development with a virtual platform.](image)

The work flow in Figure 3.3 could then be described as:

1. A specification of the wanted behaviour is made.
2. The application is implemented/modified in for example the C-language.
3. The source code is linked and compiled into an executable file.
4. The executable file is executed on the virtual platform.
5. The behaviour of the executed code can be observed through a console, log file or a GUI on the host computer.
6. The behaviour is compared with the specification and either the result is satisfying or the iteration is repeated from step 2.

Virtual platforms have also been proposed to be used instead of functional models in early development. For example Ceng [9] advocates for this with the motivation that application code developed for a virtual platform is easier to reuse in later stages compared with code developed for a functional level platform.

### 3.3.2 Minimize the hardware dependency

The amount of available hardware is always a problem in embedded system development. There are often only a few hardware prototypes available which makes co-design a hard task. Since a virtual platform is only code it can easily be replicated to any number of ordinary computers. Co-design with a virtual platform is then much more simple and many developers can work on the same problem at the same time.
Many embedded systems are battery powered and the CPUs are often designed for low power consumption. The CPUs in embedded systems are in general much slower than GPPs. The frequency on a regular CPU for embedded systems can be around 100 MHz and the frequency on regular GPP can be around 2 GHz. Therefore, it is possible to simulate a system much faster with a virtual platform on a GPP than using the physical platform.

### 3.3.3 Increase testing efficiency

In general there are many things that restrict the test and verification steps in embedded system development. This can be things such as:

- Transfer of an executable file from the development platform to the physical platform can be time-consuming.
- Execution on the physical platform can be slow, making tests take long time to complete.
- Capability to monitor internal states and other information on the physical platform is limited.
- Testing can prove to be time-consuming in general and some test cases are hard to create.
- Much testing must be done manually and is harder to automate.

A virtual platform can ease some of these problems by adding extended debugging capabilities. Since the virtual platform runs in a software environment it is easy to generate interesting information and values on nearly every part of the system just as in a regular debugger. It would also provide faster and more controllable feedback than with a on-chip debug tool.

A software environment would also simplify the creation of automated tests, which potentially can provide a significant decrease of the manual work required and an increase of the test efficiency.

One other interesting area for virtual platforms is called fault injection. It has been investigated by, for example, Kim [18]. Since a virtual platform in software is more controllable than its physical counterpart, it is possible to create and verify uncommon or special test-cases. For example, Kim [18] uses a virtual platform to test the behaviour of an embedded system when faults such as write and read failures in a flash memory occur. Such errors can be necessary to test, but may prove hard to create on a hardware platform. It can require a long time of continuous execution before any such error occur.

### 3.4 Problems and limitations with virtual platforms

With the use of virtual platforms come the general problems with simulations. How can one be sure that the simulation corresponds with reality? Or better yet,
how much must the simulation correspond with reality? The well-known trade-off between simulation accuracy and simulation speed is a central aspect in the modelling of virtual platforms.

Sungpack [14] gives an interesting aspect of this problem with a term that they call hardware affinity. Hardware affinity is defined by them as the degree to which a model on a virtual platform corresponds with RTL-code. They point out that not all parts of a model must be modelled with great accuracy in order to achieve a highly accurate platform. Some parts can just have a dummy behaviour to prevent the application code from crashing and some parts must have higher accuracy and carefully take timing into consideration.

One other problem is that a virtual platform adds additional effort to the development. If a virtual platform is going to be beneficial, it must be worth the extra time it takes to implement it. The research around virtual platforms seems somewhat inconclusive as to whether virtual platforms give enough aid in the development phase. Hong [14] comes to the conclusion that the verification can not be assured unless it is co-simulated with RTL and that the virtual platform was available too late and with unsatisfying simulation speed. On the other hand, Heunhe Han [12] praises their virtual platform as an "excellent test environment which is very similar to a real H/W ARM platform". It seems that the results of a virtual platform can be highly dependent on the actual system and what the intended use of the virtual platform is.

3.5 Development tools for virtual platforms

There is a wide variety of development tools and that can be used for the implementation of virtual platforms. Some commonly used applications and development languages are listed in the following sections.

3.5.1 Development applications

This section mentions some common applications among virtual platform development.

Synopsys

Synopsys is an American electronic design automation (EDA) company that offers a series of commercial development tools for virtual platforms. [5]

QEMU

QEMU, short for Quick Emulator, is an open source application that can be used to model virtual platforms and also to run operative systems. [4]

OVP

OVP or Open Virtual Platforms is an open source virtual platform development tool that is maintained and also initially designed by the British company Im-
peras [2]. OVP is the tool that will be used to implement the platform in this thesis.

### 3.5.2 Development languages

This section mentions some commonly used languages used for implementation of virtual platforms.

**C**

The C-language is commonly used for development of embedded systems and many virtual platform development tools use C or C-like APIs. For example, OVP uses a C interface. [2]

**SystemC**

SystemC is a superset of C++ classes and also an IEEE standard that can be used to model and simulate systems on a cycle accurate level. SystemC is similar in its behaviour to HDL-languages like VHDL and Verilog, but is more focused on the modelling aspect than the correct hardware resemblance. SystemC is created and maintained by the Open SystemC Initiative (OSCI). [6]

**TLM 2.0**

TLM stands for Transaction Level Modelling and is a SystemC interface. The interface trades simulation speed for less accurate simulations and is more focused on data exchange and bus transactions than a cycle correct behaviour. The timing is described through two coding styles called loose and approximate timing. [6]

### 3.6 Open virtual platforms

OVP is the tool that is used for the implementation of the virtual platform in this project. OVP is, as mentioned before, an open source development tool for virtual platforms and it offers the following.

- An API for the development of platforms called Innovative CPU Manager (ICM).
- An API for the development of peripherals and behavioural models called Peripheral Programming Model (PPM).
- An API for the development of processor models called Virtual Machine Interface (VMI).
- A simulator for the simulation of the complete system called OVPsim.

OVP also offers a number of models, platforms and examples together with their source code. Documentation of the APIs and guides in platform modelling is also available. [2]

The choice to use OVP in this thesis depended on a number of factors. One of these factors was that OVP already offered a model of the processor used in this
platform. A processor model is in some sense a simple ISS and can take quite a time to implement by yourself. It is therefore of great benefit to have such model available for use. Other reasons for choosing OVP were that the APIs were well documented and that it was easy to set up an initial model to work from. No other development tools were tested in detail since OVP proved to fulfil the needs of the development quite well.
4
Introduction to the physical platform

This chapter gives a short introduction to the physical platform that will be used as a reference in this thesis.

4.1 Platform overview

The physical platform is an embedded system that is used as a CPU-board in an industrial control system. The CPU-board controls, through relays, the external machinery of the system. The processes are controlled by the board based on regularly received sensor data such as temperature and water flow. The system has real time demands, but these demands are rather slow, with processes that operate in the range of seconds.

4.2 Platform components

A simplified model of the platform can be seen in Figure 4.1. The board consists mainly of a micro-controller with attached working and storage memories. The board is able to communicate with other parts of the system via the use of a CAN-bus. Information from the CPU-board can also be transferred to a work station via the use of a serial port of RS-232 type. A three pin RS-232 configuration is used with just input, output and ground as used pins. The board also contains an Ethernet connection. This connection is however only used for diagnostic purposes and is not used when the product is running. Because of this it was decided not to include the Ethernet connection in the modelling of the virtual platform.
4.2.1 CPU

The CPU is a 32-bit RISC processor with an ARM core. It consists of a core, an internal bus and a number of control modules. The control modules are listed in Table 4.1 and control things such as data transfer, memory addressing and general settings on the CPU. The CPU has a 16-pin GPIO that can be configured to support both SPI and UART protocols.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SER</td>
<td>Serial data transfer control</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory control</td>
</tr>
<tr>
<td>GEN</td>
<td>General CPU control</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct memory access control</td>
</tr>
<tr>
<td>EFE</td>
<td>Ethernet control (Not used on this platform)</td>
</tr>
</tbody>
</table>

Table 4.1: CPU modules.

4.2.2 Memories

There are three memories on the board.

- A RAM memory used as a program memory for the CPU.
- A flash memory used as storage for the application program.
- A EEPROM memory used to store data from the CAN-bus communication.

4.2.3 Peripherals

The board has a number of peripherals.
• An RTC used to keep track of time.
• Eight LEDs for diagnostics.
• SPI to CAN circuitry.
• An RS-232 controller.
• An ethernet module (Not used here)
• Power and clock generation (Not of interest for the virtual platform)
• A JTAG connection (Not of interest for the virtual platform)

4.3 Application

How the application software is designed can be described with a simple layered structure of abstraction layers as in Figure 4.2. The three highest abstraction layers are included in the application software while the lowest layer is the platform itself.

The highest layer is the application layer. It contains the actual code used to control the product.

The layer under the application layer is the middleware layer. It provides TCP/IP control functionality and other support functions to the application.

On the deepest layer of the software is the OS/Driver layer located. It contains a real time operating system (RTOS) and device drivers.

At the bottom of the application stack is the hardware located. It is this layer that will be emulated in the virtual platform. This is the most important layer when it comes to simulation of virtual platforms. An understanding of the higher layers is however crucial in making a functioning virtual platform. For example is knowledge of how the drivers are designed important, since they specify what actions the software expect the hardware to provide.

4.4 External parts

The platform is able to communicate and control the external parts via the CAN-bus. The external parts consist mainly of a user interface (UI) and an input/output controller (I/O).
Figure 4.2: The application stack for the system.
This chapter describes how virtual platforms can be modelled and simulated using OVP.

5.1 Generation of a virtual platform

To generate and run a virtual platform in OVP, three components are used:

- A platform file.
- Application code that is intended to run on the platform.
- Peripheral files.

The two first components are enough to be able to simulate a processor and a memory with the application code running on them. However, if a more complex system is going to be simulated, a number of peripheral files that models other parts of the system must be included.

5.1.1 Platform file

The platform file is a C file with the OVP API included. With the use of the API the structure of the platform can be described. The general structure of a platform file can be described in the following way:

- Set and initialize simulation parameters.
- Create one or multiple processors.
- Create buses.
- Create memory.
• Create peripherals.
• Create nets.
• Connect the parts.
• Start the simulation.
• Terminate the simulation.

The different steps of the platform file will now be described. An example of a platform file can be studied in appendix A.1.

**Initialize parameters**

The first step is to set up the parameters for the simulation environment. This can be parameters such as which information the simulator should output and also if and how a debugger should be connected.

**Create one or multiple processors**

The second step is to create the processor. Here is the path to the processor model file specified. This file is then loaded at the start of simulation. Furthermore, parameters can be provided to the processor model such as processor variant, address size, endian type and more.

**Create buses**

When the processor is finished, a bus must be created. It is possible to create different bus structures, but in this case, a single 32-bit bus that carries both addresses and data is used.

**Create memories**

With the processor and bus modelled, one or several memories can be created. The size of the memories can be specified and connected to the memory map. Attributes such as write, read, and execute privileges can be specified. This means that both RAM and ROM memories can be created.

It is possible to specify into which memory the application code will be loaded at start-up. This means that no boot-loader that transfers the application code into the RAM needs to be simulated. If more complex actions need to be simulated on a memory access, callbacks can be added on these. Callbacks will be discussed more in-depth in Section 6.3.3.

**Create peripherals**

The last parts that are needed are the peripheral modules. The procedure here is the same as with the processor. A valid file path to the peripheral file should be provided and other parameter settings can be provided to it. Here is each peripheral unit mapped to a specific place in the memory map.
5.1 Generation of a virtual platform

Create nets

A net in the virtual platform can be seen as an ordinary wire in its physical counterpart. A net connects two parts of the system and can transmit data between them. Nets are for example used to perform interrupts.

Connect the parts

With all necessary parts of the system constructed, it is only left to connect them in the right manner. If not done earlier, connections to processor, memory, buses and peripherals can be provided.

Start the simulation

Finally is everything set up, loaded and ready to be simulated. Before the start of the simulation, it is possible to set stop time and other timing related parameters to the simulation.

Terminate the simulation

When the simulation is done, it must be terminated so the used resources can be given back to the host. Here can also the stop reason for the simulation be analysed. For example a message can be written in the simulator log if the simulation has ended due to an illegal memory access or some other error.

Compilation

When the structure of the platform is completed, the platform code can be compiled with a compatible C compiler, for example gcc. The compiler does then produce one platform executable, platform.exe, and one shared library, platform.dll.

5.1.2 Application code

The application code is the software desired to run on the platform. In this case the unmodified elf file that runs on the real platform is used. For the discussion in Section 6.5.1 it can be worth to note that this application uses a big endian format.

5.1.3 Peripheral files

Peripheral files are created by using the OVP API. The files needed are listed in Table 5.1. It is not necessary to do this in this way, but it is a good way to keep an effective structure of the peripheral model.

<table>
<thead>
<tr>
<th>File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pse_attr.c</td>
<td>Specifies connections and parameters</td>
</tr>
<tr>
<td>pse.c</td>
<td>Defines content of peripherals</td>
</tr>
<tr>
<td>user.c</td>
<td>Defines behaviour of peripheral</td>
</tr>
</tbody>
</table>

Table 5.1: Peripheral file structure.
Attribute file

The attribute file specifies all connections and parameters related with the peripheral device. This can be specifications of connections to the bus port and which nets that are connected to the device. Here can also parameters such as baud rates and output files be connected with the peripheral.

PSE file

The PSE file describes the content of the peripheral. In the file registers can be added in the memory map of the peripheral. To these registers can also specific callbacks be associated. Furthermore, net connections to other peripherals can be opened. The PSE file should also contain initialization functions and the peripherals main function.

User file

The user file specifies the behaviour of the peripheral. Here is the code for callback functions written together with additional functions that models the behaviour of the peripheral. If the peripheral needs to simulate a specific device, such as an EEPROM, externally created libraries can also be included.

Compilation

The peripheral files are compiled using a special cross compiler provided with the OVP package. The compiler outputs a so called PSE file which in essence is an ELF format executable [15]. More info about the properties of PSE files will be given in Section 5.2.

5.2 Simulation

The simulation of the virtual platform is done in OVPsim (Open virtual platform simulator) which is a simulator that comes with the OVP installation. To be able to run the simulation, the simulator requires the files generated from the platform code and the files generated from the peripheral code.

The simulation starts with loading the processor model and peripherals models specified in the platform.c file. When the initiation is complete, the simulator starts to execute at the start vector in the application code.

The simulation runs on the host processor, but runs in a private space isolated from the rest of the host environment. Each peripheral unit runs on a PSE (Peripheral simulation engine), which in essence is a simple virtual machine designed to run the peripheral code. The PSE engine provides an isolated and threaded environment that separates it from the rest of the simulated object, such as the processor, memories, buses, and other peripheral units [15]. Since the peripherals run in parallel in different environments, it is possible to simulate the concurrent behaviour of hardware.

The PSE engine has limited simulation capabilities due to its restricted access to
the host environment. This means that the PSE is only capable to use normal C constructions and some libc functions. If extended functionality is needed, such as the use of a graphical library for a GUI, an interception library must be created. This library is then used to catch up certain function calls and executes them outside the simulation environment instead. [15]

The simulator can also use features such as semihosting, which means that some actions that are meant to be executed on the system are executed on the host instead. This allows for example print outs to be written in the simulator log instead of over an UART. This could potentially allow a simulation to speed up significantly. [19]

This feature is not used in the implementation of this virtual platform because the application code uses a quite old version of the standard C library. This causes incompatibility between it and the C library that the simulator uses.

Figure 5.1 gives an example of a simulation case. The virtual platform runs in the OVPsim simulator on the host computer. Inside the simulation runs the processor model in one environment and five peripheral units run in one PSE environment each. The different environments can be accessed via the memory map where each peripheral unit has been given an area. It is also possible to access another environment via a net, if one is present. Outside the simulation environment is the terminal that outputs the simulation log, interception/semihosting libraries, and a debugger located. The debugger used in this case is a GNU debugger (GDB). The debugger will be discussed later in Section 5.3.

5.2.1 The concept of time

OVPsim is an instruction accurate simulator. This means that it does not have the same precision of time as a cycle accurate simulator would have. The simulator has however some capabilities to simulate time.

The simulator uses two concepts in its representation of time:

- **MIPS** (Millions of instructions per second)
- **Time slice**

Every processor that is simulated has a MIPS value, $N_{MIPS}$, associated with it. This is the number of million instructions that it is capable to execute during a second. This value can be specified as a parameter in the platform.c file.

The time slice, $T_s$, specifies how long each simulation step shall be. OVPsim uses a just in time (JIT) compiler that compiles the time slice during runtime [19]. The longer the time slice is, the faster the simulation becomes. However, a long time slice results in lower timing accuracy. The time slice should in general be shorter than the shortest simulated delay in the system.

The two values are connected with each other in the way that:
$N_{\text{instructions}} = T_s \cdot N_{\text{MIPS}}$

For example would a time slice of 0.001s and a MIPS of 100 result in that 100 000 instructions should be executed during each time slice.

The simulation algorithm then works in the way that the processor model runs on and executes instructions from the application code. When the processor model has executed enough instructions, the time is increased according to the time slice. When an access is made to the virtual platform from the application code, the simulation time is stopped. It remains still until the virtual platform code has finished or a wait call is made in the virtual platform code. [19]

An illustration of the simulation procedure and time flow can be seen in Figure 5.2 where a write to a FIFO in a peripheral model makes the simulation stop to execute virtual platform code. Later, a hardware timer is started that runs for a while until the simulated time has elapsed so long that a time out event occurs in the virtual platform code and a time out function is executed.

How fast the simulation time proceeds in comparison with the wall clock time depends on the performance of the host processor and how much virtual platform code that is executed. For example will a virtual platform with many print outs to the simulator log result in significantly slower simulation speed than without.
5.3 Debugging

To ease the verification process of the virtual platform, a debugger can be attached to the simulation via a remote serial protocol (RSP). The debugger used in this case is GDB and is attached to the application code. This enables a wide variety of debugging features such as breakpoints, single stepping, and backtracking.

The virtual platform code was debugged with simple print outs in the simulator log, since the complexity of the code never grew so large that a real debugger was needed.

**Figure 5.2: Simulation time flow.**
This chapter gives a description of the implementation of the virtual platform. The first sections will give an overview of the platform and later sections will describe the different parts of the platform.

6.1 Overview of the virtual platform

The virtual platform consists of a number of files that run in the OVPsim simulator. A figure of the simulation environment can be seen in Figure 6.1. The files used are:

- Platform executable (platform.exe)
- Platform shared library (platform.dll)
- Application executable (application.elf)
- Processor model (ARM.dll)
- Peripheral simulation units (ser.pse, mem.pse, efe.pse, dma.pse, and gen.pse)

In Figure 6.2 the contents of the virtual platform model are illustrated. It consists of the board described in Chapter 4 and a block that generates responses on the CAN bus. In Table 6.1 all hardware models are listed. The platform can be compared with the physical board in Figure 4.1 in Chapter 4.
Figure 6.1: Overview of the simulation environment.

Figure 6.2: The virtual platform and its environment.
6.1 Overview of the virtual platform

<table>
<thead>
<tr>
<th>Hardware functionality</th>
<th>Model file</th>
<th>Included in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor core</td>
<td>-</td>
<td>ARM.dll</td>
</tr>
<tr>
<td>RAM</td>
<td>-</td>
<td>platform.dll</td>
</tr>
<tr>
<td>Flash</td>
<td>-</td>
<td>platform.dll</td>
</tr>
<tr>
<td>SPI</td>
<td>-</td>
<td>ser.pse</td>
</tr>
<tr>
<td>EEPROM</td>
<td>eeprom.h</td>
<td>ser.pse</td>
</tr>
<tr>
<td>RTC</td>
<td>rtc.h</td>
<td>ser.pse</td>
</tr>
<tr>
<td>CAN</td>
<td>can.h</td>
<td>ser.pse</td>
</tr>
<tr>
<td>RS232</td>
<td>rs232.h</td>
<td>ser.pse</td>
</tr>
<tr>
<td>LED</td>
<td>led.h</td>
<td>gen.pse</td>
</tr>
<tr>
<td>GPIO</td>
<td>-</td>
<td>gen.pse</td>
</tr>
<tr>
<td>Interrupt control</td>
<td>-</td>
<td>gen.pse</td>
</tr>
<tr>
<td>Timers</td>
<td>-</td>
<td>gen.pse</td>
</tr>
<tr>
<td>External UI-controller</td>
<td>external_sim.h</td>
<td>ser.pse</td>
</tr>
<tr>
<td>External IO-controller</td>
<td>external_sim.h</td>
<td>ser.pse</td>
</tr>
</tbody>
</table>

Table 6.1: Hardware models and their simulation location.

6.1.1 Toolchain

For the development of the virtual platform the simple toolchain in Figure 6.3 is used. It consists of Make and GNU compiler collection (GCC) for compilation of the virtual platform source code and OVPsim and GDB for simulation and debugging.

![Figure 6.3: The virtual platform toolchain.](image)

6.1.2 Memory map

The CPU has a memory map according to Table 6.2 where it can access RAM, flash, and the peripheral devices.
<table>
<thead>
<tr>
<th>Block</th>
<th>Address range</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmapped</td>
<td>0xFFE00000-0xFFFFFFFF</td>
<td>-</td>
</tr>
<tr>
<td>SER</td>
<td>0xFFD00000-0xFFFFDFFFFF</td>
<td>1 MB</td>
</tr>
<tr>
<td>MEM</td>
<td>0xFFC00000-0xFFFFCFFFFFF</td>
<td>1 MB</td>
</tr>
<tr>
<td>GEN</td>
<td>0xFFB00000-0xFFFFBFFFFF</td>
<td>1 MB</td>
</tr>
<tr>
<td>Unmapped</td>
<td>0xFFA00000-0xFFFFAFFFFFF</td>
<td>-</td>
</tr>
<tr>
<td>DMA</td>
<td>0xFF900000-0xFFFF9FFFFF</td>
<td>1 MB</td>
</tr>
<tr>
<td>EFE</td>
<td>0xFF800000-0xFFFF8FFFFF</td>
<td>1 MB</td>
</tr>
<tr>
<td>Unmapped</td>
<td>0x03000000-0xFFFF7FFFFF</td>
<td>-</td>
</tr>
<tr>
<td>FLASH</td>
<td>0x02000000-0x02FFFFF</td>
<td>16 MB</td>
</tr>
<tr>
<td>RAM</td>
<td>0x00000000-0x01FFFFF</td>
<td>32 MB</td>
</tr>
</tbody>
</table>

Table 6.2: Virtual memory map.

### 6.1.3 Terminology

This section will give explanations of some words that will be used frequently in the following sections.

**Peripheral:** The use of the word peripheral can sometimes be confusing. The meaning of the word can be different depending on what someone means that the thing is a peripheral part of.

To avoid confusion when talking about peripherals in the virtual platform, the following definitions are used:

**Board-peripheral:** A board-peripheral is the virtual platform counterpart of a peripheral that is a piece of hardware located on the board. This can be hardware such as LEDs, memories, and bus devices.

**CPU-peripheral:** A CPU-peripheral is the virtual platform counterpart of a peripheral that is located in the CPU. These peripherals give extra hardware features to a CPU in addition of the core. This can be things such as support for a certain bus protocols or interrupt control. They will sometimes also be referred to as units.

### 6.2 Modelling methodology

The goal of the virtual platform is to mimic the behaviour of the complete system. To gain this functionality a methodology depicted in Figure 6.4 has been used. This methodology was something that developed quite naturally, since the complete application code was already available from start.

The first steps have been to study the datasheets of the selected hardware and to study device drivers in the application code. With this can some information how the hardware behaves and how the programmer expects it to behave be gained.
The next step is to set up something that Sunpack [14] defined as a dummy structure. This means that a platform that can handle all accesses performed by the application code, but has no further behaviour. This allows the application code to start up.

When the application code is able to start, the next step is to connect the simulation to a debugger and analyse when the code makes a hardware accesses. With the knowledge gained from studying datasheets and drivers, the virtual platform is modified to be able to handle these accesses in an acceptable manner. After this the platform is simulated again and new hardware accesses is found and the process is repeated. This is repeated until the application is capable to initialize and run without crashing.

When the application code can run without any major flaws, the platform is analysed to see if some extra behaviour is wanted/needed. For example the simulation of delays in devices or other refinements of the platform. This process is also repeated until the functionality of the platform is satisfying.

This is a top-down method which has the benefit of being relatively fast to get a working platform. By following this method it is also often not necessary to simulate all hardware behaviour to get something that works.

The drawback is that without having a 100% code coverage of the application code it is hard to know if all needed hardware functionality is implemented and working.

### 6.3 Modelling tools

Both the C language and the OVP API offer many tools to model hardware in an effective way. The most used tools in this virtual platform are:

- Registers
- Nets
- Callbacks
- State machines
- Timers

#### 6.3.1 Registers

The register works as a simple data storage. However, the memory mapped registers in the CPU does often work as a front end for programmers to control underlying hardware. Therefore the registers must work in the way that they not only store data, but also so that functions are executed when they are accessed. To enable this functionality, callbacks are used.
6.3.2 Nets

A net in the virtual platform can be seen as an ordinary wire in its physical counterpart. A net connects two parts of the system and can transmit data between them. Nets make it possible to communicate with other parts of the system without accessing the memory map. The biggest difference between a virtual net and a physical one is that the virtual net can carry more values than just 1 and 0.

A net connection can just like a register in the memory map, carry a callback that is executed when a net is written to.

The usage of nets in the virtual platform is much like the usage of nets in a physical one. For example there is a net between the processor core and the interrupt controller which is used to generate interrupts.

6.3.3 Callbacks

A callback is a function that is called upon when a special event occurs. In the case with the virtual platform a callback is a function that is called upon when a write or read is performed at a memory address. A callback can also be executed
when a net is accessed.

An example of a callback action is showed in Figure 6.5, where a piece of application code executes a function that writes a character to the serial transmission FIFO. This write to the memory causes a callback function associated with this memory address to be executed. This function retrieves the value written to the memory and makes it appears in a terminal window.

Callback functions are an essential part of a virtual platform since they become a link between the application code and the simulation code.

![Figure 6.5: Illustration of a callback function.](image)

### 6.3.4 State machines

State machines are used to keep track of which state a device is in and what actions it should take next. State machines are used on devices that can be accessed in more than one way and that usually perform more than one action. An example of this are the SPI devices which often handle the communication over the bus via a specific protocol.

### 6.3.5 Timers

Timers are simulated as threads that run in parallel. The timers use the simulation time as reference and have no relation to the wall clock time. When a timer reaches its time-out it can be designed to perform a function that simulates some effect. A timer can be used single shot, which means that it is run once and then deleted. This is usually the case when a device delay is simulated. A timer can also be used in a looping manner where the timer is restarted once it is finished. This is the case with timers used for regular reports and the hardware timer.
6.4 Processor core

The processor core used in this virtual platform is the ARM processor model provided by the OVP package; no modification to the model has been made.

6.5 CPU-peripherals

Each CPU-peripheral have been given a part of the memory map where special registers can be accessed. Each CPU-peripheral runs in a separate PSE engine which enables concurrent execution which mimics real hardware.

An illustration of a CPU-peripheral can be seen in Figure 6.6. The figure shows how the CPU-peripheral is mapped to the memory map and how connections are done internally.

6.5.1 Endianness correction

The endian of an application defines in which order the bytes in a word should be stored. In a big endian format, the most significant byte is stored at the lowest address and in a little endian format it is instead stored at the highest location. Table 6.3 illustrates how the four byte word 0xaabbccdd can be interpreted in different ways depending on the endianness of the application.
Table 6.3: Big endian versus little endian.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Big endian</th>
<th>Little endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0xaa</td>
<td>MSB</td>
<td>LSB</td>
</tr>
<tr>
<td>0x1</td>
<td>0xbb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2</td>
<td>0xcc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3</td>
<td>0xdd</td>
<td>LSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>
| Result  | 0xaabbccdd | 0xddccbbaa

There is a problem in this case. The application code uses a big endian format, but all the peripheral units runs on the PSE engine that uses a little endian format. This means that data the application code writes to the memory will be interpreted incorrectly by the peripheral units and vice versa. The solution to this problem is that peripheral units must swap the byte order of the data before it reads or writes to the memory.

In Figure 6.7 the communication between the different devices and the memory is clarified.

Figure 6.7: Solution of the endian problem.

6.5.2 GEN

The GENeral CPU-peripheral (GEN) provides a number of registers that can control a wide variety of things on the system. They provide system control/status registers, setting of the GPIO ports, interrupt control and control of hardware timers.

Implementation

The main functionality of the GEN CPU-peripheral is interrupt control and timer generation. It also controls the GPIO. The GPIO contains chip selects, interrupt
request pins associated with the LED controller. The GEN unit also stores some values associated with the system, such as identification id and other general settings.

**GPIO**

The GEN unit handles the GPIO accesses. The GPIO is used to access the LED model, simulate incoming interrupt requests from the CAN model, and simulate the chip select pins for the SPI bus.

**Interrupt controller**

The processor model uses the regular ARM interrupts FIQ and IRQ. These are two nets that are directly connected to the processor core. The fast interrupt (FIQ) is not used in this virtual platform and therefore only IRQ is used. Since there are multiple interrupt sources in the system, an interrupt controller is used to manage the interruption process. The interrupt controller in the processor is a very simple one. It consists of three registers in the GEN CPU-peripheral:

- Enable register
- Status register
- Raw register

By writing to the enable register, specific interrupt sources can be enabled or disabled.

By reading from the status register, the status of all enabled interrupt sources can be seen.

By reading from the raw register, the status of all interrupt sources can be seen regardless if they are enabled or not.

Each bit in the registers is associated with a specific interrupt source. If some part of the system wants to initiate an interrupt, it must set the corresponding bit in the raw register. The interrupt controller then performs a bitwise AND operation on the raw and enable registers. If this operation results in a bit being set, it means that an interrupt request has been acknowledged. The interrupt controller then sets the IRQ net high.

With the IRQ net high, an interrupt function that is located in the application code will be executed. This function determines from which source the interrupt request came from and then starts the correct interrupt function associated with this interrupt request. While the interrupt source is determined, the raw bit is cleared and if no other interrupt requests remain, the interrupt will set the IRQ net to low again.

**Timers**

One hardware timer is used in the virtual platform. It is used by the OS to keep track of time. It generates an interrupt once every 100ms which is signalled to the OS so it can keep track of time.
The GEN unit also contains a number of other timers that is not connected with hardware. They are used to simulate delay and periodic reports in the virtual platform.

### 6.5.3 SER

The SERial controller CPU-peripheral (SER) provides hardware support of a number of communication protocols such as UART and SPI. The registers in the SER unit can control parameters such as transmission speeds and also provide transmission support with, for example, FIFOs that hides the underlying signal generation.

**Implementation**

The SER CPU-peripheral is the largest CPU-peripheral on the virtual platform since it takes care of nearly all external bus communication. The SPI bus is modelled in this CPU-peripheral and all models that communicate using SPI are also included here. This CPU-peripheral also opens a port to a terminal for connection with the serial RS232 controller, which is also included in this CPU-peripheral.

### 6.5.4 MEM

The MEMory controller CPU-peripheral (MEM) is used to provide an interface to an external memory. An example of this would be generating the correct control signals to a RAM or flash memory.

**Implementation**

Since OVP allows for the creation of a simple memory map, the RAM memory does not need to be simulated with control signals. Instead, RAM can easily be accessed via the memory map. Due to this reason, the CPU-peripheral only acts as a dummy structure that can provide read and write accesses to the register to prevent any faults in the application code.

### 6.5.5 EFE

The Ethernet Front End CPU-peripheral (EFE) contains a number of registers that can provide a front end interface for Ethernet communication and control a media access controller (MAC). The Ethernet communication can be performed via an external Media Independent Interface (MII).

**Implementation**

Since it was decided not to implement the Ethernet connection, the EFE unit only provides basic functionality to allow the application program to run without crashing. The application needs the external MII device to produce a product id associated with it. This functionality is added to the EFE unit, all other registers are only implemented to handle read and write accesses without any additional functionality.
6.5.6 DMA

The Direct Memory Access CPU peripheral (DMA) allows for a direct transfer of data between the memory and peripheral or memory and memory without the need of the CPU.

Implementation

DMA functionality has not been implemented, since the application only uses the DMA to transmit Ethernet data. Only basic read and write access to the DMA registers have been implemented, but without any functionality.

6.6 Board-peripherals

Most board peripherals are included as a library in a CPU peripheral, but some board peripherals, such as the SPI bus, are so tightly tied to some registers in the memory map that they are integrated in the SER unit instead. An included library gives access to the contents of the board peripheral. In general the library consists of a structure that holds things such as state machines and registers. A number of functions can then be used by the CPU peripheral to access and effect the board peripheral. Appendix A.2 and A.3 give an example of how a board peripheral library is designed.

6.6.1 RAM

The RAM is modelled using the standard RAM model available from the OVP API. See Section 5.1.1.

Timing

The RAM model does not take timing into consideration and fetches instructions as fast as possible. It is however possible to simulate some timing effects by altering the MIPS value on the processor core. A lower value on the MIPS would illustrate the delay that occurs when instructions are fetched from the RAM.

6.6.2 Flash

The flash is accessed via the memory map just like a regular RAM access. The problem with the flash is that it is not accessed in the same way as a RAM. The flash has an internal state machine that can be controlled by writing specific data to certain addresses. The memory map can therefore not be used as data storage since data will be overwritten when the flash driver tries to access the flash memory. The virtual model uses therefore the structure seen in Figure 6.8 instead.

The model simulates the flash by using an array for storage of the flash data. The array is protected from illegal accesses behind the state machine. It also mimics the real flash by having different block sizes. When the simulation starts, the array is blank and does not contain any special information.
When a read or write is done at any place in the flash area of the memory map, a read/write callback will be executed. Depending in which state the flash state machine is, it can provide ordinary ROM access, program a word, erase a memory block, or output device information. The device information is needed by the application code to identify the device.

The state machine can be seen in Figure 6.9. The small circles are substates. The state machine will return to its default state if it is in a substate and a write that is not specified in the state graph occurs. This is not illustrated in the figure.

**Timing**

According to the data sheet, the flash can perform a read in around 10-100ns and a write in 1-10ms [26].

This is a significant number, but no delay has been added to the virtual model. The application code performs a 10 ms second wait call after writing to the memory, thus removing the need for any delay simulation.

![Figure 6.8: The structure of the flash model.](image)

### 6.6.3 LED

The LED diodes serve as a basic status display on the platform. The LEDs are controlled via two pins (data and SC) at the GPIO port through a serial to parallel shift register.

The LED model is illustrated in Figure 6.10. The LED library is implemented as a structure carrying the state of the LED array. The LED register is updated by pulsing the SC (Serial clock) pin eight times, once for each LED, and during each pulse write the wished state of the LED with the data pin from the GPIO port. When the full LED array is updated, the new state is written in the simulation log.
6.6.4 RS232

The purpose of the RS232 model is to provide a serial communication link between the CPU and a terminal. From the terminal a user can issue commands and get diagnostic information from the system. An illustration of the virtual model can be seen in Figure 6.11.

There are two buffers: one send buffer and one receive-buffer. The send-buffer is located in the application code and is polled every 100 ms for any new data. If this is the case, it writes to the serial control register to generate an interrupt that starts a transmission over the channel. The RS232 controller receives this transmission and writes it in the terminal.

The receive-buffer works in nearly the same way but is emulated on the virtual platform instead. When text is written in the terminal it is placed in the receive buffer. The content of the buffer is checked every 100 ms with a timer and if new data has been placed here, a receive interrupt is executed and the data is sent to the application.
6.6 Board-peripherals

Timing

The communication over the serial port is asynchronous and only happens when someone types text. This means that transmissions over the channel happen very seldom in comparison with other events. Because of this reason, no timing aspects such as delay over the transmission line have been added to the model.

Figure 6.11: Illustration of the RS232 model.

6.6.5 SPI

The SPI model simulates the data transmission between the CPU and the devices connected on the SPI bus. An illustration of the model can be seen in Figure 6.12. The models have two purposes: to simulate the transmission delay on the SPI bus and to act as an SPI master. This means to select the right device for transmission/reception of the message.

Chip selector

Selection of which SPI device that is supposed to be used is done by analysing which chip select that is low on the GPIO port. The chip selector then selects the right device. If a chip select goes high during transmission, the chip selector informs the SPI slave device that the communication has ended. The slave device then returns to its standby state.

Timing

The delay on the SPI bus is simulated by setting a busy bit in the serial control register and starting a timer that clears the bit on time-out. See Section 7.4.1 for a theoretical calculation of the delay on the bus.

6.6.6 EEPROM

The EEPROM serves as a back-up memory for system related data on the platform. The virtual model is illustrated in Figure 6.13 and consists of an SPI interface, a state machine, and the memory itself with a status register.
SPI interface

The SPI interface handles the SPI communication between the EEPROM and the CPU. It decodes the messages from the CPU and controls the state machine.

State machine

The EEPROM state machine decides which actions should be performed. The state machine can be seen in Figure 6.14 and it supports write and read actions to both the memory area and the status register.

Memory

The EEPROM is supposed to contain settings about the system that is accessed by the application program at start-up. The EEPROM memory array therefore contains some hard-coded values.

Timing

The data sheet specifies that a write cycle can take long time (in the range of milliseconds) [22]. A write cycle can write a total of 32 bytes to the memory. The delay of the write cycle is simulated by setting a write-in-progress bit in the EEPROM status register. After this is timer started that will clear the bit on time-out.
6.6 Board-peripherals

**Figure 6.13:** Illustration of the EEPROM model.

**Figure 6.14:** State machine for the EEPROM model.

### 6.6.7 RTC

The Real time clock is a component that is used to keep track of the current wall clock time. The RTC contains a number of time registers that stores the current time and date. It also contains a 31 byte RAM for storage of general data.
Communication with the RTC can be done with an SPI protocol.

An illustration of the virtual model can be seen in Figure 6.15. It consists of an SPI control interface, a state machine, a time updating block and storage registers.

Figure 6.15: Illustration of the RTC model.

**SPI interface**

The SPI interface handles the communication with the CPU. The received data is decoded to a command that is sent to the state machine.

**State machine**

The state machine keeps track of which state the RTC is in and handles the execution of the correct commands. The RTC state machine can perform actions in three areas.

- Time actions
- RAM actions
- Magic identifier actions

The time actions read and write to the time registers. These are accessed by a burst read/write that outputs all registers in a sequence except for the century register. The century register must be read/written with a separate command.

The RAM register can be accessed using either a burst mode or a regular access mode.

The magic identifier is a value that can be used for identification purposes.

The state machine can be seen in Figure 6.16. The state machine for the time registers (Figure 6.17) and the RAM registers (6.18) have been illustrated in two separate figures due to their complexity.
Figure 6.16: State machine for RTC model.

Figure 6.17: State machine for the RTC time registers.
6. Implementation of the virtual platform

Time updater

In an ordinary RTC, the time is updated every second with the help of an oscillator. In a simulation it would not be necessary to update it so often. Therefore, the time in the virtual model is only updated when a request to read the RTC time registers is made. The time updater function utilises the time.h library to get the current system time. This is then coded into BCD format and stored into the time registers.

Timing

No timing has been implemented in this model since all actions can be performed during the time that it takes to access the SPI bus.

6.6.8 CAN

The CAN board-peripheral is a SPI/CAN converter. It takes data from the CPU which it transform into the CAN bus protocol. The structure of the model is described in Figure 6.19.

There are two SPI/CAN devices on the platform: one that communicates with a frequency transformer, and one device that performs communication with the external boards. In this implementation is only responses on the data communication bus generated. The frequency transformer is regarded as unconnected and does not generate any answers.
6.6 Board-peripherals

Figure 6.19: Illustration of the CAN model.

SPI Interface

The SPI interface receives and decodes commands and data sent via the SPI-bus.

State machine

The state machine can perform a number of actions on the CAN registers. There are the regular read and write commands, but there is also a command that directly reads the RX-buffer without the need of specifying the address. In addition to this, a special bit modify command can change parts of a register. The use of this command can be things like clearing interrupt bits and changing the operation mode. The content of the state machine can be seen in Figure 6.20.

Buffers and registers

The CAN device contains a number of registers that control the bus transmissions. The registers set things such as transmission speed, interrupt control, and operation mode settings.

In the CAN device there are four data buffers in total: two receive buffers and two transmit buffers. However the drivers in the application code only utilize one transmit buffer and one receive buffer.

Transceiver control

The transceiver control directs the data traffic on the CAN bus. It forwards transmitted messages to the response generation and sends an interrupt request to the processor when a message from the response generation reach the CAN model.

Delay

The transmission delay is simulated by setting a busy bit in the CAN transmission control register and starting a timer that on time-out clears the busy bit. This does not simulate the whole delay over the CAN bus, but it prevents the CPU from requesting a new transmission too early. The complete delay chain is controlled in the response generation.
6.6.9 Response generation

The response generation is implemented to enable responses on the CAN-bus. This is not a pure part of the virtual platform since it simulates things outside the main board. However, some kind of response on the CAN bus must be simulated to keep the application code from crashing.

The structure of this model is illustrated in 6.21. It consists of a transceiver controller that takes care of incoming and outgoing CAN messages, a model of a user interface and a model of a Input/Output unit.

A list of the messages sent over the bus can be seen in Table 6.4.

**Transceiver control**

The transceiver controller takes incoming CAN-messages, decodes them, and decides which unit to send the message to. Depending on the message it also determines if the CPU expects a response to it. If a response is to be generated, the controller starts a timer that simulates the response delay. When the time-out is reached, a function fills the rx buffer in the CAN model with the proper data. The function then signals to the CAN model that a message has arrived.

**UI controller**

The UI controller mainly simulates an LCD display, buttons, and LEDs. The text written to the display is outputted in the simulator log. Buttons and LEDs are modelled dynamically.
I/O controller

The I/O controller models an external I/O card. It contains information about relay states, digital input/output and sensor values.

Relays and digital I/O are modelled so that they can be turned on and off dynamically, but the sensor values are hard-coded. The sensor values are reported on a regular interval with the help of timers.

Timing

When a CAN message is transmitted and the response generation determines that the CPU expects answer, it starts a timer. When its time-out is reached, the RX-buffer is filled with the proper data. The timing involves the whole propagation back and forth including the device delay on the external board. A calculation of the theoretical value of the CAN bus propagation delay can be seen in Section 7.4.1.

<table>
<thead>
<tr>
<th>Messages</th>
<th>Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Settext</td>
<td>CPU</td>
</tr>
<tr>
<td>Button press</td>
<td>UI/IO</td>
</tr>
<tr>
<td>IO-state</td>
<td>IO</td>
</tr>
<tr>
<td>Sensor report</td>
<td>IO</td>
</tr>
<tr>
<td>Setrelay</td>
<td>CPU</td>
</tr>
<tr>
<td>Setleds</td>
<td>CPU</td>
</tr>
<tr>
<td>Alive</td>
<td>CPU</td>
</tr>
<tr>
<td>Isalive</td>
<td>UI/IO</td>
</tr>
</tbody>
</table>

Table 6.4: Messages sent over the CAN bus.
6.7 The complete system

With the presented parts connected together the full system can be simulated in OVPsim. As can be seen in this chapter, the amount of functionality in the different CPU-peripherals varies much. The SER and GEN units are parts that includes much functionality. It is also into these parts that the most modelling effort have been placed. Other units have nearly no functionality at all and exist only to prevent software faults.

The implementation of the virtual platform can be said to have progressed in two phases: pre-OS and post-OS.

In the pre-OS phase much effort was spent to generate enough platform functionality to boot the OS of the system. With a debugger the application code was analysed to see what kind of actions that were required to keep it running. It was sometimes hard to locate the source of some of the errors. When the system crashed it often depended on a chain of events that were not visible at the crash location. It could be things like an erroneous initial register value that caused the system to select a wrong parameter. This parameter would in that turn result in an erroneous checksum. The checksum would then be compared with the wrong value and the flash driver would then report a write error.

Not much focus was put on modelling in this phase since it was impossible to test cases that were not located in the initiation sequence of the code.

After the OS managed to start up, the controllability of the system increased. It was now easier to realize what the application code tried to do and it was easier to make different test cases that would access a specific hardware device. In general it can be said that once the base of the platform was created it was really easy to expand it with extra functionality. However some parts did still require some work effort. The parts that required the most attention were the following parts:

- The SPI protocols.
- The flash memory.
- The interrupt control.

The SPI communication protocols required much attention since many special cases needed to be covered and some of the state machines became quite complex. The most complex hardware device was the model of the flash memory. It required functionality to perform a large variety of operations such as: output manufacturer info, erase blocks of data and of course perform regular read and write operations. The transitions between the different operation states also required long write sequences, which caused complex a state machine.

The implementation of interrupts was also a non trivial action. The interrupt controller is a simple model, but the hard part was to generate the behaviour around it. This included things like setting and clearing bits in the hardware devices at the right time to generate a correct behaviour of the device.
This chapter gives a description of the comparative measurements performed on the virtual and physical platforms. The chapter begins with a description the metrics, the system parameters, and the measurement setup. Then the measurements from both the virtual and physical platforms are presented.

7.1 Overview

The goal of the measurements is to compare how well the virtual platform corresponds with the real one. The meaning of correspond well is of course always a matter of definition. The main purpose of the system is to collect and process data from external devices via the CAN bus. It is therefore of importance that the CAN bus traffic is simulated with some accuracy. Accuracy in this case means that the differences in timing between the virtual and physical platform are so small that events occurring on the physical platform can be replicated in the virtual platform without problems. In other words, data transactions happen in the right order compared with the physical platform. An event that takes 7 seconds on the physical platform should take the same time on the virtual platform. Data provided by the system such as CPU load should correspond with the values measured on the physical platform and so on.

The data traffic over the bus is performed in a number of steps, where the CAN messages are transformed into SPI messages and vice versa. Because of this reason several different delays must be simulated in the virtual platform. The measurements aim to:

- Verify if the sizes of these delays are valid or not
• Show if it is worth to simulate all delays in the chain
• Show which parameters in the simulation that effects the simulation most.

7.2 Measurement tools

In this section the different tools used for measurements on the virtual and physical platform are presented.

7.2.1 Physical platform

Traffic over the CAN bus was measured with a data logger tool. The tool could record the data sent over the bus and also put a timestamp when it received a transmission. Since every CAN message has a message id it was possible to decode the message type of every sent message.

7.2.2 Virtual platform

Data from the virtual platform was retrieved from the simulator log. The virtual platform outputs a log entry when the CAN model has received or transmitted a message. In all entries the message type and current simulation time are included.

7.3 Metrics

The metrics that will be used to compare the virtual and physical platform are presented in this section.

7.3.1 CPU load

The load of the CPU is measured with a thread that has the lowest priority in the OS. When no other actions are to be performed, this thread is executed and it increases a counter. The value of this counter is then used to determine the load. The higher the counter is, the lower the CPU load is. This counter value is then transformed into a percentage value of the maximum capacity. This value can be retrieved via the RS232 link.

7.3.2 Duration

The duration is the time that a specific defined series of transmissions take. This could for example be the time it takes for the system to start up or shut down.

7.3.3 Timing errors

The timing errors are determined by comparing one set of samples from the virtual platform with one set of samples from the physical platform. These samples are the logged CAN messages. They consist of a message identifier and a time
value of when the sample was measured. The set of samples from the virtual platform is denoted as $y$ and the set of samples from the physical platform is denoted as $x$. The start of the sample sequences is defined as the first messages sent on the CAN bus if the start-up of the system is analysed. If the shutdown is analysed instead, the first signals that orders a shutdown is used as the first sample.

**Drift error**

The drift error is defined as the difference between two corresponding samples in the virtual and physical sample set:

$$
\epsilon_{drift}(n) = |y_n - x_n| \ (s) \quad n = \{0, 1, \ldots, N\}
$$

This metric gives a value of how large the offset is between the virtual and physical samples.

**Absolute error**

The absolute error is defined as the difference between two corresponding samples in the virtual and physical set:

$$
\epsilon_{abs}(n) = |(y_{n+1} - y_n) - (x_{n+1} - x_n)| \ (s) \quad n = \{1, 2, \ldots, N - 1\}
$$

This metric is supposed to give another view of the timing by measuring the difference in time between two transmissions in the virtual and physical platform, thus removing the total elapsed time as a factor.

**7.3.4 Relative error**

The relative error can be seen as the percentage value of the absolute error. It is defined in this way:

$$
\epsilon_{rel}(n) = \frac{\epsilon_{abs}(n)}{x_{n+1} - x_n} \ (%) \quad n = \{1, 2, \ldots, N - 1\}
$$

This metric is supposed to give a view of how large the errors are on a relative scale.

**7.4 Parameters**

When a message is sent from the CPU to an external board the processor must first process the data that will be sent. This requires some time which produces a delay. After this, the message is sent via SPI to the CAN/SPI controller. This device performs a conversion and the message is then sent over the CAN bus. All of which cause delays. At last the message arrives at the external boards which
Measurements also require some time before they can generate potential responses. The delay sources have therefore been identified as:

- Processor delay
- SPI bus delay
- CAN/SPI controller delay
- CAN bus delay
- Delay on external card.

The transmission chain between the CPU and the external boards can be seen in Figure 7.1.

![Figure 7.1: Transmission chain for CAN messages.](image)

### 7.4.1 Theoretical bus transmission delays

The theoretical values of the delay on the SPI and CAN bus were calculated using information from some of the control registers in the system. These registers control the bitrate for the SPI and CAN transmissions. The calculated values on these delays will be used later in the report as reference values for some parameters in the virtual platform.

**SPI**

The CPU uses an external crystal oscillator for bit generation. This is a crystal with a frequency of 55 MHz. However, in the CPU there is a phase locked loop (PLL) that decreases this speed by a factor of five to 11 MHz. [7]

\[ f_{osc} = 11 \text{MHz} \]

There is a register in the SER unit that can be used to set the bitrate depending on its content (BRC). It follows this formula:

\[ f_{bitrate} = \frac{f_{osc}}{2(BRC + 1)} \]
In the system the value of $BRC$ is set to $BRC = 4$. This gives us:

$$f_{\text{bitrate}} = \frac{f_{\text{osc}}}{2(4 + 1)} = \frac{f_{\text{osc}}}{10} = \frac{11}{10} \text{MHz} = 1.1 \text{MHz}$$

This is the frequency of the clock on the SPI bus. Eight clock cycles are required to transmit one byte. This gives a byte frequency of:

$$f_{\text{byte}} = \frac{f_{\text{bitrate}}}{8} = 0.1375 \text{MHz} = 137.5 \text{KHz}$$

This finally gives us a transmission time of:

$$t_{\text{byte}} = \frac{1}{f_{\text{byte}}} = 7.2 \mu s/\text{byte}$$

This value is used as the nominal value on the SPI bus delay in the virtual platform measurements later in the thesis.

**CAN**

This calculation assumes that we have a relatively friendly CAN bus environment, which means that messages can be sent without retransmissions or with some kind of noise that corrupts messages. This environment is generally the case, so this is a valid assumption. The following timing information has been taken from [23].

The CAN bus controller device uses a 25 MHz crystal as a clock.

$$f_{\text{osc}} = 25 \text{MHz}$$

The nominal bit rate (NBR) on the CAN bus is defined as

$$NBR = f_{\text{bit}} = \frac{1}{t_{\text{bit}}}$$

$$t_{\text{bit}} = t_{\text{syncseg}} + t_{\text{propseg}} + t_{ps1} + t_{ps2}$$

Each of these values can be described as a multiple of time quantas $T_Q$. These are defined as:

$$T_Q = \frac{2(BRP + 1)}{f_{\text{osc}}}$$
BRP is a bit rate parameter that can be set in one of the CAN control registers. The BRP in this case is set to 24.

\[
BRP = 24 \implies T_Q = \frac{50}{f_{osc}} = 2\mu s
\]

\(t_{syncseg}\) is the time required for the CAN bus to synchronise different nodes. It is defined in the specification to be one time quanta long.

\[
t_{syncseg} = T_Q
\]

\(t_{propseg}\) is used to compensate for different physical delay between nodes. This value can be set in one of the CAN control registers and in this case is set to:

\[
t_{propseg} = 3T_Q
\]

\(t_{ps1}\) and \(t_{ps2}\) are used to compensate for phase errors on the bus. These values can be set in the CAN control register and is in this case set to:

\[
t_{ps1} = 5T_Q, t_{ps2} = 2T_Q
\]

This gives a total of:

\[
t_{bit} = 11T_Q = 22\mu s \implies f_{bit} \approx 45.5\, kHz
\]

The CAN transmitter uses the standard CAN protocol. The bit length in this protocol can be described as:

\[
l_{message} = 44 + 8N, 0 \leq N \leq 8
\]

where \(N\) is the number of data bytes transmitted. The minimum and maximum data transfer delay would then be:

\[
N = 0, l_{message} = 44 \implies t_{message} = 44 \cdot 22\mu s = 968\mu s = 0.968\, ms
\]

\[
N = 8, l_{message} = 108 \implies t_{message} = 108 \cdot 22\mu s = 2376\mu s = 2.376\, ms
\]

This gives a message frequency between:

\[
421\, Hz \leq f_{message} \leq 1033\, Hz
\]
These values are used as the nominal values on the CAN delay in the virtual platform. The size of the delay is determined by the number of bytes that are sent during each specific CAN bus transmission.

### 7.4.2 Estimation of MIPS value

During some initial measurements, it was noted that the CPU load from the virtual platform resulted in very unexpected values. With the use of the debugger it could be verified that the load counter was around five times higher than expected. This error led back to the MIPS value in the processor model. The processor will finish its tasks too quickly when the MIPS value is set too high. The processor will then enter the thread with the lowest priority, which is the load counter. This value will then become larger than intended.

A reference load was measured on the physical platform. In order to find a corresponding value on the virtual platform the loads from a series of MIPS values were also measured. This resulted in the graph in Figure 7.2.

![MIPS vs CPU load](image)

*Figure 7.2: CPU load vs MIPS. The green line is the value from the physical measurements.*

The values on the virtual and physical CPU load intersected at a value of 6.6 MIPS in the processor model. This value was therefore chosen as one of the initial parameter values for the system.

### 7.4.3 Estimation of response time on external boards

Some messages from the external boards were measured on the physical platform in order to estimate the response time for the external boards.
These messages are responses on alive requests sent from the CPU and two kinds of sensor reports that the external boards are supposed to send to the CPU. These sensor reports are supposed to occur once every 500ms for report 1 and every 1000ms for report 2.

The performed analysis can be seen in Table 7.1. There are the minimum, maximum, and average value presented together with the standard deviation.

<table>
<thead>
<tr>
<th>Value</th>
<th>Alive - Isalive IO</th>
<th>Alive - Isalive UI</th>
<th>Sensor report 1</th>
<th>Sensor report 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>16.9</td>
<td>9.6</td>
<td>502.8</td>
<td>1005.6</td>
</tr>
<tr>
<td>Min</td>
<td>2.9</td>
<td>2.2</td>
<td>473.0</td>
<td>1000.9</td>
</tr>
<tr>
<td>$\bar{x}$</td>
<td>6.4</td>
<td>4.7</td>
<td>500.1</td>
<td>1003.5</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>3.6</td>
<td>1.7</td>
<td>4.8</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 7.1: Response times for messages sent from the external boards. All values are given in milliseconds.

The sensor reports occur with a periodicity similar to the previous defined values. However, it can also be seen in the table that the response time on the alive messages greatly varies. The standard deviation on the response time is high. The response times also varies between as little as 2ms to around 17ms. These large fluctuations cannot depend on transmission delays. It is instead likely that the large differences depend on how busy the external boards are. If an external board is already busy with, for example, processing some sensor data it will take long time to generate an answer, whereas if the board instead is idle it will answer directly.

Since it was not possible to measure the load of the external boards in an easy way, it was decided to set the response time in the virtual platform to the average values of these responses. These average values can be seen in table 7.1.

### 7.4.4 Initial simulation parameter values

The nominal simulation parameters can be seen in Table 7.2. The values of the CAN and SPI have been taken from the calculations described in Section 7.4.1. The value on the processor MIPS was motivated by some initial measurements presented in Section 7.4.2. The CAN/SPI device was given no delay since it was believed that its delay was negligible. In Section 7.4.3 it was shown that the response time of the external boards can vary greatly. The response time for each message type was set to the average value based on the physical measurements. The time slice of the simulation was set to 1 µs. The intent of this was to set it to something shorter than the shortest simulated delay in the system. The shortest delay in this case is the SPI delay with a delay of 7 µs.
7.5 Measurement scenarios

Three measurement scenarios were designed to be performed during the measurements. These scenarios are depicted in figure 7.3.

The first scenario is a start-up and shutdown sequence that will be used to perform the main comparisons between the virtual and physical platform. Timing and metrics such as CPU load are measured here.

The second scenario is similar to the first scenario, but in this case, the system is set in a state where some additional activity is performed. The CPU load will therefore increase. This increase can be compared between the virtual and physical platforms to see how well they correspond with each other on that matter.

In the third scenario, a number of button presses are performed. The reasoning is that when a button is pressed, the CPU will respond with a message to light the corresponding LED as fast as possible. It is therefore possible to measure the complete delay over the transmission chain for this scenario.

An example of raw data from one physical measurement and one virtual measurement can be seen in Appendix B.1 and B.2. That raw data is from a scenario 1 measurement, but the raw data from the other measurements is quite similar to this case.

<table>
<thead>
<tr>
<th>Processor MIPS</th>
<th>6.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeslice</td>
<td>1 µs</td>
</tr>
<tr>
<td>SPI delay</td>
<td>7 µs/byte</td>
</tr>
<tr>
<td>CAN device delay</td>
<td>0 s</td>
</tr>
<tr>
<td>CAN delay</td>
<td>968µs – 2376µs / message</td>
</tr>
<tr>
<td>External delay</td>
<td>Varies depending on message</td>
</tr>
</tbody>
</table>

*Table 7.2: Nominal simulation parameters.*
7.5.1 Scenario 1

The first scenario consists of the following steps:

1. The system is started and then it performs a start-up sequence.
2. When fully initiated, the system will go into a standby loop.
3. The system remains in the standby loop for about 12 seconds.
4. The shutdown button is pressed in the UI.
5. The system performs a shutdown sequence.

This scenario is used since the device performs a number of well-defined CAN bus accesses at start-up and shutdown. These accesses can then be used to compare between the virtual and physical platform.

Between the start-up and shutdown, the system is in a standby loop where it regularly sends messages to the external boards to see that they are still active and responding. The CPU also regularly receives messages from the external boards with sensor values.

The button was manually pressed in the physical simulation and a timer was used to mimic this button press in the virtual platform.

7.5.2 Scenario 2

The second scenario is much like the first but here a button is pressed so that the system leaves the standby mode and enters a mode where it outputs one sensor value to the LCD screen during regular intervals. This gives the CPU more work to do than in the first scenario. This difference can then be measured with the CPU load of the system.
7.5.3 Scenario 3

The third scenario consists of that buttons in the UI are pressed several times. The reason for this is when a button is pressed it informs the CPU with a message. The CPU sends then, as fast as possible, a message to the UI that orders it to light the LED that corresponds to this button. This measurement gives a value of the complete delay back and forth over the CAN bus.

7.6 Performed measurements

In this section the results from the performed measurements are presented.

7.6.1 CPU load

The CPU load was measured for all three measurement scenarios on both the virtual and physical platform. This was documented in Table 7.3.

<table>
<thead>
<tr>
<th>Measurement scenario</th>
<th>Physical CPU load</th>
<th>Virtual CPU load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17 %</td>
<td>17 %</td>
</tr>
<tr>
<td>2</td>
<td>19 %</td>
<td>19 %</td>
</tr>
<tr>
<td>3</td>
<td>17 %</td>
<td>17 %</td>
</tr>
</tbody>
</table>

*Table 7.3: CPU load on different measurement scenarios.*

In scenario 1 and 3 the amount of traffic on the CAN bus is quite similar, apart from the fact that in scenario 3 there are a few extra messages that indicate a button has been pressed.

In measurement scenario 2 the work load of the CPU has increased a bit. The reason for this is that now messages are sent to the LCD screen at regular intervals.

By comparing the virtual and physical platform it can be seen that the measurements correspond well with each other. So the CPU loads seem to increase in a similar way on the virtual and physical platforms. However more measurements must be performed, especially in more load consuming states, before a more secure answer about this can be given.

These measurements were performed with a time slice of 1 \(\mu s\). Another test was made to change the time slice to a value of 1 ms. With this time slice the values proved to be very different and not reliable at all. With the help of a debugger, it was found that the CPU sometimes polls on the SPI bus too long at this time slice. For more details see Section 7.6.4.

7.6.2 Transmission delay

Measurement scenario 3 was used to determine the transmission delay over the transmission chain. The difference in time between when the pressed-button message and when the set-LED message arrived was noted. This measurement
gives a value on how long it takes to send a message and receive a response. The results can be seen in Table 7.4.

<table>
<thead>
<tr>
<th>Platform</th>
<th>$\bar{x}$</th>
<th>$\sigma$</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>4.8</td>
<td>0.5</td>
<td>6.9</td>
<td>4.5</td>
</tr>
<tr>
<td>Virtual</td>
<td>4.77</td>
<td>0.30</td>
<td>6.25</td>
<td>4.63</td>
</tr>
</tbody>
</table>

Table 7.4: Delay over the whole transmission chain, all values are given in milliseconds.

As it can be seen in the table the measurements between the virtual and physical platform correspond quite well. Both average values are located close to each other. This makes it plausible that the delay parameters of the CAN bus in the virtual platform correspond well with the physical delays.

### 7.6.3 Timing error measurements

These measurements were performed to give values on the timing errors between the virtual and physical platforms. In this case measurements from scenario 1 were used. The timing errors were analysed during the start-up and shutdown of the system. The start up sequence was defined from the first message on the CAN bus until the system enters the standby loop. During this period 17 messages are sent and the sequence takes around 1200 ms on the physical platform. The shutdown sequence was defined from the release of the shutdown button until the CPU sends a Halt message on the CAN bus. This sequence consists of 72 messages and takes around 7700 ms to complete on the physical platform.

The used scenario was first measured on the physical platform. These measurements were logged and used as reference for the subsequent measurements on the virtual platform.

The scenario was measured using some different parameter settings. In the first test case, the nominal parameter values as defined in section 7.4.4 were used. After this three different delays on the SPI bus were tested:

- No delay at all (SPI0)
- A delay at 6 $\mu$s (SPI6)
- A delay at 8 $\mu$s (SPI8)

This was done to investigate how changes on the relatively small delay of the SPI bus affected the total simulation. It was also of interest to see if it was possible to simulate without any delay at all. Finally, some changes on the MIPS value were also tested to see how these changes affected the total timing. The tested values were:

- A MIPS of 6.2 (MIPS6.2)
- A MIPS of 7.0 (MIPS7.0)
The timing metrics defined in Section 7.4.4 were calculated. The results from the start up can be seen in Table 7.5, while the results from the shutdown can be seen in Table 7.6. The following values are presented in the table:

- The duration of the sequence.
- The average error for each error type.
- The maximum error for each error type.
- The standard deviation for each error type.

In Table 7.7 were also the relative errors for both start-up and shutdown presented.

<table>
<thead>
<tr>
<th>Name</th>
<th>Duration</th>
<th>$\delta_{\text{drift}}$</th>
<th>Max($\delta_{\text{drift}}$)</th>
<th>$\sigma(\delta_{\text{drift}})$</th>
<th>$\delta_{\text{abs}}$</th>
<th>Max($\delta_{\text{abs}}$)</th>
<th>$\sigma(\delta_{\text{abs}})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>1201</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Nominal</td>
<td>1197</td>
<td>18.08</td>
<td>32.61</td>
<td>13.69</td>
<td>4.52</td>
<td>32.87</td>
<td>10.22</td>
</tr>
<tr>
<td>SPI0</td>
<td>1196</td>
<td>18.09</td>
<td>32.58</td>
<td>13.70</td>
<td>4.52</td>
<td>32.88</td>
<td>10.23</td>
</tr>
<tr>
<td>SPI6</td>
<td>1197</td>
<td>18.09</td>
<td>32.61</td>
<td>13.69</td>
<td>4.52</td>
<td>32.86</td>
<td>10.21</td>
</tr>
<tr>
<td>SPI8</td>
<td>1197</td>
<td>18.10</td>
<td>32.63</td>
<td>13.69</td>
<td>4.52</td>
<td>32.92</td>
<td>10.22</td>
</tr>
<tr>
<td>MIPS6.2</td>
<td>1222</td>
<td>20.06</td>
<td>32.65</td>
<td>12.67</td>
<td>4.63</td>
<td>32.85</td>
<td>9.48</td>
</tr>
<tr>
<td>MIPS7.0</td>
<td>1192</td>
<td>18.10</td>
<td>32.55</td>
<td>14.12</td>
<td>5.01</td>
<td>32.81</td>
<td>10.83</td>
</tr>
</tbody>
</table>

*Table 7.5: Simulation results from the start up, using a time slice of 1µs. All values are given in milliseconds.*

<table>
<thead>
<tr>
<th>Name</th>
<th>Duration</th>
<th>$\delta_{\text{drift}}$</th>
<th>Max($\delta_{\text{drift}}$)</th>
<th>$\sigma(\delta_{\text{drift}})$</th>
<th>$\delta_{\text{abs}}$</th>
<th>Max($\delta_{\text{abs}}$)</th>
<th>$\sigma(\delta_{\text{abs}})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>7682</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Nominal</td>
<td>7680</td>
<td>28.16</td>
<td>57.32</td>
<td>18.01</td>
<td>7.49</td>
<td>39.19</td>
<td>13.59</td>
</tr>
<tr>
<td>SPI0</td>
<td>7682</td>
<td>27.29</td>
<td>56.40</td>
<td>17.97</td>
<td>7.38</td>
<td>39.24</td>
<td>13.49</td>
</tr>
<tr>
<td>SPI6</td>
<td>7680</td>
<td>27.16</td>
<td>57.27</td>
<td>17.32</td>
<td>7.39</td>
<td>39.24</td>
<td>13.36</td>
</tr>
<tr>
<td>SPI8</td>
<td>7680</td>
<td>28.30</td>
<td>57.51</td>
<td>18.05</td>
<td>7.51</td>
<td>39.17</td>
<td>13.61</td>
</tr>
<tr>
<td>MIPS6.2</td>
<td>7748</td>
<td>32.28</td>
<td>86.43</td>
<td>30.5</td>
<td>9.59</td>
<td>87.9</td>
<td>18.14</td>
</tr>
<tr>
<td>MIPS7.0</td>
<td>7682</td>
<td>28.51</td>
<td>55.40</td>
<td>16.06</td>
<td>6.50</td>
<td>34.97</td>
<td>11.77</td>
</tr>
</tbody>
</table>

*Table 7.6: Simulation results from the shutdown, using a time slice of 1µs. All values are given in milliseconds.*

The difference between the durations of the virtual and physical systems is quite low, if nominal simulation values are used. The size of duration does not change much when the SPI delay is changed. This means that the SPI delay does not contribute much to the total delay of the system. It is first when the MIPS value is changed that the timing starts to deviate. This means that the MIPS value is a central parameter when it comes to obtaining a correct value of the timing in the system.

By studying the tables it can also be seen that the drift errors are quite large. The average values lie somewhere in the regions of 20 ms. Since the total duration of the sequence is mainly correct, it means that some CAN transmissions happen at
Figure 7.4: The error values during start-up

Figure 7.5: The error values during shutdown
### 7.6 Performed measurements

<table>
<thead>
<tr>
<th>Name</th>
<th>$\bar{\epsilon}_{rel}$</th>
<th>$Max(\epsilon_{rel})$</th>
<th>$\sigma(\epsilon_{rel})$</th>
<th>$\bar{\epsilon}_{rel}$</th>
<th>$Max(\epsilon_{rel})$</th>
<th>$\sigma(\epsilon_{rel})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>3.38</td>
<td>9.82</td>
<td>3.21</td>
<td>17.36</td>
<td>99.91</td>
<td>25.79</td>
</tr>
<tr>
<td>SPI0</td>
<td>3.46</td>
<td>10.27</td>
<td>3.43</td>
<td>16.90</td>
<td>94.84</td>
<td>24.32</td>
</tr>
<tr>
<td>SPI6</td>
<td>3.39</td>
<td>9.86</td>
<td>3.20</td>
<td>17.54</td>
<td>93.40</td>
<td>25.46</td>
</tr>
<tr>
<td>SPI8</td>
<td>3.38</td>
<td>9.77</td>
<td>3.18</td>
<td>17.46</td>
<td>99.75</td>
<td>25.79</td>
</tr>
<tr>
<td>MIPS6.2</td>
<td>6.04</td>
<td>29.06</td>
<td>8.28</td>
<td>19.23</td>
<td>121.59</td>
<td>28.26</td>
</tr>
<tr>
<td>MIPS7.0</td>
<td>3.72</td>
<td>9.82</td>
<td>3.23</td>
<td>15.93</td>
<td>82.12</td>
<td>22.51</td>
</tr>
</tbody>
</table>

*Table 7.7: Relative errors from both start-up and shutdown, using a time slice of 1µs. All values are given in percent.*

the wrong time on the virtual platform. These errors result in that a drift in the total system timing occurs. The size of the drift error did not change much with the different parameter selections.

By instead studying the absolute error it can be seen that the average error is around 5ms. This is significantly smaller than the drift error. The interpretation of this is that the majority of the messages are sent at the right time in relation to each other. However, a small amount of messages are sent on the wrong time. Some motivations for these errors will be shown in Chapter 8. Note that some of the maximum values on the absolute error is larger than the drift error. This is possible since the absolute error is the difference between two transmissions in the virtual set and two transmissions in the physical set. This error can be large if for example two transmissions happens close in time in the virtual platform will the same two transmissions would happen with a large delay in the physical platform. The absolute error is therefore a good way to spot these errors in the measurements.

In Figure 7.4 and 7.5 the drift error and absolute error are plotted over time. In Figure 7.4 is the start-up plotted and in Figure 7.5 is the shutdown plotted.

It can be seen in Figure 7.4 that one sample is delayed an creates a large error. This results in that the drift error remains quite large for a long time. The absolute error does instead drop to a lower value directly after the next sample. This illustrates the difference between the drift error and the absolute error quite well. The drift error depends on earlier errors while the absolute error only looks at the present error for each sample.

In Figure 7.5 it can be seen that the errors takes a sawtooth looking formation. This glitchy behaviour is due to that the message sequence mainly consists of sensor updates and alive requests. Both these messages transactions are repeated under regular intervals. This can be seen in the figure since the shapes are repeating themselves once every 1000 ms.

The reason that this error occurs is that the sensor updates on the virtual platform are in good phase with those on the physical platform. On the other hand there is a phase difference between the alive request on the virtual platform and the
physical platform. Therefore the simulation jump from being in phase in for some sample and not in phase for others. This jumping back and forth results in the sawtooth looking behaviour.

The relative error can be observed in Table 7.7 here it can be seen that the relative error is around 5% for the start-up and around 15% to 20% for the shutdown. In the shutdown it can be observed that the maximum relative error is large and for some parameters even larger than 100%. The reason for this was located in the data sets. There it was observed in one instance that one transmission happened nearly directly after another transmission in the virtual set. In the physical set there was instead a delay of a couple milliseconds. This resulted in that the error was larger than the delay between the transmission. This resulted in an error around or even larger than 100%.

From the tables it can be seen that the parameter setup that gives the least errors in this case are the nominal values. The difference in the errors when it comes to the SPI delay does not really affect the timing in the simulation so much. The measurements also show that the MIPS value must be precisely tuned to achieve the correct timing. The reason that the average response time is used on the external boards is also responsible for a part of the error size. To obtain low errors the greatest effort should therefore put on modelling a correct MIPS value and achieve proper response timing on the external boards.

The changes in the SPI delay did not seem to affect the total simulation of the delay. Therefore another test was done to simulate the parameters with a time slice that was intentionally larger than the SPI delay. The time slice was set to 1 ms, which is still smaller than the other simulated device delays. The intent of this was to get a faster simulation time and to see if the measurements still were stable at this speed. These measurements were taken during the start up of the system and the results can be seen in Table 7.8.

| Name         | Duration | $\bar{\epsilon}_{drift}$ | $Max(\epsilon_{drift})$ | $\sigma(\epsilon_{drift})$ | $\bar{\epsilon}_{abs}$ | $Max(\epsilon_{abs})$ | $\sigma(\epsilon_{abs})$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>1203</td>
<td>17.85</td>
<td>31.80</td>
<td>12.92</td>
<td>4.24</td>
<td>32.34</td>
<td>9.84</td>
</tr>
<tr>
<td>SPI0</td>
<td>1159</td>
<td>18.09</td>
<td>32.58</td>
<td>13.70</td>
<td>4.52</td>
<td>32.88</td>
<td>10.23</td>
</tr>
<tr>
<td>SPI6</td>
<td>1264</td>
<td>19.75</td>
<td>63.21</td>
<td>16.19</td>
<td>9.88</td>
<td>41.87</td>
<td>14.78</td>
</tr>
<tr>
<td>SPI8</td>
<td>1203</td>
<td>17.85</td>
<td>31.82</td>
<td>12.92</td>
<td>4.24</td>
<td>32.35</td>
<td>9.84</td>
</tr>
<tr>
<td>MIPS6.2</td>
<td>1312</td>
<td>28.13</td>
<td>111.17</td>
<td>25.24</td>
<td>13.95</td>
<td>75.87</td>
<td>21.97</td>
</tr>
<tr>
<td>MIPS7.0</td>
<td>1195</td>
<td>18.20</td>
<td>31.84</td>
<td>12.87</td>
<td>4.46</td>
<td>32.34</td>
<td>9.90</td>
</tr>
</tbody>
</table>

Table 7.8: Simulation results from start up, using a time slice of 1ms. All values are given in milliseconds.

As can be seen in the table the timing of the simulation is much less stable in this case. Small changes on the SPI result in large changes of the duration and the errors change in an indeterministic way. It can however be worth to note that even if the timing on these measurement were erroneous all messages did arrive and in the right order. There is therefore no problem to use a time slice of 1ms if only a functional simulation is wanted.
7.6.4 Simulation speed

The simulation speed depends mainly on two things. The first factor is the virtual platform code. Since the simulation time is paused when the simulator executes virtual platform code, complex or unoptimised code will result in a slower simulation. For example, the timers that simulate the device delays were changed from being destructed at time-out to instead enter a standby mode. This saved a large amount of time that was spent on unnecessary timer initiation. This change resulted in a notably better simulation speed.

The second factor is the time slice. With a time slice of 1 ms, the simulation speed usually ends up around 30 to 35 MIPS which is around 4 to 5 times faster than the physical speed. The simulation still behaves reasonably well and does not crash. This time slice however gives low accuracy on events that are smaller than this time slice. This results in that the SPI accesses can behave weird. For example, if the processor polls on the SPI bus too long it will result in erroneous values on the CPU load.

With a time slice of 1 µs, the simulation ends up at 10 to 15 MIPS instead. This is around 1.5 to 2 times faster than the physical simulation, but here the timing on the SPI bus behaves much better and no strange behaviour occurs.

In Table 7.9 values on how much time each PSE unit required from the host are shown. In the simulation a time slice of 1 µs was used and the application was put into standby mode. It was then allowed to run for 673 seconds in real time. This resulted in a simulated time of 1312 seconds.

<table>
<thead>
<tr>
<th>PSE unit/thread</th>
<th>Time (s)</th>
<th>Number of executed callbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SER</td>
<td>15.74</td>
<td>8969423</td>
</tr>
<tr>
<td>MEM</td>
<td>0.01</td>
<td>874</td>
</tr>
<tr>
<td>GEN</td>
<td>7.41</td>
<td>5112304</td>
</tr>
<tr>
<td>DMA</td>
<td>0.08</td>
<td>53654</td>
</tr>
<tr>
<td>EFE</td>
<td>0.47</td>
<td>856024</td>
</tr>
</tbody>
</table>

Table 7.9: Required host time for the different PSE units.

In the table it can be seen that the majority of hardware access goes to the SER and GEN units. This is not surprising since the most of the functionality is located in these two units. The EFE unit has a large number of accesses due that the unit is repeatedly polled during the initiation to verify if an Ethernet connection is available or not.

7.6.5 SPI accesses

The SPI accesses were logged during regular intervals of the simulation. This measurement was only performed on the virtual platform. The reason for this was that no effective measurement tool was available for physical measurements on this node. These measurement were performed with the intent to show how
a virtual platform, in an easy way, can analyse otherwise tedious measurement situations.

In this case measurement scenario 1 was used. The number and types of accesses can be seen in Table 7.10 and Figure 7.6.

In this case a read means that the CPU reads a value from the SPI bus and a write is when the CPU sends a value over the SPI bus. Because of this, 1000 EEPROM writes are not the same as 1000 writes to the EEPROM memory. There are always some overhead in the SPI protocols used by these devices. The CPU must for example make two writes (selection of operation mode and address) and one read to get the content of one byte in the EEPROM.

The complete simulation which took 28 seconds in simulated time and around 21000 SPI accesses were logged.

The general conclusion from this measurement is that the SPI accesses grow quickly during the start-up and shutdown of the system. During the normal operation the SPI accesses grow rather linearly. The rapid accesses indicate that the SPI delay can contribute with accuracy to the time it takes at start up and shutdown the system. It can therefore be worth the effort to simulate the SPI delay even if it does not affects the message timing over the CAN bus so much.

Some analyses of the individual SPI accesses to each SPI slave device are shown below:

EEPROM

The EEPROM accesses grow rapidly at start-up and shutdown of the system. At start-up the content of the EEPROM is read and at shutdown a number of writes are performed to it. Otherwise, the EEPROM remains rather untouched during the rest of the simulation.

RTC

The content of the RTC is mostly read and most write accesses are mainly protocol overhead. The RTC accesses grow linearly during execution apart from in the beginning when a large amount of reads are done.

CAN

The CAN bus used for device communication (CAN-COM) and the bus used for control of the frequency regulator (CAN-EXT) behave a little bit differently. The accesses on the later grow linearly since it regularly polls on the bus for a response, but it gets none. No device is connected to it. The accesses on the CAN-COM grow linearly, but with higher frequency at start-up and after around 20 seconds, when the shutdown button is pressed.
Perform measurements

Figure 7.6: Measurements of SPI accesses on the virtual platform.
<table>
<thead>
<tr>
<th>SPI action</th>
<th>Number of accesses</th>
<th>Percent of accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC writes</td>
<td>323</td>
<td>1.6</td>
</tr>
<tr>
<td>RTC reads</td>
<td>1432</td>
<td>7.0</td>
</tr>
<tr>
<td>EEPROM writes</td>
<td>5623</td>
<td>27.5</td>
</tr>
<tr>
<td>EEPROM reads</td>
<td>4785</td>
<td>23.5</td>
</tr>
<tr>
<td>CAN-COM writes</td>
<td>1862</td>
<td>9.1</td>
</tr>
<tr>
<td>CAN-COM reads</td>
<td>1596</td>
<td>7.8</td>
</tr>
<tr>
<td>CAN-EXT writes</td>
<td>3205</td>
<td>15.7</td>
</tr>
<tr>
<td>CAN-EXT reads</td>
<td>1567</td>
<td>7.7</td>
</tr>
</tbody>
</table>

*Table 7.10: Accesses over the SPI bus.*
This chapter presents the results and conclusions from the measurements. It also presents some general comments about the modelling of the platform and a short comparison with other development tools.

8.1 Comparisons between the virtual and physical platform

8.1.1 Reliability

The timing in the system seems to work sufficiently well. It means that the simulation will behave very much like the real platform. Messages will be sent in the same period of time, no messages will be dropped and the application code can be executed without faults. Differences do however occur in the timing between the virtual and physical platform. The message timing and the simulated CPU load seems to correspond well if there is sufficient size on the time slice. The response time over the CAN bus also corresponds well, as illustrated in Table 7.4.

8.1.2 Simulation speed

The ever-present trade off between accuracy and simulation speed is illustrated in the simulation. High simulation speed can be acquired at the cost of timing accuracy.

It is possible to acquire a simulation that is around 4 to 5 times faster than the physical platform by using a time slice of 1ms. The transmissions over the CAN bus will work without larger errors, but the measurements in previous chapter
show that the timing on SPI bus becomes unstable and that the CPU load cannot be relied upon.

By lowering the time slice to 1 µs, these values become more reliable again, but the simulation speed will be lowered to around 1.5 to 2 times faster than the physical platform. This is something that should be taken into consideration. If there is no need for timing on a low level and a more functional model is wanted, it could be possible to simulate with a larger time slice and thus gain higher simulation speed.

8.1.3 Timing

The results in Table 7.5 and 7.6 show that the duration of the start up and shutdown is nearly the same in the virtual and physical platform. This means that the timing is relatively accurate here. The greatest problem is that the drift error is quite large during both the start up and shutdown. The reason for this was found in the data sets. The explanation is that a few samples have a large difference in timing between the virtual and physical platform. These errors shift all other samples in time, thus resulting in larger errors.

Most of the transmissions occur with good resemblance between the virtual and physical platform. This can be seen when the absolute error is smaller, usually around 5ms. The size of the absolute error can be attributed to the fact that the average values on the response time are used.

From the simulation it can be concluded that the MIPS and the response time on external boards have the greatest effect on timing. By changing the MIPS value the duration on the start up and shutdown changes, as expected. Changes in the SPI delay have a smaller effect in the metrics. The timing at 1 µs correspond well with the reality overall. However, if a time slice of 1ms is used, it can be seen in Table 7.8 that the behaviour is much more unstable if some parameters are changed. The simulation can, however, still run on and it is just the timing that becomes less accurate.

The reason that the timing of the virtual and physical platform can be some milliseconds off should not be attributed to that the simulation is unreliable, as long a proper time slice is used. The measurements indicate instead that the main reason is because the behaviour of the complete system is indeterministic. The randomness in the system can depend on things like:

- Random delay before reaccess of hardware devices.
- Reception of CAN messages.
- Response time on external boards.

When the application tries to access a hardware device that is busy, that thread will be suspended until the next timer tick. A timer tick happens once every 10ms. Depending on when the access occurs, the delay until the reaccess can then be anything between 0-10 ms. One other reason for the uncertainty is that if the CPU receives a CAN message when it is about send a message the transmission
will be suspended until later. The CPU will then handle the received message instead. This means that messages can be suspended for a long time if the CPU receives a message at the "wrong" time. Finally is it hard to know how long an external board will take to respond on a message since it can depend on how busy the board is.

It can be discussed if it is necessary to simulate the timing with such accuracy that these factors are taken into consideration. The application is designed to be able to handle the randomness of the system. The application can be kept running without faults even if some metrics may be large, such as the drift error.

The timing that is controlled by the OS works generally without any problems. If the application wants to send a message every second this can be done with small timing error. This works well since the timer in the OS is controlled by a hardware timer that is controlled by the simulation time. The real problem is how to connect the simulation time and real time with each other. This is especially problematic with actions whose timing is not controlled by the OS. Here it is only the MIPS value that decides how fast the timing will proceed. If the value of the MIPS does not correspond to the real execution speed of the system it will result in timing errors.

**MIPS**

The MIPS value is one of the most important parameters in the simulation since it determines how fast the simulation time will proceed. It can be discussed if this is a good value to use as a parameter for the simulation time. The MIPS value from a CPU producer can be significantly different from the real execution speed on an embedded system, where things such as the memory architecture can effect the speed. Therefore, measurements on the physical platform are the only way to know which MIPS value the virtual platform should have. This can be a problem if software with time critical needs are to be designed with the help of the platform and no physical hardware is available yet. In that case the designers must be aware of how fast the real execution probably will be, in order to set an appropriate value on the MIPS.

**External board delay**

The external board delay is another important parameter in the simulation. The reason is that the responses on these boards take the longest time in the transmission chain. They also show a random behaviour, thus creating the largest error sources. In this platform the average values on the response times are used, but they produce a lot of difference in timing between the virtual and physical platforms. If these response times are to be simulated more accurately, some kind of model of the external boards must be used. It can then be possible to know when the external devices are busy or not.

**SPI delay**

The conclusion is that the SPI delay is necessary to be simulated accurately if the CPU load is required to be simulated accurately. Otherwise, the SPI accesses are
too few and happen too fast in comparison with the CAN timing to be of relevance. The SPI bus is the only hardware that is polled until it is ready. If an access is polled too long or too short it will result in erroneous values on the simulated CPU load.

**CAN device delay**

The delay of the CAN/SPI converter must most likely not be simulated. The delay of this device is most probably smaller than the size of the time slice and happens too seldom to effect the complete result.

**CAN bus delay**

From the simulations of the response time on the bus in Section 7.6.2 it seems that the delay on the CAN bus corresponds well with reality.

### 8.2 Modelling

The author of this thesis agrees with the conclusion that Sunpack [14] made. Not all parts of an embedded system must be simulated with high accuracy to obtain a relatively well-timed system. Some parts of the system can even be completely disconnected from timing. Parts that have few accesses can often be neglected without effecting the main simulation too much. For example, the RS232 is not connected in any critical transmission chain and is mainly only accessed when a user writes text in the terminal. This happens so seldom that there is no need to simulate the transmission delay that occurs when bytes are transmitted over the channel.

Often, as in Section 7.6.5, it is best to analyse when and how often different parts of the system are used and then determine which level of accuracy is necessary for that device. In general it is often necessary to simulate the delay in most hardware devices, but the delay must not be completely correct. The main purpose of these delays is to prevent too fast accesses by the CPU. For example it does not really matter if 100 accesses on the SPI bus take 600µs or 650µs. The thing that matters is that it would not take 5µs as it maybe would do if no delay was simulated. The exact size of delays does not matter much for the simulation accuracy, as long they are within a reasonable range. The application is quite relaxed when it comes to this and often waits longer than worst case. This is reflected in the measurements where it for example was found out that the CPU waits for 30ms before sending a new message in some cases. This is much longer than the actual time that an external board would require to process the message.

The part that should be modelled with the highest accuracy is the bottleneck of a transmission chain. Even if the SPI delay is modelled correctly at the right microsecond, it will not matter if the response time on an external board can vary with a couple of milliseconds.

The authors of [21] conclude that simulation of memory is one of the most complex parts of a virtual platform. It is most notable in this virtual platform where
just a simple memory is modelled, but still it causes much problems in the simulation. If a more complex memory structure would be used, for example one that contains a cache, the problems associated with the simulation would become even more complex.

It could be possible to compensate for some of the simulation errors that appear without increasing the simulation accuracy. For example the paper [13] presents an empirical correction factor for the calculation of power consumption of a system running on a virtual platform. If some relation between, for example, the CPU load and the memory accesses could be found, it would be possible to compensate for that and get a correct value on the CPU load without the need of high accuracy in the simulation.

8.3 The developed virtual platform compared with other tools

The developed virtual platform should mainly be seen as a tool intended for software development and testing. It offers a simulation with a speed that is a bit faster than the physical platform. The accuracy is good enough to keep the application running without failing to keep the deadlines that the RTOS must fulfill. However should design of time critical parts not be performed without a physical platform available or knowledge of the final physical execution speed. The reason for this is the fact that the MIPS value of the virtual platform must be tuned with a physical reference to obtain proper timing.

The virtual platform gives portability to the system, allowing concurrent development without the need of large amounts hardware and testing equipment. The virtual platform also gives a more controllable environment, where automated tests and better debugging capabilities could be possible. For example the virtual platform can log accesses at places where regular measurements would be too tedious to perform.

The virtual platform should be seen as instruction accurate with some sense of timing and is not in any way cycle accurate. If this behaviour is needed it is probably better to use SystemC or RTL to simulate the platform.

The most important question that virtual platform designers should ask themselves is what they want to simulate. There are many different full system simulators available, but they are often designed to fulfill some particular need. If the wrong tool is used it is probable the result will be as [21] formulates it: "The result is accurate answers to irrelevant questions."

There are in general three approaches when it comes to developing platforms:

- RTL (cycle accurate)
- ISS + RTL/ ISS + SystemC (cycle accurate or instruction accurate)
- Functional tool (instruction accurate)
Which development tool to choose depends on how accurate and how fast the simulation should be. The paper [10] presents some comparisons between the different approaches. The paper states that a boot of the Linux kernel would take 28 hours to simulate with RTL models in Verilog. By using an ISS together with SystemC the time can be diminished to around a couple of hours, depending on how much optimization that is performed. It is first when the cycle accuracy is removed that the simulation time can be decreased to a couple of minutes.

The paper [10] also presents some comparisons between different simulation tools, both cycle accurate and instruction accurate. The general conclusion here is that instruction accurate simulators can obtain a simulation speed of around 1-100 MIPS depending on which simulation method is used and how accurate things like timing is. Cycle accurate simulators can at best reach 1 MIPS.

The intent of this platform was to simulate a complete system at a speed of the physical platform or even faster. The obvious choice then was to choose a functional tool that removes some accuracy in benefit for speed.

An alternative to the developed platform could be to use an ISS together with SystemC to simulate the system. This would probably result in better accuracy on the timing, but with a simulation time in the reasonable range.

As a general conclusion of the comparisons some of the features and drawbacks of the developed virtual platform are presented.

The reasons to use the developed platform are:

- Can simulate the application code without software faults.
- Fast simulation speed can be achieved.
- The development tool is free of charge.
- A simple and powerful API for development is available.

The drawbacks of the developed platform are:

- The platform is an application specific solution, not a general simulator.
- Few factors can effect how the timing is represented in the simulation.
- The development tool does not have a large development community.
The thesis shows that it is possible to implement a virtual platform of a rather simple embedded system in a limited amount of time. This is possible without the need to modify the executable designed to run on the system.

The simulation can run up to 4 to 5 times faster than the physical system if some accuracy losses on the SPI simulation and CPU load are acceptable. It is possible to simulate these accurately, but then the speed will slow down to around 1.5 to 2 times faster than the physical platform.

C and OVP provide an easy and powerful way to model hardware, but the OVP simulator is highly optimised for speed and should therefore be seen more as a tool designed to help software development rather than hardware verification. If a system should be simulated with a more cycle accurate nature, SystemC or even RTL may be a better choice.

The conclusions from the measurements on the physical system are that the virtual platform corresponds with the physical sufficiently well to enable the simulation of the system on a functional level. This means that the virtual model can run the application and keep communication over the CAN bus without any undefined errors.

Regarding timing, events that are controlled by the OS can be simulated accurately. However is some behaviour, like response times on the external boards indeterministic. This makes it hard to simulate the system with a high level of precision. This would probably require modelling of the work load on the external boards.

The top-down methodology used to develop the virtual platform proved to be effective when it comes to making a platform that in short time is capable of exe-
cuting the application code. The general conclusion about modelling and delays is that delays in peripherals often must be simulated, but there is not always a need to simulate them with high accuracy. The focus should instead be put on the parts that have the largest effects on the system and to simulate these well.
This chapter proposes some further improvements that can be made on the virtual platform.

10.1 Dynamic response generation

In this thesis a simple response generation unit was implemented. This unit tries to simulate the external boards that the platform can communicate with. However, these replies on the CAN bus are quite static. To simulate the whole environment, a more dynamic response generation could be implemented. This would mean that the user could generate more customizable answers. To this could also a GUI be added to simulate the user interface. This would give a more complete simulation where a user or a script easier could control the input and output to the simulation.

10.2 Ethernet/DMA implementation

In this thesis support for Ethernet was not implemented since it not was considered necessary for the construction of a working virtual platform. Future work could consist of adding extra functionality to the EFE-module to provide an Ethernet interface and to model the external MII component. This would require study of the TCP/IP middleware to learn how it is implemented. Since the Ethernet communication utilizes the DMA, it would also be necessary to implement this functionality in the DMA-module and also add support for DMA interrupts.
10.3 RAM timing

During the initial measurements it was shown that the load of the CPU was much lower on the virtual platform than in the physical. This was solved by tuning the MIPS value in the processor model. It is believed that the reason of this difference occurs because the OVP RAM model does not take timing factors, such as access time, into consideration. Therefore it could be possible to make a more complex model of the RAM memory where, for example, wait states can be modelled. Would such model of the RAM give a better simulation or would this create too much negative effects on the simulation speed?
Appendix
A.1 platform.c

This appendix contains an example of how a platform file can be written with the OVP API.

```c
#define SIM_ATTRS (ICM_ATTR_DEFAULT)

const char ∗arm7model;
const char ∗semihosting;
const char ∗applicationName;

int main(int argc, char ∗∗ argv)
{
  
  if ((argc!=2)&&(argc!=3))
  {
    // Incorrect command line arguments
    icmPrintf("Incorrect number of arguments.");
  }
```
A Example code

Specify elf-file and/or debug port.

```c
return -1;
```

```c
applicationName = argv[1];
unsigned int debugPort = 0;
Boo enableDebug = False;
Boo autoGDBConsole = False;

if(argc==3)
{
    enableDebug = True;
    if (!strcmp(argv[2], "auto"))
        autoGDBConsole = True;
    else
        sscanf(argv[2], "%d", &debugPort);
}
```

```
unsigned int icmInitAttrs = ICM_VERBOSE;
if(autoGDBConsole)
{
    icmInitAttrs |= ICM_GDB_CONSOLE;
}
icmInit(icmInitAttrs, enableDebug ? "rsp" : 0, debugPort);
```

```
unsigned int icmInitAttrs = ICM_VERBOSE;
if(autoGDBConsole)
{
    icmInitAttrs |= ICM_GDB_CONSOLE;
}
icmInit(icmInitAttrs, enableDebug ? "rsp" : 0, debugPort);
```

```
// Select processor library components

const char *vlnvRoot = NULL; // Use default library

// Get ARM processor core file path
arm7model = icmGetVlnvString(vlnvRoot, "arm.ovpworld.org", "processor", "arm", "1.0", "model");
icmAttrListP icmProcessorAttr = icmNewAttrList();

// Set processor variant
icmAddStringAttr(icmProcessorAttr, "variant", "ARM7TDMI");

// Set endian ordering
icmAddStringAttr(icmProcessorAttr, "endian", "big");

// Set compatibility with GDB
icmAddStringAttr(icmProcessorAttr, "compatibility", "gdb");

// Set disassembly output UAL syntax
icmAddStringAttr(icmProcessorAttr, "UAL", "1");

// Set processor MIPS
icmAddDoubleAttr(icmProcessorAttr, "mips", 7.15);
```
```c
// Create processor
icmProcessorP processor = icmNewProcessor(
    "cpu1", // CPU name
    "arm", // CPU type
    0, // CPU cpuId
    32, // address bits
    arm7model, // model file
    "modelAttrs", // morpher attributes
    SIM_ATTRS, // enable tracing or register values
    icmProcessorAttr, // user-defined attributes
    0, // No semihosting used
    "modelAttrs" // semihosting attributes
);

// Choose to debug this processor
icmDebugThisProcessor(processor);

// Create and connect bus

// Create the processor bus
icmBusP bus = icmNewBus("bus", 32);

// Connect processor to the bus
icmConnectProcessorBusses(processor, bus, bus);

// Create and connect memory

// Create memory map
icmMemoryP main_memory = icmNewMemory("main_memory", ICM_PRIV_RWX,
    0x01fffffff);
icmMemoryP main_memory2 = icmNewMemory("main_memory2", ICM_PRIV_RWX,
    0xff6fffffff−0x03000000);

// Create external Flash device
icmMapExternalMemory(bus,"external",ICM_PRIV_RWX,0x02000000,0x02fffffff,
    mem_read_cb,mem_write_cb,0);

// Connect the memory onto the buses
icmConnectMemoryToBus(bus,"mpl", main_memory, 0x00000000);
icmConnectMemoryToBus(bus,"mpl", main_memory2, 0x03000000);

// Load the application executable file into processor memory space
if (!icmLoadProcessorMemory(processor, applicationName,
    ICM_LOAD_DEFAULT, False, True))
    return −1; // Operation failed

// Create Peripherals

// EFE module
const char *efePSE = icmGetVlnvString(vlnvRoot, "wexiodisk");
```
"peripheral", "EFE","1.0","efe.pse"));

icmAttrListP efeAttrs = icmNewAttrList();

// Create peripheral
icmPseP efe_module = icmNewPSE("efe_module",efePSE,efeAttrs,NULL,NULL);

// Connect peripheral to memory map
icmConnectPSEBus(efe_module, bus,"busPort",False,0xff800000,0xff8fffffff);

/////////DMA module

// Get filepath for module
const char *dmaPSE = icmGetVlnvString(vlnvRoot, "wexiodisk",
"peripheral","DMA","1.0","dma.pse");

icmAttrListP dmaAttrs = icmNewAttrList();

// Create peripheral
dma_module = icmNewPSE("dma_module",dmaPSE,dmaAttrs,NULL,NULL);

// Connect peripheral to memory map
dicmConnectPSEBus(dma_module, bus,"busPort",False,0xff900000,0xff9fffffff);

/////////GEN module

// Get filepath for module
const char *genPSE = icmGetVlnvString(vlnvRoot, "wexiodisk",
"peripheral","GEN","1.0","gen.pse");

icmAttrListP genAttrs = icmNewAttrList();

// Create peripheral
gen_module = icmNewPSE("gen_module",genPSE,genAttrs,NULL,NULL);

// Connect peripheral to memory map
genConnectPSEBus(gen_module, bus,"busPort",False,0xffb00000,0xffbffffffff);

/////////MEM module

// Get filepath for module
const char *memPSE = icmGetVlnvString(vlnvRoot, "wexiodisk",
"peripheral","MEM","1.0","mem.pse");

icmAttrListP memAttrs = icmNewAttrList();

// Create peripheral
mem_module = icmNewPSE("mem_module",memPSE,memAttrs,NULL,NULL);

// Connect peripheral to memory map
memConnectPSEBus(mem_module, bus,"busPort",False,0xffff0000,0xffffffff);

/////////SER module

// Get filepath for module
const char *serPSE = icmGetVlnvString(vlnvRoot, "wexiodisk",
"peripheral","SER","1.0","ser.pse");
A.1 platform.c

```c
201 icmAttrListP serAttrs = icmNewAttrList();
202 // Port number for RS232 terminal
203 icmAddUns64Attr(serAttrs, "portnum", 12345);
204
205 // Create peripheral
206 icmPseP ser_module = icmNewPSE("ser_module", serPSE, serAttrs, NULL, NULL);
207
208 // Connect peripheral to memory map
209 icmConnectPSEBus(ser_module, bus, "busPort", False, 0xffd00000, 0xffffffffff);
210
211 // /////////// Create nets /////////////
212
213 // Create and connect interrupt nets to processor
214 icmNetP fiq_net = icmNewNet("NFIQ");
215 icmConnectProcessorNet(processor, fiq_net, "fiq", ICM_INPUT);
216 icmConnectPSENet(gen_module, fiq_net, "NFIQ", ICM_OUTPUT);
217
218 icmNetP irq_net = icmNewNet("NIRQ");
219 icmConnectProcessorNet(processor, irq_net, "irq", ICM_INPUT);
220 icmConnectPSENet(gen_module, irq_net, "NIRQ", ICM_OUTPUT);
221
222 // Create and connect chip select nets
223 icmNetP cs_can1_net = icmNewNet("CS_CAN1");
224 icmConnectPSENet(ser_module, cs_can1_net, "CS_CAN1", ICM_INPUT);
225 icmConnectPSENet(gen_module, cs_can1_net, "CS_CAN1", ICM_OUTPUT);
226
227 icmNetP cs_can2_net = icmNewNet("CS_CAN2");
228 icmConnectPSENet(ser_module, cs_can2_net, "CS_CAN2", ICM_INPUT);
229 icmConnectPSENet(gen_module, cs_can2_net, "CS_CAN2", ICM_OUTPUT);
230
231 icmNetP cs_eeprom_net = icmNewNet("CS_EEPROM");
232 icmConnectPSENet(ser_module, cs_eeprom_net, "CS_EEPROM", ICM_INPUT);
233 icmConnectPSENet(gen_module, cs_eeprom_net, "CS_EEPROM", ICM_OUTPUT);
234
235 icmNetP cs_rtc_net = icmNewNet("CS_RTC");
236 icmConnectPSENet(ser_module, cs_rtc_net, "CS_RTC", ICM_INPUT);
237 icmConnectPSENet(gen_module, cs_rtc_net, "CS_RTC", ICM_OUTPUT);
238
239 // Create and connect nets between GEN and SER modules
240 icmNetP rts_net = icmNewNet("RTS");
241 icmConnectPSENet(gen_module, rts_net, "RTS", ICM_INPUT);
242 icmConnectPSENet(ser_module, rts_net, "RTS", ICM_OUTPUT);
243
244 icmNetP can_int_net = icmNewNet("CAN_INT");
245 icmConnectPSENet(gen_module, can_int_net, "CAN_INT", ICM_INPUT);
246 icmConnectPSENet(ser_module, can_int_net, "CAN_INT", ICM_OUTPUT);
247
248 icmNetP can_receive_net = icmNewNet("CAN_RECEIVE");
```
icmConnectPSENNet(ser_module, can_receive_net, "CAN_RECEIVE", ICM_INPUT);
icmConnectPSENNet(gen_module, can_receive_net, "CAN_RECEIVE", ICM_OUTPUT);

// /// Start simulation //////////////////////////////////////////////////////////////////////////////////

// Begin to simulate the platform
icmProcessorP final = icmSimulatePlatform();

// /// End simulation //////////////////////////////////////////////////////////////////////////////////

// Was the simulation interrupted or did it complete as normal?
if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
    icmPrintf("*** simulation interrupted\n");
}

// Generate some simulation statistics
icmDumpRegisters(processor);
icmPrintf("Simulation finished %64u instructions executed\n", icmGetProcessorICount(processor));

// Exit simulation
icmTerminate();

return 0;
}
This appendix contains the h-file for the RTC model. This file is included into a module to give it the capability to access a RTC.

```c
#include "pse.h"

typedef enum
|
RTC_STANDBY,
RTC_READ_BURST,
RTC_WRITE_BURST,
RTC_READ_CENTURY,
RTC_WRITE_CENTURY,
RTC_WRITE_RAM_BURST,
RTC_READ_RAM_BURST,
RTC_READ_RAM,
RTC_WRITE_RAM,
RTC_READ_MAGIC,
RTC_WRITE_MAGIC
|
rtc_state_machine;

typedef struct RTC_CONTROLLER
|
rtc_state_machine state;
Uns32 registers[10];
Uns32 ram[31];
Uns32 counter;
Uns32 ram_address;
Uns32 spi_out;
double spi_read;
double spi_write;
RTC_CONTROLLER;

double get_rtc_spi_read();
double get_rtc_spi_write();
inline void rtc_action(Uns32 *user, Uns32 spi_in, Uns32 chip_select);
void deselect_rtc();
```
A.3 rtc.c

This appendix contains the c-file for the RTC model. It contains function to control and model a RTC.

```c
#include "rtc.h"
#include "pse.h"
#include <time.h>

#define READ_BURST 0xBF
#define WRITE_BURST 0x3F
#define READ_CENTURY 0x93
#define WRITE_CENTURY 0x13
#define READ_MAGIC 0xA5
#define WRITE_MAGIC 0x25
#define RAM_READ_BURST 0xFF
#define RAM_WRITE_BURST 0x7F
#define RAM_READ ( ( (spi_in & 0x40) != 0) && \\
                 ((spi_in & 0x01) == 1) && \\
                 ((spi_in & 0x80) == 0x00000080))
#define RAM_WRITE ( ( (spi_in & 0x40) != 0) && \\
                 ((spi_in & 0x01) == 1) && \\
                 ((spi_in & 0x80) == 0x0))

#define RTC_SECONDS rtc.registers[0]
#define RTC_MINUTES rtc.registers[1]
#define RTC_HOURS rtc.registers[2]
#define RTC_DATE rtc.registers[3]
#define RTC_MONTH rtc.registers[4]
#define RTC_DAY rtc.registers[5]
#define RTC_YEAR rtc.registers[6]
#define RTC_CENTURY rtc.registers[8]
#define RTC_MAGIC rtc.registers[9]

RTC_CONTROLLER rtc =
{
    .state = RTC_STANDBY,
};
```
.registers[0] = 0x3, // Seconds
.registers[1] = 0x5, // Minutes
.registers[2] = 0x1, // Hours
.registers[3] = 0x5, // Date
.registers[4] = 0x2, // Month
.registers[5] = 0x1, // Day
.registers[6] = 0x2, // Year
.registers[7] = 0x0, // Control
.registers[8] = 0x20, // Century
.registers[9] = 0xFF, // Magic
.counter = 0x0,
.ram_address = 0x0,
.spi_out = 0x0,
.spi_read = 0,
.spi_write = 0
);

const int output_RTC_info = 0; // Output details in simulator

// Function: rtc_read_value
// Input:    address
// Output:   Uns32 value
// Purpose:  Outputs a value from one of the RTC registers

Uns32 rtc_read_value(Uns32 address)
{
    return rtc.registers[address];
}

// Function: rtc_write_value
// Input:    value,address
// Output:   None
// Purpose:  Writes a value to one of the RTC registers

void rtc_write_value(Uns32 value, Uns32 address)
{
    rtc.registers[address] = value;
}

// Function: deselect_RTC
// Input:    None
// Output:   None
// Purpose:  Deselects the RTC and puts it into its initial state.

void deselect_RTC()
{
    rtc.state = RTC_STANDBY;
}
Example code

```c
enum
counter = 0;
rtc.ram_address = 0;
}

// Function: encode_bcd
// Input: value
// Output: bcd coded value
// Purpose: Encodes a binary value to BCD code

Uns32 encode_bcd(Uns32 value)
{
    return ((value / 10) << 4) & 0xF0 + (value % 10) & 0x0F;
}

// Function: update_time
// Input: None
// Output: None
// Purpose: Updates the time in the RTC time registers

void update_time()
{
    time_t rawtime;
    struct tm *timeinfo;
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    RTC_SECONDS = encode_bcd(timeinfo->tm_sec);
    RTC_MINUTES = encode_bcd(timeinfo->tm_min);
    RTC_HOURS = encode_bcd(timeinfo->tm_hour + 2); // +2 due to timezone
    // and daylight saving
    RTC_DATE = encode_bcd(timeinfo->tm_mday);
    RTC_MONTH = encode_bcd(timeinfo->tm_mon + 1); // January = 0 in time.h and
    // January = 1 in RTC.
    RTC_DAY = timeinfo->tm_mon + 1; // Monday = 0 in time.h
    // and Monday = 1 in rtc
    RTC_YEAR = encode_bcd(timeinfo->tm_year - 100); // time.h starts at 1900
    // RTC is set at 2000
}

// Function: get_RTC_spi_read
// Input: None
// Output: Double
// Purpose: Returns current number of spi reads to the RTC

double get_RTC_spi_read()
{
    return rtc.spi_read;
}
```

// Function: get_RTC_spi_write()
double get_RTC_spi_write()
{
    return rtc.spi_write;
}

void rtc_standby_action(Uns32 spi_in)
{
    rtc.spi_write++;

    if(spi_in == READ_BURST)
        rtc.state = RTC_READ_BURST;
    else if(spi_in == WRITE_BURST)
        rtc.state = RTC_WRITE_BURST;
    else if(spi_in == READ_CENTURY)
        rtc.state = RTC_READ_CENTURY;
    else if(spi_in == WRITE_CENTURY)
        rtc.state = RTC_WRITE_CENTURY;
    else if(spi_in == RAM_READ_BURST)
        rtc.state = RTC_READ_RAM_BURST;
    else if(spi_in == RAM_WRITE_BURST)
        rtc.state = RTC_WRITE_RAM_BURST;
    else if(spi_in == READ_MAGIC)
        rtc.state = RTC_READ_MAGIC;
    else if(spi_in == WRITE_MAGIC)
        rtc.state = RTC_WRITE_MAGIC;
    else if(RAM_READ)
    {
        rtc.state = RTC_READ_RAM;
        rtc.ram_address = ((spi_in & 0x0000003e) >> 1);
    }
    else if(RAM_WRITE)
    {
        rtc.state = RTC_WRITE_RAM;
        rtc.ram_address = ((spi_in & 0x0000003e) >> 1);
    }
    else
    {
        bhmMessage("F","RTC_STATE", "Unrecognized command %x",spi_in);
    }
}

// Input: None
// Output: double
// Purpose: Returns current number of spi writes to the RTC
// ////////////////////////////////////////////////////////////////////////////////
void rtc_read_burst_action(Uns32 *user)
{
    rtc.spi_read ++;
    if (rtc.counter == 0)
        update_time();
    rtc.spi_out = rtc_read_value(rtc.counter);
    rtc.counter ++;
    if (rtc.counter == 8)
        rtc.counter = 0;
        rtc.state = RTC_STANDBY;
    *(Uns32*)user = rtc.spi_out;
    if (output rtc info)
        bhmMessage("I","RTC","Reads value from RTC register %x at time %f seconds",*(Uns32*)user,bhmGetCurrentTime()/1000000);
}

void rtc_write_burst_action(Uns32 spi_in)
{
    rtc.spi_write ++;
    rtc_write_value(spi_in, rtc.counter);
    rtc.counter ++;
    if (rtc.counter == 8)
        rtc.counter = 0;
        rtc.state = RTC_STANDBY;
    if (output rtc info)
        bhmMessage("I","RTC","Writes %x to RTC register at time %f seconds", spi_in, bhmGetCurrentTime()/1000000);
}
Function: rtc_read_century_action
Input: user (location to place result)
Output: None
Purpose: Reads the value at the Century address at the RTC

```c
void rtc_read_century_action(Uns32 *user)
{
    rtc.spi_read++;
    rtc.spi_out = RTC_CENTURY;
    *(Uns32*)user = rtc.spi_out;
    rtc.state = RTC_STANDBY;

    if(output_rtc_info)
    {
        bhmMessage("I","RTC","Reads %x from CENTURY address at time %f seconds",*(Uns32*)user,bhmGetCurrentTime()/1000000);
    }
}
```

Function: rtc_write_century_action
Input: spi_in
Output: None
Purpose: Writes a value to the Century address at the RTC

```c
void rtc_write_century_action(Uns32 spi_in)
{
    rtc.spi_write++;
    RTC_CENTURY = spi_in;
    rtc.state = RTC_STANDBY;

    if(output_rtc_info)
    {
        bhmMessage("I","RTC","Writes %x to CENTURY address at time %f seconds",spi_in,bhmGetCurrentTime()/1000000);
    }
}
```

Function: rtc_read_magic_action
Input: user (location to place result)
Output: None
Purpose: Reads the value at the Magic address at the RTC

```c
void rtc_read_magic_action(Uns32 *user)
{
    rtc.spi_read++;
    *(Uns32*)user = RTC_MAGIC;
    rtc.state = RTC_STANDBY;
}
```

Function: rtc_write_magic_action
Input: spi_in

```c
void rtc_write_magic_action(Uns32 spi_in)
{
    rtc.spi_write++;
    RTC_MAGIC = spi_in;
    rtc.state = RTC_STANDBY;
}

// Function: rtc_read_ram_action
// Input: user (location to place result)
// Output: None
// Purpose: Reads a single value to the RAM section of the RTC

void rtc_read_ram_action(Uns32 *user)
{
    rtc.spi_read++;
    *(Uns32*)user = rtc.ram[rtc.ram_address];
    rtc.state = RTC_STANDBY;
}

// Function: rtc_write_ram_action
// Input: spi_in
// Output: None
// Purpose: Writes a single value to the RAM section of the RTC

void rtc_write_ram_action(Uns32 spi_in)
{
    rtc.spi_write++;
    rtc.ram[rtc.ram_address] = spi_in;
    rtc.state = RTC_STANDBY;
}

// Function: rtc_write_ram_burst_action
// Input: spi_in
// Output: None
// Purpose: Writes a value to the RAM section of the RTC
// This can be done sequentially without issuing a new address.

void rtc_write_ram_burst_action(Uns32 spi_in)
{
    rtc.spi_write++;
    rtc.ram[rtc.ram_address] = spi_in;
    rtc.ram_address++;
    if(rtc.ram_address == 31)
    {
        rtc.ram_address =0;
        rtc.state = RTC_STANDBY;
    }
}
```
Function: rtc_read_ram_burst_action
Input: user (location to place result)
Output: None
Purpose: Reads a value from the RAM section of the RTC
This can be done sequentially without issuing a new address.

```c
void rtc_read_ram_burst_action(Uns32 *user)
{
    rtc.spi_read ++;
    *(Uns32*)user = rtc.ram[rtc.ram_address];
    rtc.ram_address ++;
    if (rtc.ram_address == 31)
    {
        rtc.ram_address = 0;
        rtc.state = RTC_STANDBY;
    }
}
```

Function: rtc_action
Input: user (location to place result), spi_in, chip_select
Output: None
Purpose: Decides what action to do depending on the state of the RTC

```c
inline void rtc_action(Uns32 *user, Uns32 spi_in, Uns32 chip_select)
{
    if (chip_select == 0)
    {
        switch (rtc.state)
        {
            case RTC_STANDBY :
                rtc_standby_action(spi_in);
                break;
            case RTC_READ_BURST :
                rtc_read_burst_action(user);
                break;
            case RTC_WRITE_BURST :
                rtc_write_burst_action(spi_in);
                break;
            case RTC_READ_CENTURY :
                rtc_read_century_action(user);
                break;
            case RTC_WRITE_CENTURY :
                rtc_write_century_action(spi_in);
                break;
        }
    }
```
case RTC_READ_MAGIC :
    rtc_read_magic_action(user);
    break;

case RTC_WRITE_MAGIC :
    rtc_write_magic_action(spi_in);
    break;

case RTC_READ_RAM_BURST :
    rtc_read_ram_burst_action(user);
    break;

case RTC_READ_RAM :
    rtc_read_ram_action(user);
    break;

case RTC_WRITE_RAM_BURST :
    rtc_write_ram_burst_action(spi_in);
    break;

case RTC_WRITE_RAM :
    rtc_write_ram_action(spi_in);
    break;

default:
    break;

else
|
|
deselect_rtc();
}
B.1 Start-up

Here follows one example of the two sample sets from a measurement. The virtual values were retrieved using nominal simulation parameters.

<table>
<thead>
<tr>
<th>Physical time (ms)</th>
<th>Virtual time (ms)</th>
<th>Message id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0x01</td>
</tr>
<tr>
<td>20.3</td>
<td>20.00</td>
<td>0x02</td>
</tr>
<tr>
<td>29.7</td>
<td>29.41</td>
<td>0x03</td>
</tr>
<tr>
<td>53.3</td>
<td>53.04</td>
<td>0x04</td>
</tr>
<tr>
<td>468.4</td>
<td>501.01</td>
<td>0x05</td>
</tr>
<tr>
<td>497.7</td>
<td>529.48</td>
<td>0x06</td>
</tr>
<tr>
<td>527.2</td>
<td>559.39</td>
<td>0x07</td>
</tr>
<tr>
<td>532.3</td>
<td>564.08</td>
<td>0x04</td>
</tr>
<tr>
<td>558.0</td>
<td>589.39</td>
<td>0x07</td>
</tr>
<tr>
<td>589.5</td>
<td>620.66</td>
<td>0x06</td>
</tr>
<tr>
<td>979.8</td>
<td>1000.28</td>
<td>0x05</td>
</tr>
<tr>
<td>1051.1</td>
<td>1072.26</td>
<td>0x08</td>
</tr>
<tr>
<td>1080.5</td>
<td>1099.73</td>
<td>0x08</td>
</tr>
<tr>
<td>1111.0</td>
<td>1129.39</td>
<td>0x09</td>
</tr>
<tr>
<td>1141.9</td>
<td>1159.39</td>
<td>0x09</td>
</tr>
<tr>
<td>1174.9</td>
<td>1189.44</td>
<td>0x04</td>
</tr>
<tr>
<td>1200.8</td>
<td>1196.45</td>
<td>0x0A</td>
</tr>
</tbody>
</table>

B.2 Shutdown

Here follows one example of the two sample sets from a measurement. The virtual values were retrieved using nominal simulation parameters.
<table>
<thead>
<tr>
<th>Physical time (ms)</th>
<th>Virtual time (ms)</th>
<th>Message id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0x05</td>
</tr>
<tr>
<td>32.3</td>
<td>30.39</td>
<td>0x05</td>
</tr>
<tr>
<td>62.1</td>
<td>60.39</td>
<td>0x05</td>
</tr>
<tr>
<td>92.2</td>
<td>90.39</td>
<td>0x05</td>
</tr>
<tr>
<td>122.5</td>
<td>120.45</td>
<td>0x05</td>
</tr>
<tr>
<td>152.5</td>
<td>150.50</td>
<td>0x05</td>
</tr>
<tr>
<td>182.7</td>
<td>180.39</td>
<td>0x05</td>
</tr>
<tr>
<td>212.7</td>
<td>211.35</td>
<td>0x05</td>
</tr>
<tr>
<td>426.4</td>
<td>406.60</td>
<td>0x0B</td>
</tr>
<tr>
<td>490.7</td>
<td>470.90</td>
<td>0x0C</td>
</tr>
<tr>
<td>534.2</td>
<td>501.58</td>
<td>0x0D</td>
</tr>
<tr>
<td>537.6</td>
<td>508.01</td>
<td>0x0E</td>
</tr>
<tr>
<td>562.7</td>
<td>530.24</td>
<td>0x0F</td>
</tr>
<tr>
<td>567.1</td>
<td>534.90</td>
<td>0x10</td>
</tr>
<tr>
<td>927.8</td>
<td>908.20</td>
<td>0x0B</td>
</tr>
<tr>
<td>951.5</td>
<td>908.22</td>
<td>0x05</td>
</tr>
<tr>
<td>975.8</td>
<td>930.38</td>
<td>0x05</td>
</tr>
<tr>
<td>1006.0</td>
<td>960.42</td>
<td>0x05</td>
</tr>
<tr>
<td>1036.5</td>
<td>990.91</td>
<td>0x05</td>
</tr>
<tr>
<td>1067.2</td>
<td>1020.42</td>
<td>0x05</td>
</tr>
<tr>
<td>1096.3</td>
<td>1050.44</td>
<td>0x05</td>
</tr>
<tr>
<td>1126.7</td>
<td>1080.42</td>
<td>0x05</td>
</tr>
<tr>
<td>1156.7</td>
<td>1110.50</td>
<td>0x05</td>
</tr>
<tr>
<td>1429.2</td>
<td>1409.60</td>
<td>0x0B</td>
</tr>
<tr>
<td>1493.5</td>
<td>1473.90</td>
<td>0x0C</td>
</tr>
<tr>
<td>1538.9</td>
<td>1501.59</td>
<td>0x0D</td>
</tr>
<tr>
<td>1545.3</td>
<td>1508.01</td>
<td>0x0E</td>
</tr>
<tr>
<td>1566.9</td>
<td>1530.24</td>
<td>0x0F</td>
</tr>
<tr>
<td>1569.8</td>
<td>1534.90</td>
<td>0x10</td>
</tr>
<tr>
<td>1930.5</td>
<td>1911.20</td>
<td>0x0B</td>
</tr>
<tr>
<td>2341.8</td>
<td>2412.60</td>
<td>0x0B</td>
</tr>
<tr>
<td>2496.1</td>
<td>2476.90</td>
<td>0x0C</td>
</tr>
<tr>
<td>2544.1</td>
<td>2501.59</td>
<td>0x0D</td>
</tr>
<tr>
<td>2547.9</td>
<td>2508.02</td>
<td>0x0E</td>
</tr>
<tr>
<td>2571.0</td>
<td>2530.24</td>
<td>0x0F</td>
</tr>
<tr>
<td>2576.1</td>
<td>2534.90</td>
<td>0x10</td>
</tr>
<tr>
<td>2933.2</td>
<td>2914.20</td>
<td>0x0B</td>
</tr>
<tr>
<td>3434.5</td>
<td>3415.60</td>
<td>0x0B</td>
</tr>
<tr>
<td>3498.8</td>
<td>3479.90</td>
<td>0x0C</td>
</tr>
<tr>
<td>3546.6</td>
<td>3501.55</td>
<td>0x0D</td>
</tr>
<tr>
<td>3550.6</td>
<td>3507.97</td>
<td>0x0E</td>
</tr>
<tr>
<td>3575.2</td>
<td>3530.27</td>
<td>0x0F</td>
</tr>
<tr>
<td>3578.8</td>
<td>3534.93</td>
<td>0x10</td>
</tr>
<tr>
<td>3935.8</td>
<td>3917.20</td>
<td>0x0B</td>
</tr>
<tr>
<td>4437.2</td>
<td>4418.60</td>
<td>0x0B</td>
</tr>
<tr>
<td>4501.5</td>
<td>4482.90</td>
<td>0x0C</td>
</tr>
<tr>
<td>4551.3</td>
<td>4501.64</td>
<td>0x0D</td>
</tr>
<tr>
<td>4556.3</td>
<td>4506.30</td>
<td>0x0E</td>
</tr>
<tr>
<td>4579.3</td>
<td>4530.27</td>
<td>0x0F</td>
</tr>
<tr>
<td>4583.4</td>
<td>4536.70</td>
<td>0x10</td>
</tr>
<tr>
<td>4938.6</td>
<td>4920.20</td>
<td>0x0B</td>
</tr>
<tr>
<td>5439.9</td>
<td>5421.60</td>
<td>0x0B</td>
</tr>
<tr>
<td>5504.1</td>
<td>5485.90</td>
<td>0x0C</td>
</tr>
<tr>
<td>5555.5</td>
<td>5501.60</td>
<td>0x0D</td>
</tr>
<tr>
<td>5560.9</td>
<td>5508.02</td>
<td>0x0E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>58</td>
<td>5583.4</td>
<td>5530.27</td>
</tr>
<tr>
<td>59</td>
<td>5589.1</td>
<td>5534.93</td>
</tr>
<tr>
<td>60</td>
<td>5941.2</td>
<td>5923.20</td>
</tr>
<tr>
<td>61</td>
<td>6442.6</td>
<td>6424.60</td>
</tr>
<tr>
<td>62</td>
<td>6506.9</td>
<td>6488.90</td>
</tr>
<tr>
<td>63</td>
<td>6560.1</td>
<td>6502.91</td>
</tr>
<tr>
<td>64</td>
<td>6563.6</td>
<td>6507.57</td>
</tr>
<tr>
<td>65</td>
<td>6587.6</td>
<td>6530.27</td>
</tr>
<tr>
<td>66</td>
<td>6591.7</td>
<td>6536.70</td>
</tr>
<tr>
<td>67</td>
<td>6943.9</td>
<td>6926.20</td>
</tr>
<tr>
<td>68</td>
<td>7445.2</td>
<td>7427.60</td>
</tr>
<tr>
<td>69</td>
<td>7509.5</td>
<td>7491.90</td>
</tr>
<tr>
<td>70</td>
<td>7522.6</td>
<td>7560.37</td>
</tr>
<tr>
<td>71</td>
<td>7578.5</td>
<td>7594.97</td>
</tr>
<tr>
<td>72</td>
<td>7621.7</td>
<td>7620.22</td>
</tr>
<tr>
<td>73</td>
<td>7653.2</td>
<td>7650.46</td>
</tr>
<tr>
<td>74</td>
<td>7682.4</td>
<td>7680.28</td>
</tr>
</tbody>
</table>
Bibliography


Upphovsrätt
Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.
Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.
Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.
För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright
The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.
The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on 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/

© Adam Sandstedt