Diploma thesis

'Benchmark and comparison of real-time solutions based on embedded Linux'

Submitted in partial satisfaction of the requirements for the degree of
'Diplom Ingenieur (FH) der technischen Informatik'
at Hochschule Ulm

Peter Feuerer
July 30, 2007

Caretaker:

HS Ulm: Prof. Dr.- Ing. Schied
Yellowstone-Soft: Dipl.- Ing. Betz

Preface

Eigenständigkeitserklärung (German)

Ich versichere, dass ich die vorliegende Diplomarbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

.....
Ort, Datum, Unterschrift

Abstract

This diploma thesis gives an overview about current available real-time Linux approaches and deals with creation of a test environment to compare them to each other. The comparison is done with an abstraction layer as a standardized base and includes qualitative as well as quantitative benchmarks.
Furthermore every benchmark aims to give reproducible results from a very practical point of view. Thus the outcome of the benchmarks can be directly used by clients who order a real-time embedded system for choosing the platform which fits best their needs.

Acknowledgments

I want to thank all people who made this diploma thesis possible, while my special thanks go to:

My family and friends for supporting me in any matter and for assisting by word and deed in stressful days.

Prof. Dr.- Ing. Schied for supervising me while creation of the diploma thesis and for giving helpful hints to improve the documentation.

Dipl.- Ing. Betz for supervising and offering technical experience and knowledge which was important for finishing this thesis.

Patrick Reinwald for giving support for the PowerPC architecture.

Linux community for working so hard on the open source operating system, the real-time approaches and its components. Many thanks to Bernhard Kuhn, Thomas Gleixner, Wolfgang Denk, Wolfgang Grandegger and many more who responded to my emails and helped to get the things working.

Contents

Preface **I**

1. Introduction **1**
1.1. Motivation . . . 1
1.2. About the document . . . 2

2. State of the art **3**
2.1. Linux . . . 3
2.2. Real-time solutions . . . 3
2.2.1. Rtai-Linux . . . 5
2.2.2. Xenomai . . . 6
2.2.3. Real-time Preemption Patch . . . 7
2.3. Hardware . . . 8
2.3.1. Intel x86 . . . 8
2.3.2. ARM . . . 9
2.3.3. PowerPC . . . 10
2.4. Measurement hardware - Meilhaus Scope . . . 11
2.5. Software . . . 12
2.5.1. ORF - Open Realtime Framework . . . 12
2.5.2. SofCoS . . . 16
2.5.3. Coryo . . . 16

3. Preparations **19**
3.1. Linux development environment . . . 19
3.2. Windows development environment . . . 19
3.3. Toolchain installation . . . 21
3.3.1. Intel x86 toolchain . . . 21
3.3.2. ARM toolchain . . . 23
3.3.3. PowerPC toolchain . . . 23
3.4. Target setup . . . 25
3.4.1. Intel x86 target . . . 25
3.4.2. ARM target . . . 31
3.4.3. PowerPC target . . . 32
3.5. ORF implementations . . . 35
3.5.1. Dynamical loaded libraries . . . 36
3.5.2. Character devices . . . 38
3.5.3. I/O-API . . . 44
3.5.4. Interrupt handling . . . 46

4. Benchmarks **51**
4.1. Interrupt latency . . . 51
4.1.1. ORF integration . . . 52
4.1.2. Scope implementation . . . 53
4.2. Jitter . . . 55
4.2.1. ORF integration . . . 55
4.2.2. Scope implementation . . . 56
4.3. Maximal frequency . . . 57
4.3.1. ORF integration . . . 59
4.3.2. Scope implementation . . . 60
4.4. Inter-process communication . . . 63
4.4.1. ORF integration . . . 63
4.5. Overload behavior . . . 65
4.5.1. ORF integration . . . 65
4.5.2. Scope implementation . . . 66
4.6. Priority functionality . . . 68
4.6.1. ORF integration . . . 68
4.6.2. Scope implementation . . . 69

5. Results **73**
5.1. Frequency . . . 74
5.2. Interrupt latency . . . 74
5.3. Inter process communication . . . 76
5.4. Jitter . . . 77
5.5. Overload . . . 78
5.6. Priority . . . 79

6. Conclusion **81**

A. Bibliography **83**

B. Glossary **87**

C. Listings **89**

D. License **93**

List of Figures

2.1. Utility / costs - function of hard real-time . . . 4
2.2. Utility / costs - function of soft real-time . . . 5
2.3. Rtai Linux architecture . . . 6
2.4. Kernel preemption . . . 8
2.5. Kontron - embedded Geode system . . . 9
2.6. Incostartec's ep9315 distribution board . . . 10
2.7. Frenco's MEG32 embedded system . . . 10
2.8. Meilhaus Mephisto Scope . . . 11
2.9. ORF as an abstraction layer . . . 12
2.10. ORF's architecture . . . 14
2.11. Coryo user interface . . . 17
2.12. OrfCoS user interface . . . 17

3.1. User interface of wxDev-C++ . . . 20
3.2. Principle x86 toolchain architecture . . . 21
3.3. Loading and unloading shared objects . . . 37
3.4. Flowchart of changes to enable dynamical loaded libraries . . . 39
3.5. Communication between user-space and ORF using character devices . . . 42
3.6. Calls of I/O-API functions . . . 45
3.7. Interrupt device approach . . . 46
3.8. Interrupt thread approach . . . 46
3.9. Interrupt handling - modifications to thread . . . 48
3.10. Interrupt handling - modifications to RProg . . . 49

4.1. Interrupt latency scope graph . . . 53
4.2. Flowchart, scope implementation of interrupt latency measurement . . . 54
4.3. Jitter scope graph . . . 56
4.4. Control flow of jitter implementation . . . 58
4.5. Scope graph of frequency benchmark . . . 61
4.6. Control flow of the frequency benchmark . . . 62
4.7. Flow diagram of the echo test function . . . 64
4.8. Principle graph of overload test 1 . . . 65
4.9. Principle graph of overload test 2 . . . 65
4.10. Scope graph of the overload test . . . 66
4.11. Flow diagram of overload's scope application . . . 67
4.12. Graph of the priority test . . . 69
4.13. Flow of the priority benchmark . . . 71
4.14. Flow diagram of the preemption check . . . 71

5.1. Results of the frequency benchmark 74
5.2. Interrupt latency of Intel x86 architecture with Linux 2.6 and Xenomai 75
5.3. Interrupt latency of x86 architecture with Linux 2.6 and Rtai 75
5.4. Interrupt latency of Rtai on a x86 target with Linux 2.4 75
5.5. Results of inter-process communication benchmark 76
5.6. Jitter benchmark - PowerPC vs. Intel x86 77
5.7. Jitter benchmark - Geode gx1 system 78

Chapter 1.

Introduction

1.1. Motivation

To satisfy all requirements for the German degree of "Diplom Ingenieur (FH) der technischen Informatik" submission of a final thesis is needed. This work is such a final thesis and was elaborated at Yellowstone-Soft company Ehingen in southern Germany. The project is about benchmarking and comparing different real-time solutions based on Linux.

Real-time operating systems are getting more and more important for different uses in industry and Linux made good process in becoming a full hard real-time operating system, especially in the last few years. Due to the *GPL* under which Linux is licensed, companies don't have to pay the high licensing fees which are very common for real-time operating systems. But Linux itself does not yet meet all the requirements of a hard real-time OS. That's why there are several additions to add real-time functionality to Linux. Currently the three most popular approaches are Rtai, Xenomai and the RT-Preempt patch.
Engineers who develop embedded systems with real-time usage do have a new challenge, besides choosing the hardware platform for their project. They have to evaluate which real-time approach should be used. Therefor different aspects of a real-time operating system are important. For example the interrupt response time, data transfer rate of inter-process communication, behavior under overload and many more.

In this diploma thesis a test environment is created and the three mentioned real-time approaches are tested and benchmarked on such important attributes. Additionally the three most popular embedded platforms, Intel x86, ARM and PowerPC are also be compared to each other in their real-time ability. The Open Realtime Framework, developed by Yellowstone-Soft is used as base for all tests and benchmarks. It offers a high level API for all necessary real-time functions and is designed to being portable for all kinds of real-time approaches and hardware platforms.

1.2. About the document

This thesis combines the world of control engineering with the world of software development, what brings the problem that there are readers who may don't have deep knowledge in both areas. Thus the keywords of both worlds are explained in the glossary so that everybody should be able to understand the work. Those keywords are shown in an *italic* font.
To refer to an item of the bibliography box brackets and numbers are used. For example referring to the first item looks like that [1].

The first chapter aims to give an introduction and a short overview about the work.
The second chapter of this document contains information about the technology needed to accomplish this project. It includes general information about what characteristics a real-time operating system must fulfill, functionality and assembly of the used hardware and some data about the software which relate in a direct way with the project.
One of the main parts of this work is setting up development environments, creating and installing toolchains for different target systems. Additionally some modifications on the ORF system had to be done. Those tasks are described in chapter three.
Chapter four deals about the ideas and realization of the benchmarks and tests which results are argued in chapter five.
The last chapter contains a summary about the whole work, ideas and where this project can lead to in the future.

The appendix includes a bibliography, a glossary and some important code listings.

Chapter 2.

State of the art

This chapter includes basic knowledge about the technology on which the work of this thesis depends. It gives a short overview about what Linux is, the basic idea behind real-time processing and the abstraction layer which is used for running the benchmarks.

2.1. Linux

GNU/Linux is an open source implementation of the Unix operating system which has been completely rewritten and published under the GPL. Linux itself is just the kernel although most people mean the whole operating system including the GNU programs and tools when speaking of Linux.
In 1991 Linus Torvalds, a Finnish student, published the first version of the kernel as a non-commercial replacement for the *minix* system. The open source enthusiasts under Richard Stallman who were programming on the GNU system had already made an open source replacement for nearly every tool of a full Unix system, only the kernel was missing. So Linus Torvalds and the GNU people worked together to bind GNU and Linux to get a completely open sourced operating system.
The community around Linux and the GNU system was growing extremely and the success story of this OS was unstoppable. In present Linux is one of the most used operating systems and due to the license it gets more and more used for things Linux was never meant to be used for. It is highly platform independent and has been ported to nearly every hardware out there.

2.2. Real-time solutions

Most people think of extreme fast computer hardware, when they hear the term real-time computing. But that's wrong, a real-time system must not be a high end system, most often

the opposite is true. Real-time systems are commonly used for *embedded* projects and in the embedded world reliability and determinism counts much more than gigahertz or the amount of ram the system has. These attributes can be achieved by using especially designed hardware and a real-time operating system on top of it.

So what does real-time mean? Real-time means that there are specified timings, and absolutely deterministic behavior. For example an airbag system, it is designed to open the airbag before the head of the driver hits the steering wheel. The system is not allowed to stall the execution of the airbag opening procedure just because the CPU is currently decoding mp3 music. Besides the fact, that in a real car the airbag control system and the multimedia functionality are strictly separated, would it be possible to realize such this combination on one physical computer with a real-time operating system. The task which is needed for playing music gets lowest priority and the sensor in the front of the car has an interrupt with highest priority. As soon as the interrupt appears, the music playing task will be stopped and the interrupt routine which opens the airbag gets full CPU time.

There are two major categories of real-time definitions, hard real-time and soft real-time.

Hard real-time

This real-time definition is the most complicated one to achieve. It defines that every deadline must be strictly adhered. If one deadline missed, it will cost money or could even harm people.
An environment that needs hard real-time is for example a laser welding machine. Every timing must fit exactly for this application. If the computer takes too long to react on an event the workpiece will be damaged and will have to be trashed. Following graph shows the gain depending on time of the hard real-time.

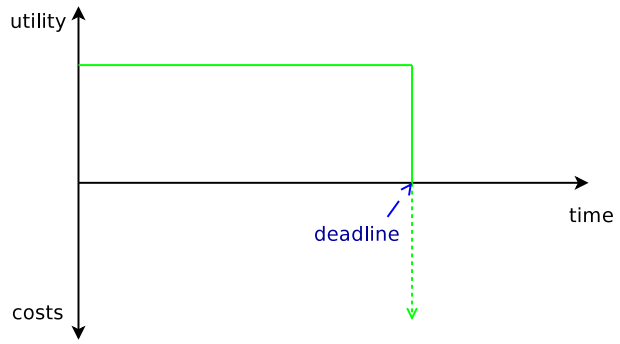

Figure 2.1.: Utility / costs - function of hard real-time

Soft real-time

This is the weakest real-time definition, and says just, that it would be great if the deadlines are adhered because then the gain would be maximum. But there is still some gain, even after the deadline elapsed. Soft real-time can be achieved with a normal personal computer with any mainstream operating system like Windows, Mac OS, or normal Linux. It is used for example in automation of uncritical processes like closing the shutters of the windows of a house. For such an usage it does not matter whether the shutter are closed with a delay of some second or even minutes.

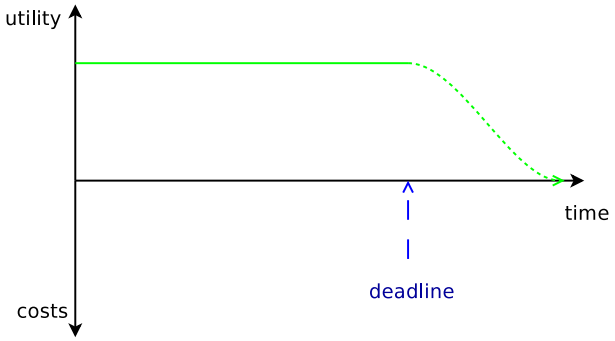

Figure 2.2.: Utility / costs - function of soft real-time

2.2.1. Rtai-Linux

Rtai-Linux[1] was one of the first approaches to enhance Linux by real-time functionality. The idea behind Rtai is to have a dual kernel system. One very minimalistic Rtai kernel and the normal Linux kernel. The Rtai kernel consists of a real-time scheduler with priority based scheduling and a hardware abstraction layer to enable interrupt handling while still taking care about the high priority real-time tasks. The Linux kernel has just to be modified to run not directly on the real hardware but on the hardware abstraction layer of the minimalistic kernel.

The minimalistic kernel runs the Linux kernel within its idle task, thus Linux is running with lowest priority and gets the interrupts from the Rtai kernel after they were handled by it. Rtai offers a special API for programming real-time applications so it is not possible to turn a normal Linux application into a real-time application just by recompiling. Additionally the real-time tasks cannot be run in user-space of Linux, they must be compiled as a Linux kernel.
Within the Linux kernel environment the Rtai API can be used to create real-time threads, set the periodicity, the scheduling or the priority of such threads. When loading a compiled Rtai kernel module into the Linux kernel, the Rtai API calls are then passed through the

Linux kernel to the Rtai kernel, which handles them and creates for example a new thread with real-time ability. Figure 2.3 shows this behavior.

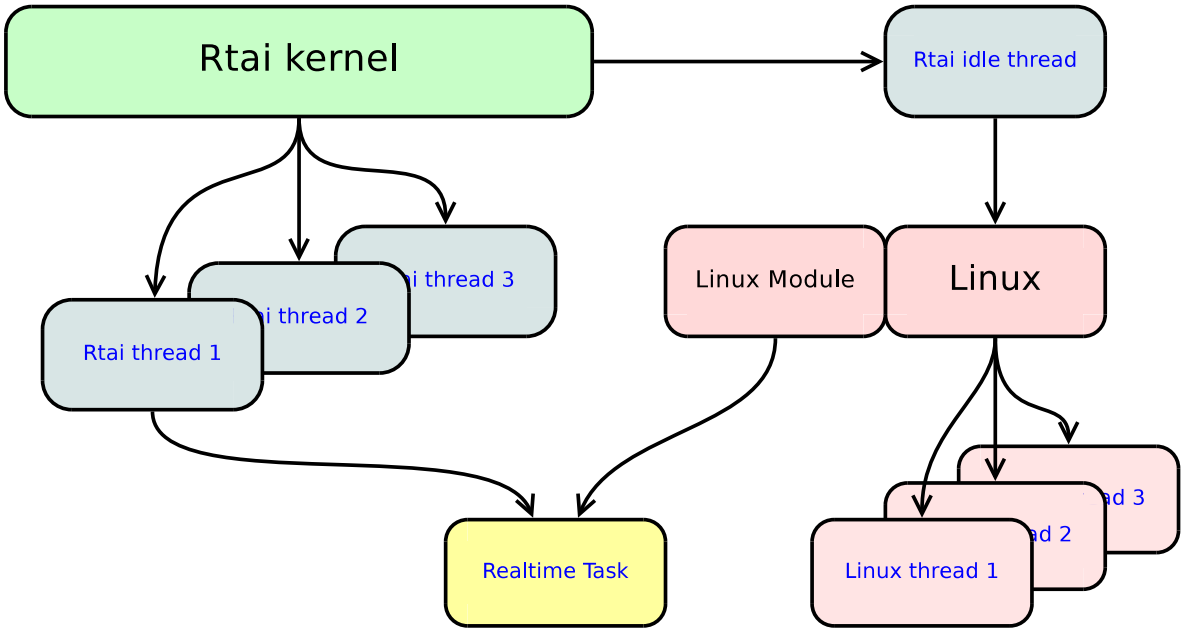

Figure 2.3.: Rtai Linux architecture

2.2.2. Xenomai

The Xenomai project[3] was started in 2001 with the idea of providing an open source alternative for industrial applications ported from the proprietary world. It is, like Rtai, based on the dual kernel approach. That and some other similarities to Rtai made it possible to combine those two projects. The resulting Rtai/Fusion project existed for about 2 years, until the people working on Xenomai decided to work independently from Rtai again.
Xenomai is now focusing on so called skins. A Xenomai skin is used to easily migrate real-time programs from another real-time approach or real-time operating system to Xenomai, e.g. the Rtai skin offers an API which behaves like Rtai, so that nearly nothing has to be done to port an existing Rtai application to Xenomai. There are much more skins, like Posix, VxWorks.
One of the major issues of the dual kernel approach are that the real-time programs must be created as a Linux kernel module which cannot easy be debugged using the *gdb* and even worse they can easily crash the kernel because there is no memory protection. That's why they implemented an user-space API in their native skin what allows to write real-time programs running in user-space.
The main differences between Rtai and Xenomai are that Rtai aims to offer lowest possible

latencies and Xenomai focuses mainly on portability, extensibility and maintainability. In the future the Xenomai project wants not only to base on the dual kernel technology, but also to support the real-time preemption patch for the normal kernel.

2.2.3. Real-time Preemption Patch

This real-time solution[6] is a patch for vanilla 2.6 Linux kernel to enable hard real-time within the normal Linux kernel. So there is no additional API needed and the real-time programs don't have to be compiled as kernel-modules. They can be started like normal user-space programs what brings following advantages:

- Debugging possible via gdb
- Memory protection
- Non-real-time programs can easily be ported to fulfill real-time requirements
- As soon as the patch is completely merged into the vanilla kernel, nothing needs to be patched or installed

Unix-legacy operating systems and also Linux were not meant to be used for real-time applications. They were designed for offering high throughput and progress using a fair scheduling. Thus the way to get a deterministic hard real-time capable Linux kernel was not easy and Ingo Molnar has been working very hard to achieve this goal. First of all the kernel had to be made more preemptive that it already was, therefore preemptible mutex with priority inheritance (*PI mutex*) were implemented into the kernel. Then every *big kernel lock*, *spinlock*, *read-write lock*, has been converted to use a PI mutex. Figure 2.4 shows the difference in preemtible code between vanilla Linux kernel 2.6 and the rt-preempt patched kernel. But replacing the locks is not enough, a new way of handling interrupts was necessary. The former way to handle interrupts was to directly call the interrupt service routine as soon as the interrupt occurs, that's a great thing for non real-time systems. But if deterministic behavior is needed, there must not be an interrupt which preempts a real-time task for an unpredictable time period.
To solve this issue, interrupt handling threads were introduced. The former interrupt service routine was replaced by deterministic interrupt service routines that just wake up their corresponding interrupt thread. This interrupt thread runs with a specified priority and can be preempted by higher prioritized tasks. This way it is possible to have unpredictable interrupt handling while still offering hard real-time. The low light of this solution appears when an interrupt occurs and the real-time task needs full CPU time for a longer timer period, then the interrupt latency becomes very high.
Another important part of the preemption patch is the high resolution timer, implemented by Thomas Gleixner. This timer takes care about precise high frequency timings like needed for doing a nanosleep. Due to differing timer hardware on every platform this code is not

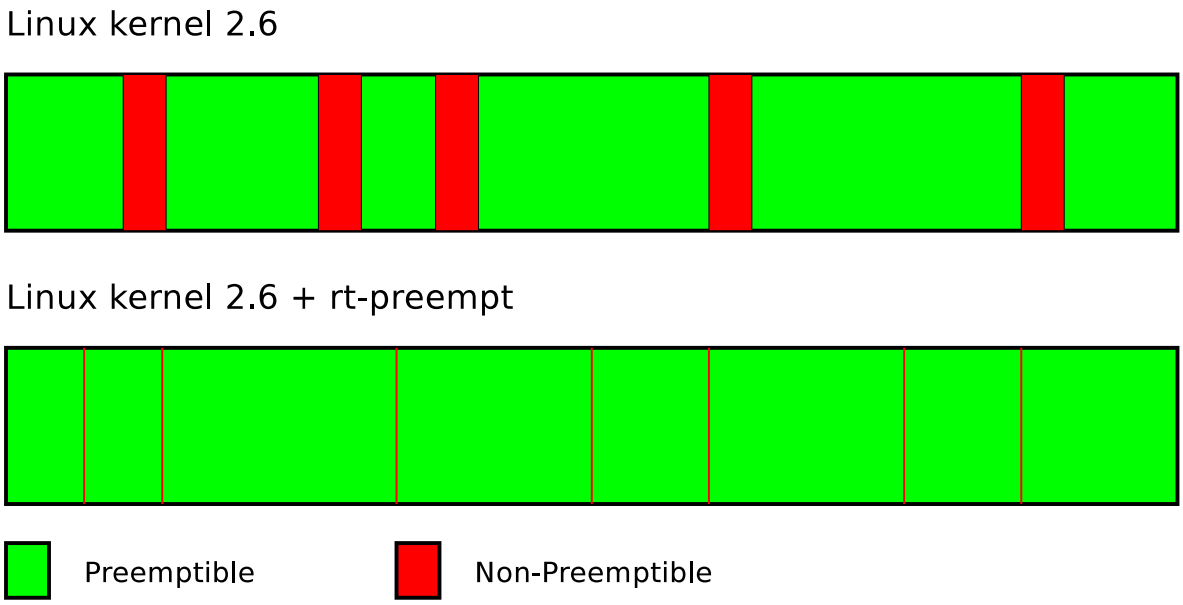

Figure 2.4.: Kernel preemption

hardware independent. It has to be ported to the target hardware before the rt-preemption patch will work correctly.

2.3. Hardware

This section contains information about the hardware that is used for the thesis. The hardware can be split up into two parts, the development environment hardware and the hardware which is used as real-time embedded system. As development environment normal Intel x86 PC's do their duty, so nothing special here. The target systems are more unique that's why they are listed separately in this section.

2.3.1. Intel x86

Two different systems are used for covering the Intel x86 part. A standard desktop computer with an AMD K7 processor running at 600MHz, 128MiB of ram and a normal IDE hard disk. This system has been chosen because it is some kind of a standard system and most people are familiar with it. So it can be taken as a test environment and of course as a reference.
The second Intel x86 target is an embedded system from Kontron. This computer is a very common embedded system and comes with a National Semiconductor Geode gx1 32-Bit processor clocked at 300MHz, 128MiB of ram and a compact flash to IDE adapter. Thus the operating system and additional programs are stored on a compact flash card. As the

complete hardware is designed for embedded purposes it is much more reliable and robust than a standard x86 desktop computer.

Figure 2.5.: Kontron - embedded Geode system

2.3.2. ARM

As ARM platform the "LILLY-9xx" board from Incostartec is used. The board is a redistribution of the ep9315[8] *SOC* processor of Cirrus Logic which has an 32-Bit ARM920T processor clocked at 200MHz and contains several additional hardware like a network controller, a video controller and an IDE controller. On the mainboard are 32MiB of ram soldered. These attributes and the available *MMU* enables the SOC processor to run embedded versions of Windows and Linux.
To boot up the operating system redboot[9] comes preinstalled on this target. Redboot is a bootloader especially for embedded architectures and supports downloading of the operating system via various protocols over network and a serial connection as well as booting from flash memory.
The IDE controller accesses directly a compact flash card, which can be mounted on top of the controller. In summary the complete system, without connectors, fits on a board of about 5x7 centimeters. Furthermore it consumes very less energy and must not be cooled. So the combination of less space usage and less energy consumption makes this board to a very effective and beneficial embedded solution.

Figure 2.6.: Incostartec's ep9315 distribution board

2.3.3. PowerPC

The PowerPC section is covered using the MEG32 as target. MEG32 is an embedded system developed especially for measurement tasks by the companies Frenco, Eckart GmbH and Gall EDV Systeme GmbH. The complete system consists of a 19" rack with 15 plug-in slots. This makes MEG32 very modular and allows many different use-cases. Usually the main board is plugged into the first slot. It contains the 32-Bit PowerPC G2 which is clocked at 300MHz, 128MiB of ram and a flash chip with 32MiB.

Figure 2.7.: Frenco's MEG32 embedded system

2.4. Measurement hardware - Meilhaus Scope

The Mephisto UM202 scope[10] from Meilhaus is a combination of oscilloscope, logic analyzer and data-logger which can be connected to any Windows driven PC via USB. The software that comes with the scope enables the same operational area as normal oscilloscopes or logic analyzer have. But the big advantage of this scope is that it comes additionally with a programmable C-API. Using this API it is possible to expand the operational area by many use-cases because the values of the measured lines are stored in the scopes memory and can be directly read and computed in C.
In oscilloscope mode it has two input lines, and both lines value's are represented in a 16 Bit wide variable. Its timebase can be set from 1μs to 1s and it can store up to 50000 values per channel per measurement. That results in maximal 50ms data when the timebase is set to 1μs, what's way enough and very precise for the tests and benchmarks described in chapter 4. The data-logger and digital logic analyzer modes are slower and the shortest timebase available for these two modes is 10μs but the data-logger has a very important advantage; it can stream the data. This way it can acquire much more data than the scope can store in its memory.

Figure 2.8.: Meilhaus Mephisto Scope

2.5. Software

The software which is used or is very closely related to the project of this thesis are described in this section. It handles mainly about the Open Realtime Framework, what is used to have a base for all platforms to ensure comparable results. Additionally some tools to work with ORF are explained.

2.5.1. ORF - Open Realtime Framework

Principle functionality

The Open Realtime Framework[12] is an open source project developed by Yellowstone-Soft which aims to offer a standardized API for real-time applications. It is from scratch designed for being very platform independent and portable. That's why it is very easy to enhance ORF to run on a new hardware platform or another real-time approach.
The current state of ORF is compiling just for Linux yet, but it should not be a big task to compile it for Windows or any other operating system. Under Linux it can be compiled to run in user-space or in kernel-space as kernel-modules. The principle functionality of ORF is shown on figure 2.9.

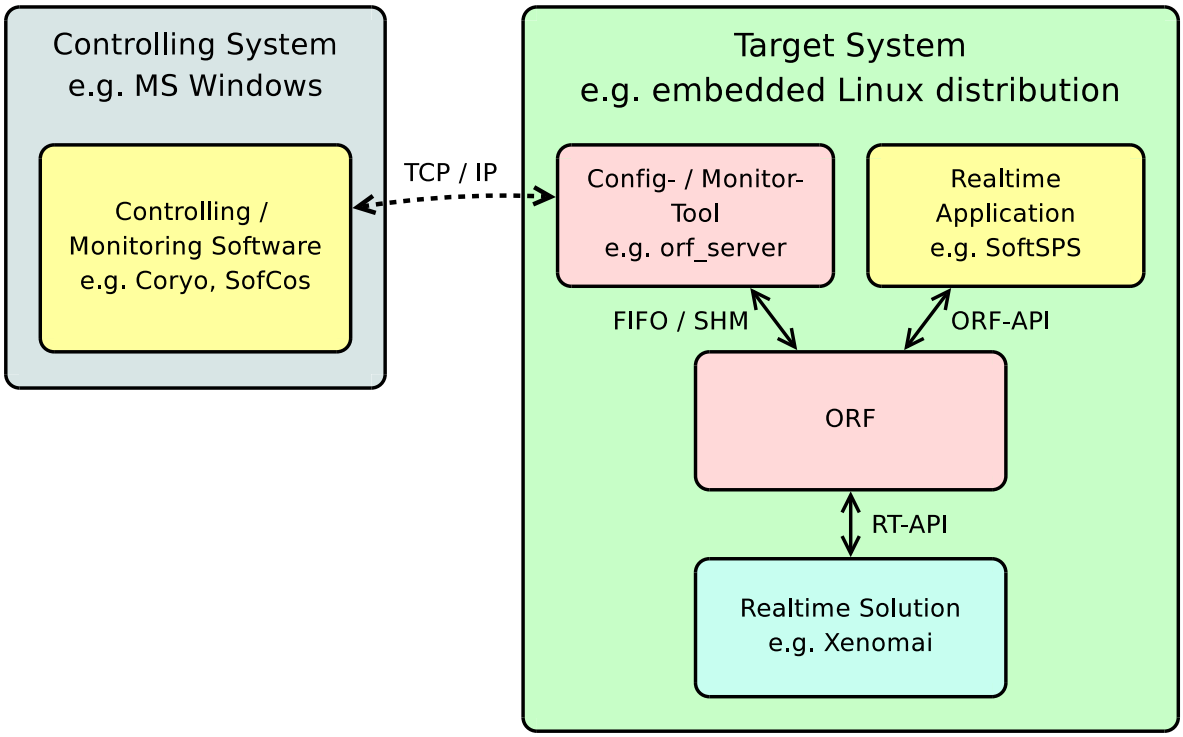

Figure 2.9.: ORF as an abstraction layer

- ORF communicates with the real-time solution using the API offered by the real-time solution.
- ORF has FIFO files, which are used for setting up, controlling and monitoring the complete environment. There are several tools which can be used to communicate through the fifos. Two of the most used tools are `orf_server` and `orf_startup`. `orf_startup` is built for the initial setup of ORF, it loads a ".ini" file and passes the ORF commands in it to ORF. The `orf_server` utility however is used to monitor and control the running ORF environment. It opens a TCP/IP server to which another tool can be connected and send commands or data.
- ORF offers an API which allows hardware platform and real-time solution independent programming of real-time applications.

Architecture

The architecture of ORF is focused on cyclical working sequential controls like it is true for a *PLC* of Siemens, e.g. a S7 PLC. ORF consists of following parts:

PLC: The whole ORF runtime image is called PLC and includes everything listed here.

Device: A device stands for one sequential control in the PLC. There can be more than just one device in a PLC and they are working completely independent from each other. Every device has its very own shared memory to which no other device has access. This ensures data consistence.

Page: A shared memory region which is assigned to a device is called page. Any program of a device can store variables, data and debug information in this memory region. There is one special shared memory page, called "Zero-Page" which holds data about the state of the complete PLC.

Thread: Every device has a so called "Thread0" in which all programs of a device are processed sequential. That means, when a device contains more than one program, Thread0 calls the first program and as soon as the program has reached its end, the next program is called. Due to the fact, that the PLC is mostly running on a single CPU computer the threads of the different devices can preempt each other. To ensure that only a more important task interrupts another task the threads get a priority assigned.

Program: An ORF module which uses the ORF API and does some of the real-time applications work is named program in ORF context. There are two different kinds of programs, the non-real-time programs "UProgs" and the real-time porgrams "RProgs". The UProgs are not interesting from this project's point of view, that's why they are not mentioned in the rest of the document anymore. The RProgs can use the shared

memory page of the device for whatever they want to and due to not existing parallelism within one device they don't need to take care about mutex. When writing a RProg the programmer must comply the RProg structure very strictly to keep the full platform independence ORF offers. A RProg consists in general of at least 4 functions:

- An init_module function which is called by the kernel when the RProg module is linked into the kernel. It is used to register the RProg to the ORF system.
- An init function which is called by ORF when ORF gets the command to execute its known init functions.
- A main function containing the code which is executed when the RProg is called by the thread0.
- And a cleanup_module function which cares about unregistering the RProg from ORF, when the module is removed from the kernel.

Figure 2.10 shows the cohesion between those 5 elements.

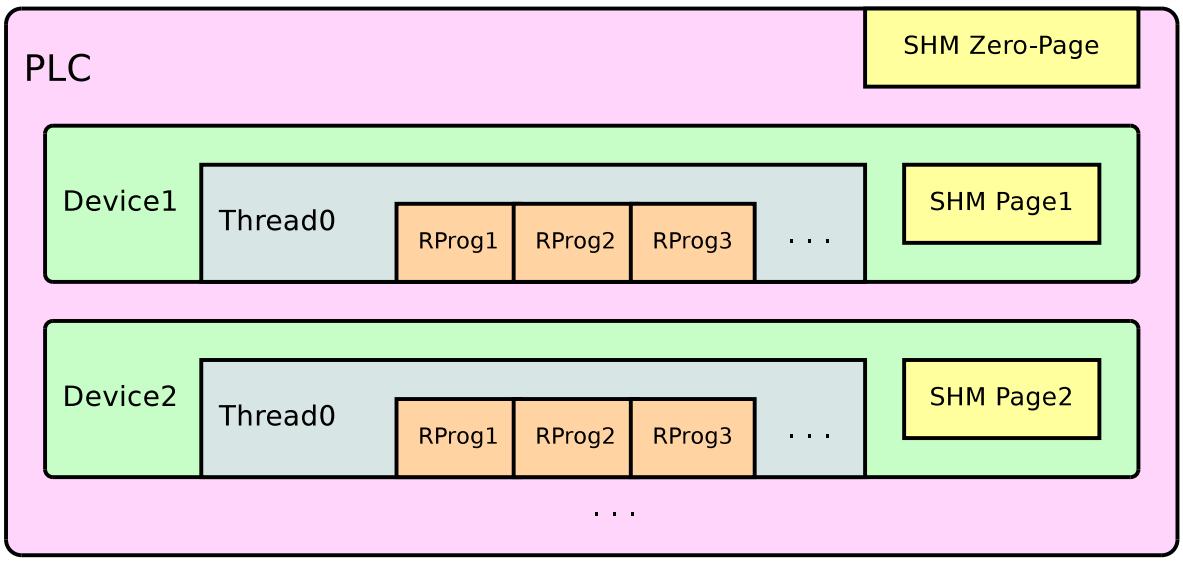

Figure 2.10.: ORF's architecture

Simple use-case

The functionality of ORF can easily be explained by an example. In this case the example just contains one RProg which toggles the state of the first pin on the I/O port. It should run with highest priority and with a periodic shot every 100ms.

RProg: The RProg's main function which is called when the program is started contains just these lines:

Listing 2.1: RProg.c

```
int toggle_io(int device,int id,long para)
{
    /* read byte from I/O and XOR it with 1
     * then output it on I/O */
    orf_outb(orf_inb()^0x1);
    return 0;
}
```

A complete example of a RProg can be found in Appendix C.1

Init.ini: To initialize ORF, add the RProg to the runtime image and start the thread0 with the correct period and highest priority following init file must be processed by orf_startup:

Listing 2.2: Init.ini

```
# [...] ORF default init stuff
# execute all initfunctions of loaded rprog modules
ORF_DO_INITFUNCT;*

# Create Thread 0 for Page 0 with priority 5 and periodic with
# 0x186A0 us = 100ms cycle duration
ORF_CREATE_THREAD0;0;5;186A0;1

# Create Realtime Progs
# start registerd prog TOGGLE_IO on thread0
ORF_CREATE_RPROG;0;1;0;TOGGLE_IO;;;

# Start PLC
ORF_START_PLC;0
```

A complete example of an Init.ini file can be found in Appendix C.2. And an overview about supported ORF commands is included in the ORF specification[13].

Startup: For this example a Xenomai environment is used, so ORF is running in kernel-space and everything has to be compiled as a kernel module. These kernel modules must then be loaded in a given order:

1. `insmod orf_methods_real.ko` - this module contains the API functions of ORF.
2. `insmod krn_orf.ko` - it contains the functions needed for handling requests through the pipes.
3. `insmod RProg.ko` - the module compiled of the RProg code above.

After all modules are loaded ORF can be initialized using the command line `orf_startup Init.ini`.
That's it, now ORF is running and the RProg which toggles the first pin of the I/O port is launched every 100ms. With hard real-time guarantees and reliability the chosen platform offers.

2.5.2. SofCoS

SofCoS[14] is a Software PLC under a commercial license that shows the same usability as a normal hardware PLC like Siemens S7. In the past it was designed to directly run on the target system but when the ORF project was started SofCoS has been altered to run as a RProg within ORF. By now SofCoS is one of the main applications which use ORF in industrial automation. It interprets and processes platform independent SofCoS binaries and supports all PLC function blocks defined in *IEC 61131-3* standard.

2.5.3. Coryo

Coryo[15] is a closed source product of Yellowstone-Soft. It provides a programming GUI to create applications for embedded systems or PLC's like the SofCoS PLC in many different programming languages.
An application can be written using the graphical Function Block Diagram (*FBD*), the textual languages Structured text (*ST*) or Instruction List (*IL*). Those languages are based on the sequential control principle which is a very common way of thinking for electronical engineers but very different from the way a computer engineer thinks. Computer engineers think in loops and functions and to meet their demands Coryo can also compile code snippets in C or C++. Coryo can directly connect to a SofCoS PLC running on an ORF environment and download the compiled code without restarting the target. Different debug modes enable an ease and efficient way of finding and fixing bugs.
The Coryo package contains some more important utilities e.g. "OrfCoS". It can connect to the TCP/IP server the `orf_server` opens on the target system and visualize the complete state of the PLC. It can start, stop or block the ORF PLC, control the state of all devices

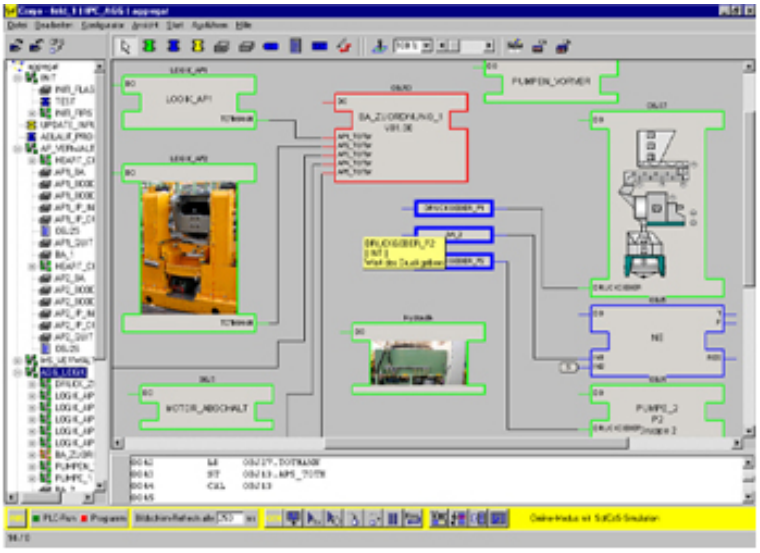

Figure 2.11.: Coryo user interface

in ORF and additionally grants read and write access an all shared memory pages. Thus "OrfCoS" is very useful for debugging ORF.

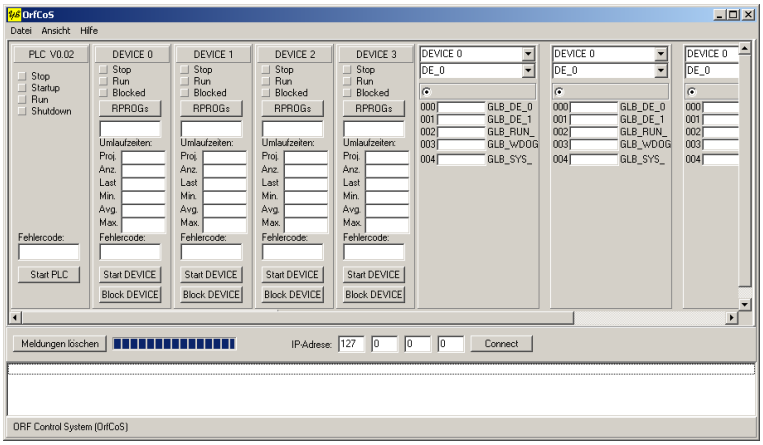

Figure 2.12.: OrfCoS user interface

Chapter 3.

Preparations

This chapter handles about setting up a Linux environment to develop ORF, a Windows environment for working on the scope's applications and about installing toolchains to compile Linux and ORF for the targets. The last part of the chapter contains information about the modifications which were made on ORF to offer the features needed for benchmarking.
When setting up just a testing environment it is enough to install the particular toolchain and the target itself. The measurment applications are compiled to Windows executables and the modifications of ORF are already implemented, so it must be just compiled for the target.

3.1. Linux development environment

Due to the ability of ORF of running as a normal program in user-space it makes sense to implement the new features and test it in user-space, because then it runs within its own virtual memory and cannot crash the kernel in case of a bug. Therefore a Linux development environment on one of the Intel x86 based desktop computers is needed.
A standard OpenSUSE 10.1 installation containing vim, the gnu compiler suite gcc version 4.0 and the makefile tools are enough to work on ORF.

3.2. Windows development environment

As already mentioned in section 2.4 is there only a Windows driver for this usb scope and the original measurement software is running under Windows only too. Thus a Windows system for programming the measurement applications using the scope's C-API is needed. Windows 2000 was chosen because it is running with very high performance within the virtualbox[11] virtualization software.
For programming C-programs under windows there exist a great open source development suite called "wxDev-C++"[17]. It combines following open source projects to an easy to install and use environment:

- DevC++[18] a C/C++ Windows *IDE* written in Delphi. It provides a visual GUI editor and can use different compilers as back-end.

- wxWindows[19] - a cross-platform GUI toolkit. It offers an API for writing platform independent GUI applications. But the API has become more than just a GUI toolkit, it gives things like network programming, file access programming and much more.

- MinGW[20] Gnu compiler collection, the windows port of the gcc compiler, gdb and other tools which are running natively under Windows and can create native Windows executeables. That's different from cygwin gcc which uses some kind of wrapper library to translate Unix syscalls to Windows syscalls.

In summary wxDev-C++ has the look and feel of the Borland C++ Builder and provides a very intuitive and stable IDE for free.
The next step is compiling the scope example programs of Meilhaus and link them against

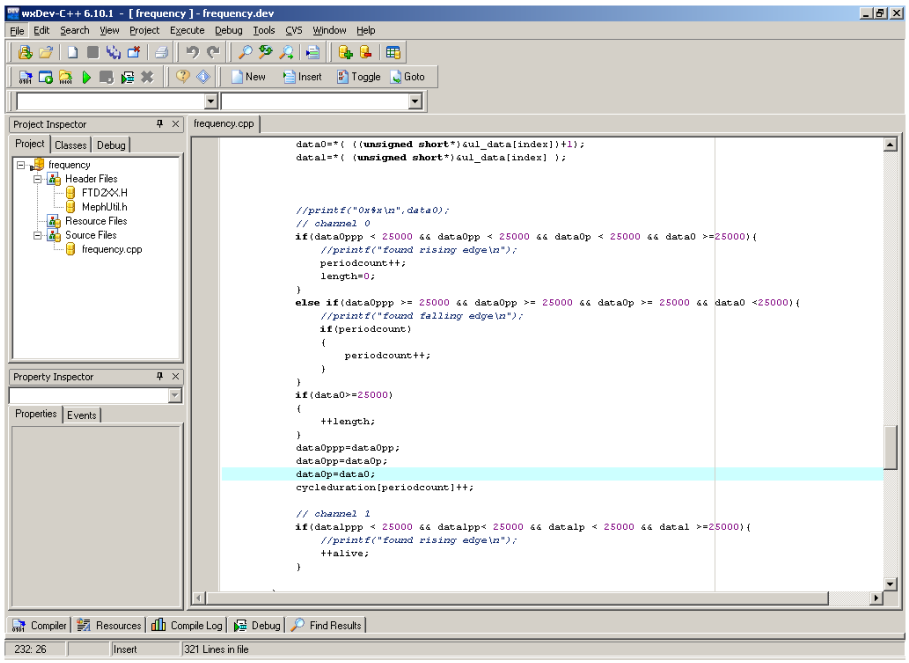

Figure 3.1.: User interface of wxDev-C++

their scope API library. This can be done by including the header files contained in the examples archive and adding the API library to the linked libraries in the compiler setup dialog of wxDev-C++.

3.3. Toolchain installation

The targets used for this diploma thesis do not have the resources needed to run a complete development environment or even a compiler suite. For that reason a normal desktop computer can be used to cross compile the things rather the target system itself. But as the targets may have different CPU architectures or different runtime libraries, special cross-compile toolchains must be deployed. Usually the toolchain installations includes a possibility to create a filesystem image which can be copied to the targets flash memory. From which the target system can boot. The advantage of a combination of toolchain and target filesystem is that both are proved to work correctly together with each other.

3.3.1. Intel x86 toolchain

There are several approaches to get a toolchain for x86 targets, like ptxdist[21], buildroot[22] and some more. But none of them did meet all the needs, because they use feature stripped versions of libraries and programs. But the target has enough disk space to use full featured libraries and programs, just the things which are really useless on an embedded system must be removed to save some storage space.
Additionally it makes sense to have a working package management for the target's filesystem, so that the embedded system can easily be extended by additional packages or unneeded stuff can easily be removed. One possible way to accomplish this needs is to install ArchLinux[23] into a *chroot* environment and write a wrapper for the package management, which shrinks the packages and install them into a target-root directory.
The individual parts of the toolchain's architecture shown in figure 3.2 are described in detail

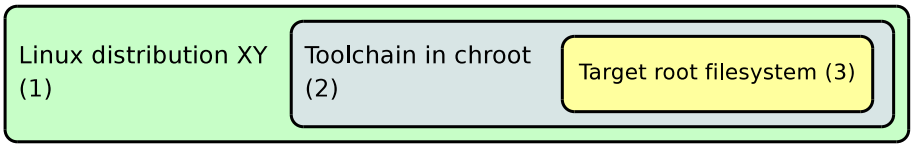

Figure 3.2.: Principle x86 toolchain architecture

by their number in the next few lines.

1. Every recent Linux distribution can be used on the development machine. In principle it could also be a colinux system running in Windows. The toolchain is installed into any folder by launching the installation script as root user. Only root user can install it because some device files must be created and only root is allowed to do that.

2. The "start_toolchain" script can be executed to log into the toolchain. It changes into the toolchains directory and executes chroot within there. So the user can work without

being afraid of destroying something of his real Linux distribution.
All packages of the ArchLinux 0.7.2 release can be easily installed using the package management pacman. Packages which are not on the official release list of ArchLinux can be checked out of their CVS and then be installed.
To create a very basic root filesystem a list (see appendix C.3) of packages must be installed using the epacman command. Epacman extracts all files from the original package, removes documentations, header and development files, locales and many other things which are not needed on an embedded system. Then epacman compresses the remaining files, creates a shrinked package and installs it into the target's root directory.

3. The target's root filesystem is located in the /rootfs folder of the toolchain environment. It contains the complete runtime environment of the target but not the kernel and the *bootloader*.

The advantages of this embedded solution of ArchLinux are:

- Precompiled packages can be used. That saves a lot of compilation time.
- Exactly the same versions of programs and libraries in toolchain and target root-filesystem.
- Programs and libraries are taken, which are used by the Linux community and not only by the embedded community. Thus the packages have been tested by many more people.
- Packages can be easily built because an ArchLinux package is in principle just a tar archive containing the binary files and a file containing information about the package, e.g. dependencies.
- Packages can be very easy installed and removed from the root-fs.

The target's root directory can be directly used as filesystem for the target system.
But there are two suggestive ways of using the root filesystem for the target machine. On the one hand the root filesystem can be directly copied onto the target's flash memory and on the other hand it can be stored as a compressed ramdisk image.
The first solution might be useful for testing purposes but not for real industrial uses. To ensure the filesystem does not get corrupted when the hardware is powered off it must be mounted with read-only access. But this brings some problems when a program wants write a file. It can't simply be mounted with read and write access by arguing with a journaling filesystem as there would be log-files written onto flash memory. And as flash memory does not yet have infinite write cycles will this lead to a system crash sooner or later.
That's why it makes sense to store the whole filesystem into a ramdisk which is loaded into ram on boot up. Then the filesystem is read- and writeable and does not get corrupted due to an unexpected power-off. The disadvantage of this solution is for sure that the memory

which is needed for holding the filesystem cannot be used for something else anymore. Also changes to the filesystem must be made on the development computer if they should remain after a reboot.

3.3.2. ARM toolchain

Cirrus Logic offers a toolchain, detailed installation and usage documentations on their homepage[24]. The toolchain consist of two packages, the ARM cross-compiler and the arm-elf file linker package (arm-elf-gcc-3.2.1-full.tar.bz2 and arm-linux-gcc-3.4.3-1.0.1.tar.bz2). Both can be downloaded from the website and contain prebuilt binaries which can then be installed just by copying them to /usr/local/arm/ and adding the paths containing the executeables to the PATH environment variable.

The root filesystem is not included in the toolchains setup files, so another package has to be obtained from their website called cirrus-arm-linux-1.4.5-full.tar.bz2. This package contains a Makefile suite to generate root filesystems for several targets, one of those targets is the EP9315 based hardware. To compile everything starting "make" in the edb9315 directory does the whole job, it compiles all files of the root filesystem, the Linux kernel and redboot. After that's done the ep9315 directory contains a new ramdisk.gz, zImage and redboot.bin which can then be used for booting up the target.

3.3.3. PowerPC toolchain

The Embedded Linux Development Kit[25] (ELDK) by DENX Software Engineering is used as toolchain to cross compile for the PowerPC target. CD images containing everything what's needed to set up such a toolchain can be downloaded from their homepage. They divided the distribution into two parts, one for the 4xx series of PowerPC processors and one for Freescale family (8xx, 6xx, 74xx and 85xx). After downloading the image for Freescale it can be either burned on a blank recordable CD or directly mounted using the loopback device. The current version of ELDK 4.1 is designed for compiling a target system using a Linux kernel of the 2.6 line and it turned out that it has some serious issues with kernels of the 2.4 line. Thus two different toolchains must be obtained, the ELDK 3.1 for compiling 2.4 kernels and ELDK 4.1 for 2.6 kernels.
To install the toolchains just the install script on the top level directory of the CD has to be executed. It uses an independent rpm package-management contained on the CD and installs everything into the directory, from where the install script is called. After the installation is finished some environment variables must be set.

- "`export CROSS_COMPILE=ppc_8xx-`" - Responsible for setting some compiler flags for the particular target architecture.
- "`export ARCH=ppc`" - Makes the "Makefile" tool aware of the architecture it should cross compile to.
- "`export PATH=$PATH:/opt/eldk/usr/bin:/opt/eldk/bin`" - Needed for Linux to know where to search for executeables. Must be set to the correct locations.

These variables can also be set in some kind of toolchain start-up script, so they don't have to be typed every time again.
Besides the toolchain a *tftp* and *nfs* server should be installed on the development machine. The tftp server is needed to download the kernel image from the development machine into the ram of the target machine. The nfs server shares then the root filesystem and the target mounts it while boot-up.

Tftp server setup

Setting up a tftp server is quiet simple for an OpenSUSE system and should be very similar with other distributions. First of all the tftp server must be installed using "Yast2", a point and click installation tool for OpenSUSE. Then the executable "in.tftpd" must be started with option "-l", option "-s" and the directory which should be shared added as start-up argument. If problems occur the firewall should be check to not filter port 69/udp and the /etc/hosts.allow file must contain "in.tftpd: ALL" if the system has deny all policy.

Nfs server setup

To set up a nfs server is not as easy as the tftp server. Not because there is much to do, more because there are so many things which could go wrong. In general just the following three points must be proceed to get the server up:

1. NFS server must be installed using "Yast2".
2. Following line must be modified to fit the correct path and added to /etc/exports:

   ```
   /media/disk/ELDK-3.1.1-20050607/ELDK_-_Umgebung \
   *(rw,no_root_squash,no_subtree_check)
   ```

3. The daemon /etc/init.d/nfs must be started.

But as already mentioned are there a lot of traps someone could run into. Some hints to start the searching for the reason of a problem are: Is portmap running? Does the firewall block the ports needed? Is the line in /etc/exports absolutely correct?

3.4. Target setup

Besides the root filesystem a Linux kernel has to be built especially for the target. But that's quiet usual for embedded systems. It does not make much sense to use a precompiled kernel with hundreds of additional kernel modules that support every piece of hardware, which even does not exist for the target. Furthermore the real-time extensions modify the Linux kernels, so actually these kernels have to be compiled anyway.

3.4.1. Intel x86 target

The Intel architecture is probably the simplest one for compiling everything because no cross-compiling has to be done here. All following descriptions suggest that every step is done within the ArchLinux embedded toolchain (chapter 3.3.1).

Rtai on Intel x86

The first real-time solution that should be installed on the target is Rtai. At the beginning the needed packages must be downloaded from their respective homepages. In fact that is Rtai-3.5, the Linux kernel 2.6.19 and the Linux kernel 2.4.34. All of those packages should be extracted to the /usr/src/ folder of the toolchain and the kernel folders should be called something like "linux-2.4.34", so that they don't get mistaken or lost.

Then the kernels must be patched using the Adeos I-pipe patches[28] contained within the Rtai package. The Adeos patch modifies the Linux kernel that it can be ran within the idle task of the Rtai kernel and that it offers the interfaces needed for the Rtai kernel. To apply the patch following command-line must be executed within the kernel tree:

```
patch -p1 < /usr/src/rtai/base/arch/i386/patches/\
hal-linux-x.x.xx_rx.patch
```

The next step is to configure and compile a kernel for the target system. Configuration can be started using the "`make menuconfig`" command within the kernel source tree. Only hardware which is really built in the target machine should be compiled into the kernel, to keep the kernel small and stable. Furthermore every module should be compiled into the kernel, no extern kernel-modules should be created. This will really simplify the whole process, because no additional modules have to be installed and loaded while boot-up. At least the options necessary for Rtai should be activated, those options are:

- "prompt for development and/or incomplete code/drivers" in "code maturity level options", set to yes.
- In "loadable modules support", "Enable loadable module support" must be enabled and "Module Versioning support" disabled.
- "Preemptible kernel" and "Use register arguments" of the "processor type and features" section must be both disabled and "interrupt pipeline" must be enabled.
- "/proc file system support" in "Pseudo filesystems" subsection should be enabled for monitoring the Rtai environment.

Example configurations for the used systems can be found on the CD. The kernel compilation is started by "`make bzImage`". A kernel image will be created and must be copied from kernel-tree/arch/i386/boot/bzImage to the /boot/ directory of the target's hard-disk or flash-drive.

To boot up the kernel a bootloader is needed. A very good and easy bootloader for Intel x86 architecture is grub. To install it on the targets disk the disk must be connected to the development computer and mounted e.g. to /mnt/target/ then following command sets up grub on the disk:

```
grub-install --root-directory=/mnt/target hdb
```

The parameter of this call must be set very carefully, as choosing a wrong device will probably break the development computer.
To make grub aware of the new kernel, some additional lines in the configuration file of grub are needed. The configuration file is located at target-harddisk/boot/grub/menu.lst. An entry for booting a kernel looks like this:

```
title  2.6.19-rtai-3.5
root   (hd0,0)
kernel /boot/bzImage-2.6.19-rtai-3.5 ro ramdisk_size=59000 \
   root=/dev/ram0 mem=0x7000000
initrd /ramdisk.img.gz
```

While the title sets the string how the boot option should be displayed in grub's boot menu, root specifies which hard-disk contains the kernel. The "initrd" line tells grub which ramdisk should be loaded. Actually that is the ramdisk containing the root filesystem of the target. The line with information about the kernel contains the path to the kernel-image and some kernel parameters:

ramdisk_size=...: This option specifies the size of ramdisks. The value must be bigger than the size of the ramdisk created of the root filesystem.

root=...: specifies which device contains the root-filesystem. In the case of a ramdisk it is /dev/ram0.

mem=...: This kernel boot argument is used to define how many memory the Linux kernel is allowed to use and manage. This has to be limited because ORF is using its own memory management and the kernel is not allowed to not touch it.

After changing into the Rtai source directory and typing "`make menuconfig`" the configuration interface of Rtai is started. The default configuration should already fit most of the needs. Just the kernel location must be set to the correct path of the Rtai patched kernel sources. A complete Rtai configuration for 2.4 as well as for 2.6 Linux kernel can be found on the CD. Compilation and installation of Rtai is invoked by the "`make && make DESTDIR=/rootfs/opt/rtai2.6 install`" command. As destination directory any directory on the targets root-filesystem can be chosen, but it makes sense to include at least the kernel version in the file-name to differ between the kernel modules compiled for 2.4 and the ones compiled for 2.6.

Xenomai on Intel x86

Compiling and installing Xenomai on the target is quiet similar to the way Rtai is installed. First of all the Xenomai package of release 2.3.1 and the Linux kernel 2.6.20 should be downloaded, it turned out that this is a stable combination. After extracting them to /usr/src it makes sense to rename the directories to meaningful names like "linux-2.6.20-xenomai", because renaming or moving the folders after compilation will lead to errors if something has to be recompiled later.
Applying the kernel patch is different from Rtai, Xenomai offers a script to do that:

```
scripts/prepare-kernel.sh --arch=i386 \
  --adeos=ksrc/arch/i386/patches/\
   adeos-ipipe-2.6.20-i386-X.Y-ZZ.patch \
  --linux=/path/to/kernel/tree
```

After this step the Linux kernel must be configured using "`make menuconfig`" within the kernel source tree. The same "less is more" rule as true for Rtai applies here, only hardware which is built into the target hardware should be compiled in the kernel. In order to get a Linux kernel with proper running Xenomai extension "Xenomai" and "Nucleus" within the "Real-time sub-system" section must be enabled. Additionally the interrupts must be enabled in the "Interfaces" sub section, at least for the "Native API". As already mentioned for the

Rtai kernel every module should be built directly into the Linux kernel, also the Xenomai extensions. A complete kernel configuration can be found on the CD.
The build process is started by invoking the command "`make bzImage`". As soon as it is finished the Xenomai Linux kernel image is located at arch/i386/boot/bzImage and can be copied on the targets disk into the /boot/ directory.

The Xenomai Linux kernel needs also an entry into grubs configuration. The three lines which have to be added look like this:

```
title  2.6.20-xenomai-2.3.1
root   (hd0,0)
kernel /boot/bzImage-2.6.20-xenomai-2.3.1 ro \
   ramdisk_size=59000 root=/dev/ram0 lapic mem=0x7000000
initrd /ramdisk.img.gz
```

ramdisk_size=... / root=... / mem=...: already described in the Rtai part of this chapter, have a look at page 26.

lapic: This option enables the local APIC (Advanced Programmable Interrupt Controller) even if it is disabled in the bios. Xenomai needs this option to enable real-time operations.

To compile the rest of the Xenomai environment like user-space libraries, the command "`./configure --enable-x86-sep --prefix=/opt/xenomai`" initializes the compilation process and "`make DESTDIR=/rootfs/opt/xenomai install`" finally compiles and installs it.

Rt-preempt patch on Intel x86

The last part of the set of real-time solutions is the rt-preempt patch[32] for the vanilla kernel. Due to the lack of ACPI support of the gx1 target machine only kernels up to version 2.6.18 and rt-preempt patch 2.6.18-rt5 can be used, because starting with rt6 "pm_timer" of the acpi subsystem is needed in order to get a working rt-preempt patched Linux kernel. So the Linux kernel version 2.6.18 and the rt-preempt patch 2.6.18-rt5 must be downloaded and extracted to /usr/src/ path, again keeping in mind clearly named directories. For example "2.6.18-rt5-preempt". After changing into the Linux kernel source tree the patch can be applied using following command:

```
patch -p1 -i /path/to/patch/patch-2.6.18-rt5
```

As already mentioned for the compilation of the previous two kernels, only drivers of hardware built into the target system should be enabled when configuring the kernel using "`make menuconfig`".
Other important settings are:

- "High-Resolution-Timer Support" in "Processor Type and Features" section set to yes.
- Power management options like APM or ACPI should be all disabled - at least for kernel 2.6.18 with rt5 preempt patch. If a more recent version is used then the ACPI option must be enabled, while all sub options of ACPI should still be disabled, because they could break the reliability of the real-time system.
- In the "Kernel Hacking" menu entry are several options to monitor the state of the kernel regarding real-time ability. These could be enabled for testing purposes, but must be disabled for running benchmarks because they could falsify the result.

As usual the kernel can be compiled using the "`make bzImage`" command and the built kernel image is located at arch/i386/boot/bzImage. After copying it onto the targets disk the menu configuration file of the bootloader must be altered again.

```
title  2.6.18-rt5-preempt
root   (hd0,0)
kernel /boot/bzImage-2.6.18-rt5 ro ramdisk_size=59000 root=/dev/ram0
initrd /ramdisk.img.gz
```

All those boot-parameters are already explained on page 26.

Creation of the ramdisk

After the different kernels and real-time environments are compiled it is time to create the ramdisk and boot up the target to test if the software is running correctly.
The advantages of using a ramdisk are:

- The image can be compressed, what saves about 2/3 disk-space.
- Once loaded into ram, file-access is very fast.
- It is mounted with read and write access, files can be changed, removed and new ones can be created.
- The image itself cannot be modified, what makes the installation resistant against unexpected power-offs. If anything breaks while runtime, a reboot restores exactly the original state of the system.

But using a ramdisk brings also some disadvantages:

- Changes made to the target while the target is running are not saved and lost when the target is powered off.
- The amount of ram which is needed to hold the filesystem can't be used by other processes anymore.
- Boot-up procedure may be delayed, because the kernel has to extract and check the filesystem.

A possible solution to weaken the disadvantages is to combine the ramdisk with a normal mounted partition. This way it is possible to remove some parts of the ramdisk and add them to the mounted partition. That makes the ramdisk smaller, what affects directly the speed of boot-up and the amount of needed ram. Additionally there is a persistent space which can be used for files which should still exist after a reboot. But if too much is swapped from the ramdisk to such an additional partition the advantages of the ramdisk get lost again. E.g. when the "/usr/" folder is on the partition instead of in the ramdisk, important files could be permanent modified or removed what could prevent the target from booting correctly. A good compromise is to outsource the "/opt/" folder to the partition and install there any program and data needed for the respective application of the embedded system. If this data gets lost the target will still boot up and can be controlled via network connection. To mount the second partition of the target's disk just a line has to be added to the "/etc/fstab" file of the target:

```
/dev/hda2 /opt ext3 defaults 0 1
```

To finally create the ramdisk a script called "makeramdisk" has been build and can be found on the CD. It processes in general following steps:

1. Calculating the size of the root filesystem without the "/opt/" folder. Adding some megabytes to have free space in the ramdisk for log-files and so on.
2. Creating a file with exactly the size calculated the step before and formating it with a ext2 filesystem.
3. Mounting the file using a loopback device and copying the data preserving permissions and other attributes.
4. Unmounting the file and compressing it using the gz compressor.
5. Compressing the files of "/opt/" to an archive with preserving file attributes

Now the compressed ramdisk image can be copied over into the top level directory of the first partition of the target's disk and the archive containing files of the "/opt/" folder can be extracted to the top level directory of the second partition.

Testing the target

As a short summary, the Intel x86 target is now able to boot up 2.4.34 and 2.6.19 with Rtai extension, 2.6.20 with Xenomai and the 2.6.18 rt-preempt patched kernel. But to ensure that the real-time extensions are really running and to prevent getting in troubles later it makes sense to test every kernel with its real-time environment. Therefore little kernel-modules or programs should be way enough which can be found on the CD. In principle all of them just create a real-time thread using their respective API and after the kernel-module was loaded without throwing any error it can be checked whether the thread is listed in their scheduler as a real-time scheduled task or not.

To print out the list of Rtai scheduled threads the line "`cat /proc/rtai/scheduler`" does the job. For Xenomai it's nearly the same, just Rtai is replaced by Xenomai: "`cat /proc/xenomai/sched`". This procedure is little different for the rt-preempt patch, there the real-time application is running in user-space, not in kernel space. So for this environment it can be checked whether the program is running with real-time scheduling when by checking the process list using the "`ps -eo pid,rtprio,ni,comm`" command. Where "pid" shows the process id, "rtprio" the real-time priority, "ni" the *nice* value and "comm" the program name. "rtprio" is unset for normal scheduled applications and is set for real-time scheduled ones.
If those simple tests pass and in "`dmesg`" and in the log-files in "/var/log" does not appear any strange error it is suggested that the real-time solution works correctly.

3.4.2. ARM target

The setup of the ARM target is much more complicated than for the Intel x86 target. That is mainly related to the fact, that the vanilla Linux kernel does not fully support the cirrus EP9315 SOC processor. Thus a patch for the respective kernel version is needed, but cirrus only offers a patch for Linux kernel versions 2.4.21 and 2.6.8.1. The problem is that the preemption patch does only exist for kernel versions 2.6.18 and later and Adeos patches for this architecture are only for 2.6.14 and later kernels available.

There was a project to add ep9315 support to recent Linux kernels of the 2.6 line, but it seems like this project died and the homepage has gone offline while this diploma thesis was

written. The website can still be accessed through web archive[33]. The goal of the project was to get support for some targets, including ep9315 platforms, into the mainline Linux kernel. The last Linux version which was claimed to be fully supported was version 2.6.15. Unfortunately the Linux kernels patched with this patch did not boot up. Some kind of *JTAG* would probably be useful to find out where the problem is located.

Another approach to support the ARM architecture within this project would be to use another hardware. After some research in the internet the "ARM & EVA"[34] board from Conitec Datensysteme seems to be very suitable for embedded real-time solutions based on ARM. It contains an AT91RM9200 processor, which has a very active Linux support community[35]. It is officially supported[36] by the Xenomai project and searching the Rtai mailing list[37] pointed out that there has been some effort in porting Rtai to this platform. Due to the Linux patches which offer compatibility to this platform for latest Linux kernels of the 2.6 line it should be possible to get the rt-preempt patch working within a limited period of time.

In summary setting up a Linux real-time solution on the ARM platform would probably require another piece of hardware or porting of some patches for the ep9315 target to a more recent Linux kernel. But as the time of this diploma thesis is very limited, will the ARM platform not be considered anymore.

3.4.3. PowerPC target

As the PowerPC is a very common platform for running real-time applications the support is not as bad as it is for ARM. All steps in this subsection require ELDK 3.1 and 4.1 correctly installed and environment variables set like described in chapter 3.3.3.

Rtai on PowerPC

As done for the Intel x86 target the first real-time approach set up for the PowerPC target is Rtai. The Denx company offers in its *Git*[38] tree a version of the 2.4.25 Linux kernel specially patched for the PowerPC platform. It contains additional drivers, e.g. the serial port driver of the MEG32 hardware and there are Ipipe patches on the Adeos ftp exactly for this version of the Linux kernel. So after the Linux sources have been downloaded using the command-line "`git clone git://www.denx.de/git/linuxppc_2_4_devel.git linux-2.4.25-rtai`" the fitting Ipipe patch can be applied by switching into the linux-2.4.25-rtai directory and executing following command:

```
patch -p1 < /path/to/adeos-ipipe-2.4.25-ppc-..patch
```

To save time and nerves a default configuration for different PowerPC targets is included in the kernel source tree. To use this default configuration the command-line "`make mrproper; make TQM820_config`" initializes the .config in the kernel source tree. Within the kernels configuration interface, which is started by invoking "`make ARCH=ppc CROSS_COMPILE= ppc_8xx- menuconfig`", the same options like described for the Intel x86 architecture should be set (Have a look at chapter 3.4.1). After saving the configuration the kernel can then be built by following command:

```
make  ARCH=ppc CROSS_COMPILE=ppc_8xx- uImage
```

When the compilation is done the uImage has to be copied into the directory which is shared by the tftp server. To boot the kernel on the target system a boot loader is needed, fortunately the MEG32 system is shipped with a preinstalled u-Boot[39] bootloader. The first installation of this bootloader has to be done with a JTAG connection, but further upgrades can be easily done with u-Boot's command prompt and a tftp server. First of all the uImage must set to be downloaded to the target by typing "`tftp 200000 uImage`", while the target address 200000 is given by the defined by the specifications of the hardware. Boot-arguments can be passed to the kernel by setting the "bootargs" variable of u-Boot. And the command "bootm" finally boots the target from the ram address where the kernel has been stored. The following line shows how the bootargs variable must be set to boot up the kernel correctly for usage with ORF and using a nfs share as root file system. A text file containing all variables of the u-Boot environment can be found on the CD.

```
root=/dev/nfs rw nfsroot=serverip:rootpath
```

At last the Rtai modules have to be compiled. Therefore the Rtai package version 3.5 which has already been downloaded in course of Intel x86 target setup can be extracted. "make menuconfig" has to be launched within the extracted sources and the kernel tree must be set to the path, where the Rtai kernel is located. Then the compilation and installation process is started by "`make && make DESTDIR=/install/path/ install`" keeping in mind that the installation directory must be within the nfs shared path, so the target system can access it. Compilation may fail with strange errors, like missing headers. If this is the case, explicitly preparing the kernel by "`make oldconfig && make prepare`" could fix the issue. Make prepare is usually called implicitly while kernel compilation but in some strange cases it seems not to work.

Xenomai on PowerPC

As Xenomai has Linux kernel 2.6 support for PowerPC systems it makes sense to setup a 2.4 Linux kernel as well as a 2.6 Linux kernel with the Xenomai extension. In fact another copy

of Denx 2.4 PowerPC Linux kernel must be obtained using git (see Rtai section of 3.4.3) and the vanilla kernel version 2.6.19 must be downloaded from the official kernel.org site. After the kernel packages and the Xenomai-2.3.1 packages are extracted and are renamed to have meaningful names the kernels can be patched.

```
scripts/prepare-kernel.sh --arch=ppc \
--adeos=ksrc/arch/powerpc/patches/adeos.patch \
--linux=/path/to/kernel
```

The "adeos.patch" of course has to be replaced by the respective patch for the kernel. It is important to carefully separate the generation of the 2.6 kernel and the 2.4 kernel because those do have different toolchains. The 2.6 kernel compiles only within the ELDK 4.1 toolchain and the 2.4 kernel within 3.1. Now as the kernel is patched the default kernel configuration can be created using "`make mrproper; make TQM820_config`". Within the kernel configuration interface which is opened by running "`make ARCH=ppc CROSS_COMPILE=ppc_8xx-menuconfig`" the important options "Xenomai" and "Nucleus" within the "Realtime subsystem" must be enabled. Also the interrupts within "Native API" must be built into the kernel. The following command starts then the generation of the u-Boot compatible kernel image:

```
make ARCH=ppc CROSS_COMPILE=ppc_8xx- uImage
```

The images of both kernels can be copied into the folder which is shared by tftp to the Rtai kernel image.

The last step of setting up Xenomai is to compile the user-space libraries. This is done by configuring Xenomai for using the cross-compilation tools which can be achieved by execution of the following command within the Xenomai source tree:

```
configure --build=i686-linux --host=ppc CC=ppc_8xx-gcc \
CXX=ppc_8xx-g++ LD=ppc_8xx-ld
```

When it is done Xenomai can be compiled and installed by
"`make DESTDIR=/installation/path/ install`",
keeping in mind that the installation path is within the nfs shared folder.

Rt-preempt on PowerPC

In principle the Rt-preempt patch is available for PowerPC targets starting with kernel version 2.6.18. But the Linux kernel developers began to implement the PowerPC architecture for the Linux 2.6 kernel in a different way than it was done for the 2.4 line. To still support PowerPC targets to which of 2.4 line, they created two architecture directories within the kernel tree. One is called "ppc" and the second one "powerpc". This can be really confusing and after doing some research it turned out, that the "ppc" is the old architecture just copied over from 2.4 kernel line and thus unsupported. The "powerpc" folder contains the new PowerPC architecture which is actively developed and is handled as the supported PowerPC architecture of the 2.6 kernel line.
That's why the rt-preempt patch is exclusively build to support the newer "powerpc" architecture and not the old "ppc" one. Unfortunately the processor of the MEG32 hardware is not yet in the newer architecture, so simply applying the rt-preempt patch is not possible. First thing would probably be to port the platform from "ppc" to "powerpc". Wolfgang Denk responded to the question how long that would take: "if we should perform such a port for a customer, we would probably estimate 2 to 3 weeks; and we do have some experience in this area". What means that somebody who has not done any similar task yet would easily need one to three months. So this real-time solution will be skipped on the PowerPC platform for this thesis.

Testing the PowerPC target

In summary the kernel 2.4.25 with Rtai and Xenomai extension and 2.6.19 with Xenomai extension are booting on the PowerPC target. To ensure that the real-time extensions are working correctly the same little test modules which are used to verify the Intel x86 target (see 3.4.1) are compiled. Then they can be loaded while the target is running with the corresponding real-time extension and the scheduler can be checked by reading from the respective scheduler file in the /proc/xenomai or /proc/rtai folder.

3.5. ORF implementations

The Open Realtime Framework is a project which grows with its applications, what means, that only things which have been need by any projects are yet implemented. Thus a lot of features are missing to finally build the benchmarks on top of this framework. This section handles about the design and implemenation of the modifications done to ORF to offer the additional features needed for running the benchmarks.

3.5.1. Dynamical loaded libraries

When compiling ORF as kernel-modules there isn't just one kernel-module generated. It is split into at least two kernel-modules, "orf_methods_real.ko" and "krn_orf.ko". The first one contains the API functions of ORF and the second one does the initialization and communicates through the pipes with user-space programs. Those two modules can easily work together because they get linked to each other when loading them into the Linux kernel.

But ORF is designed to also work in user-space, where the modules cannot be easily linked together. User-space support can't also easily be skipped because when running the target with the rt-preepmt patch, ORF must run in user-space. And linking all objects together right after compiling isn't an option neither, because then the whole ORF has to be rebuilt just because a RProg needs to be added or removed. Thus a solution which offers dynamical linking in user-space must be implemented. It should offer an easy way to link and unlink e.g. RProgs while the rest of ORF is not influenced, to allow changing a RProg while the system is still running. Besides that the new feature should not conflict with the specifications of ORF or break any existing functionality. The best way would be to enhance existing functions by the functionality needed for handling with the linked libraries.

Principle design of dynamical loaded libraries within ORF

As it is not really needed to have the two files "orf_methods_real" and "krn_orf" separated, they could be linked together to one executable while compilation. That reduces the cases when dynamical linking is needed to adding and removing of RProgs. A good point would be, to keep the program structure of the RProgs, so that RProgs look the same, independent if they are thought for running within the kernel or in user-space. Since the kernel requires at least an static init function to load a module, this function can also be used in user-space, just without the static keyword. After reading the specification of ORF it turned out, that following approach fits best the requirements:

- RProgs are compiled as shared object. Compiling a C program to be a shared object requires some more compiler-flags than compiling a normal executable. This is how such a command-line in principle looks like:

  ```
  $(CC) -fPIC -shared -Wl,-soname,rt_prog.so -o rt_prog.so \
  rt_prog.c -lc
  ```

  -fPIC: tells gcc to produce position independent code and avoid any limit on the size of the global offset table.

  -shared: gcc produces a shared library when this parameter is given.

-Wl,-soname,rt_prog.so: the -Wl argument passes the followed comma separated argument list directly to the linker. In this case, the linker gets the command to name the shared object "rt_prog.so"

-o rt_prog.so: output name of the binary produced by gcc set to rt_prog.so.

-lc: the option -l specifies against which library the generated code is dynamically linked. In this case the generated shared library is linked against standard c library.

- The file-names of the RProg libraries are unique and are used to associate the RProg within ORF with a file in filesystem and the other way around.
- The ORF functions "orf_add_initfunc" and "orf_delete_initfunc" get modified to handle with the dynamic linked objects.

A principle cycle of loading and unloading such a shared object realized with the syntax of a orf_startup ini file looks somehow like shown on figure 3.3

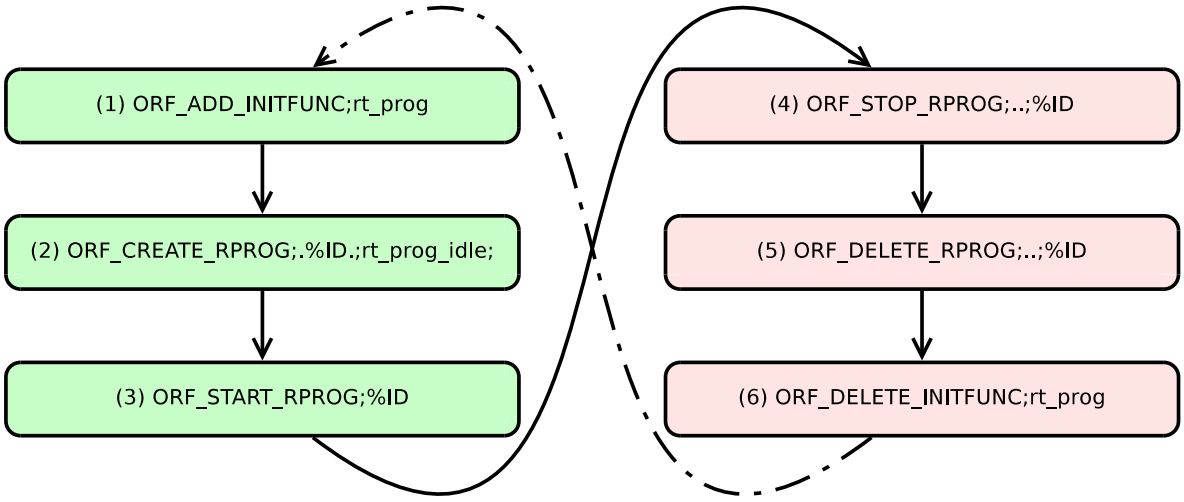

Figure 3.3.: Loading and unloading shared objects

1. Loads the shared library with the file-name "rt_prog.so".
2. Adds the main function of the RProg to a Thread0 of a given device and associates it with the given RProg id.
3. Sets the RProg into the running state.
4. If the RProg should be removed, e.g. because the behavior of the RProg has changed, this call sets the former RProg into stopped state.

5. Now the RProg can be deleted. If the stop command would have not been invoked, ORF would crash at the next call of the main function, because of a function call outside the valid memory.

6. This function unlinks the RProg of the runtime image. The file can be replaced by the new RProg and with starting at call NR. 1 the RProg can be again loaded into ORF.

Implementation of dynamical loaded libraries in ORF

As already mentioned in the section before, are there at least two functions which have to be modified to support the dynamical loaded shared objects within ORF. These are the functions "orf_add_initfunc" and "orf_delete_initfunc". Until now both functions just had one parameter, a pointer which points to the init function which should be added. Both function need to be expanded by a second parameter, a string, containing the name of the RProg, which should be linked or unlinked. Figure 3.4 shows the modifications made to the original "orf_add_initfunc" and "orf_delete_initfunc" in form of a flowchart. Besides the changes to the functions the shared memory zero-page must be extended by an array holding the handle, the name and the function pointers of loaded libraries.

Some important C calls for dealing with shared libraries are:

handle=dlopen("rt_prog.so", RTLD_LAZY); opens the shared library named "rt_prog.so" and returns the pointer which is used as a handle. The second parameter is a flag which defines how the library should be opened. RTLD_LAZY means that the symbols are resolved when they are accessed.

initfunc=dlsym(handle,"init"); searches the library to which the handle variable points for a symbol called "init" and returns the respective pointer. Thus initfunc contains the pointer to the init function of the RProg after the dlsym call returned.

dlclose(handle); closes the shared library to which the handle points to.

errstr=dlerror(); returns a human readable error string if an error appeared while handling with dynamic linked libraries.

3.5.2. Character devices

When ORF is running in user-space, the communication is established using standard Linux pipe device files which are polled continuously by ORF at 10Hz frequency. This does not make sense at all when ORF is running in kernel-space, because then the data must be copied from user-space to kernel-space and the other way around. One of the possible solutions to

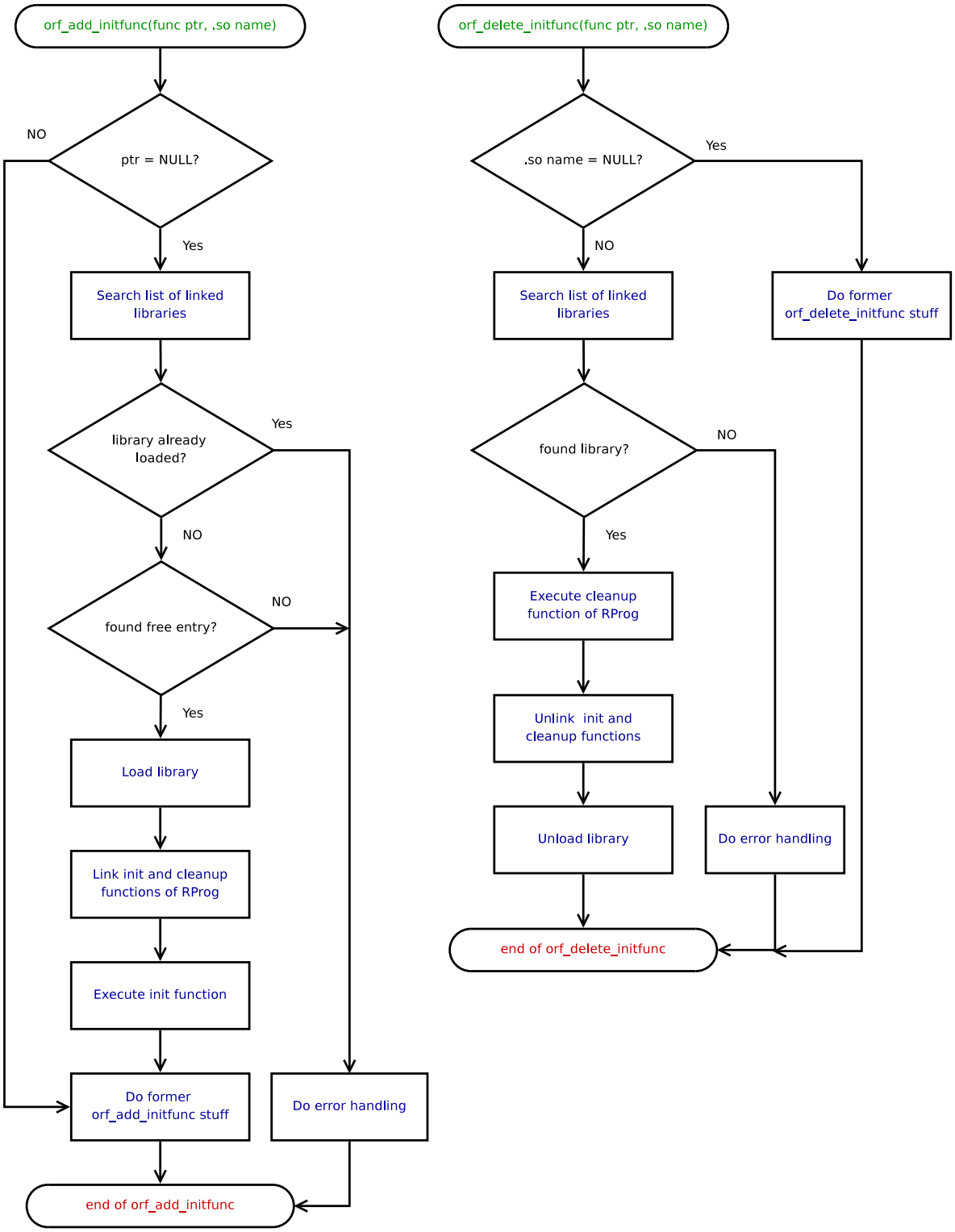

Figure 3.4.: Flowchart of changes to enable dynamical loaded libraries

do so is to use the fifo files of the real-time extensions. Rtai and Xenomai come both with support of real-time capable pipes and the actually former way of communication between ORF and user-space was exactly to use this pipes. The pipes were just implemented for Rtai and not for Xenomai yet.

But there is a general difference between the Rtai fifos and the Xenomai fifos. Fifos in Rtai work like interrupts are proceed in general. Once the handler for the fifos is set up, a handler function gets called every time such a fifo is accessed. That means, that the normal program flow is interrupted and the handler function is executed.
Unfortunately there is no such handler for Xenomai. Xenomai however just supports polling the fifos what requires a second thread, so that the normal program flow is not disturbed. After implementing this thread as a Xenomai task, and testing it with some basic communication it looks good in first place. But it turned out, that some important memory functions cannot be done within an Xenomai task. For example the function call "ioremap" which is highly required by ORF to access the shared memory. Calling this function within a Xenomai task ends up in a kernel panic.

As the communication is established to the user-space no real-time capable communication is needed, because the user-space program itself is also not real-time capable. Thus the real-time extension's pipes are not needed, so the more flexible approach is to use normal kernel device files. Those can be easily added to ORF and offer also some kind of handlers, what means that there is no extra polling thread needed.

Principle design of character devices within ORF

The best place to add the character device functionality is in the krn_orf module, because the orf_methods_real module should contain only the functions of the ORF API.
Furthermore ORF's specification pretends that there must be more than just one fifo file for communication. This allows simultaneously connections to ORF. The best way to offer more than one character device per kernel module for the same functionality is to use the same Major number, but a different Minor number. The command "`mknod /dev/rtf0 c 240 0`" creates the character device file "/dev/rft0" with Major number 240 and Minor number 0. Major numbers are used within the Linux kernel to associate character device files (usually found in "/dev/") with the corresponding driver. Minor numbers are taken to identify different device files within one kernel module.

Registering a character device within a kernel module can be done with just one command:

```
register_chrdev(MAJOR_NUMBER, "orf_pipe", &fops_pipe)
```

MAJOR_NUMBER: just a precompiler definition, containing the major number used for the character device. Linux supports up to 255 major numbers while no conflict is allowed, so a number should be chosen which is not already used by any other Linux driver. A list of used major numbers can be found in the kernel source tree, in the "documentation/devices.txt" file.

"orf_pipe": the name under which the character device is registered within the kernel's list. This list can be printed by displaying the file "/proc/devices"

&fops_pipe: a pointer to the structure containing pointers to the handler functions of the character device. A character device has at least four handlers:

Open Contains the function pointer of the code which is executed when the device file is opened by any process. Within this code block the driver should care about mutexing the resource and about the minor number of the accessed device. Also initial stuff should be done here.

Read The function where this pointer leads to is launched when any process begins reading from the device file. The function must serve the reading process with data and end serving by sending the end of file signal.

Write Points to the function called when an user-space application writes data to the device file. The job of this code is to receive data from the user-space process and probably already also processing it.

Release This pointer refers to the routine called when the user-space process closes the device file.

One complete communication cycle between ORF and a user-space program like "orf_startup" should look like the flow on figure 3.5.

1. The user-space program opens the device file using any standard file-open call.
2. The open handler function of the kernel module is called.
3. The user-space program writes its command into the device file.
4. The write handler function is executed and copies the data from user-space into kernel-space and starts the processing of the command.
5. The command takes a while to be completely processed. The result is then stored.
6. The user-space process wants to read the result of the command, but the command is still in execution. So the user-space program is blocked until the execution of the command is finished and the result is stored.

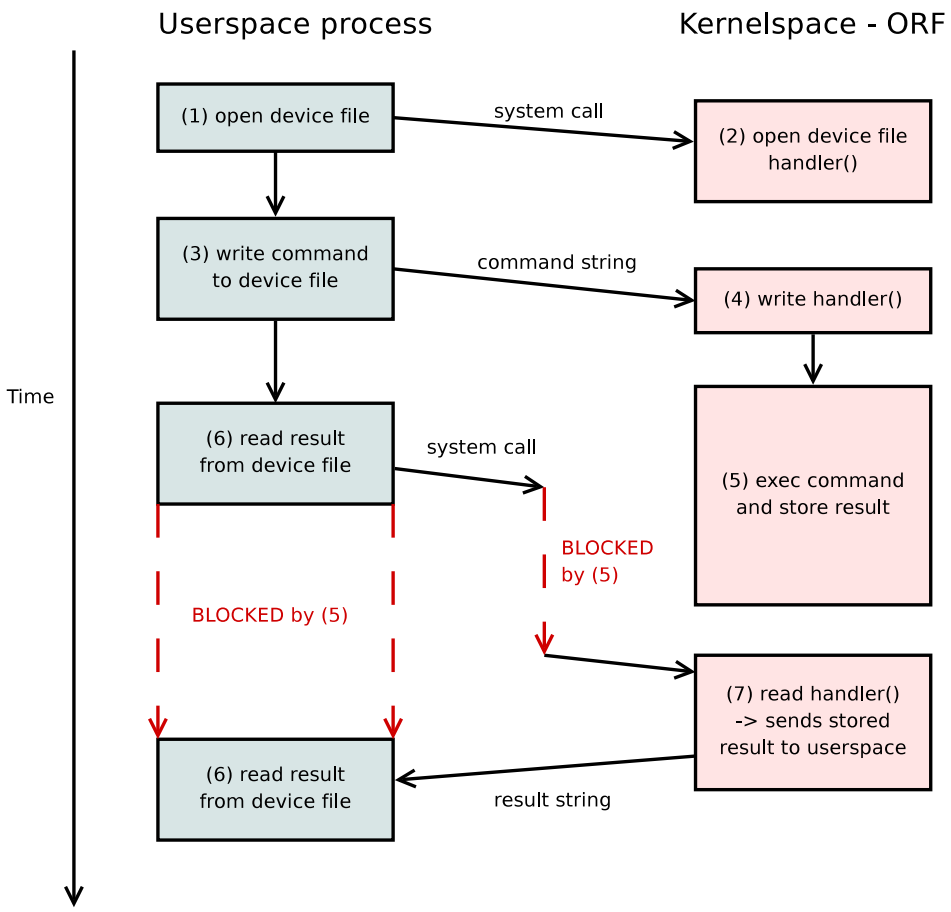

Figure 3.5.: Communication between user-space and ORF using character devices

7. Is called by step 6. and copies the stored result from kernel-space into user-space. While step 6. gets unblocked and reads it from the device file.

Implementation

The concrete implementation consists just of the code of the four handlers functions and a little modification to the "orf_rtf_server" routine. The former "orf_rtf_server" routine was directly set to be the handler of the Rtai real-time fifo. This changes and the "orf_rtf_server" gets a new parameter, a string, which contains the received command. The functionality of this function is not touched at all.

Before implementing the handler functions a structure for holding some information for each character device must be created:

Listing 3.1: pipestruct

```
typedef struct pipestruct_st
{
```

```
    /* Number of device, equal to the minor number */
    int number;
    /* String buffer for holding the result of a cmd */
    char buffer[MAX_FIFO_SIZE];
    /* Pointer to next character which should be sent
     * to userspace - needed for read handler */
    char *ptr;
    /* States if the device is in use */
    int in_use;
} pipestruct_t;
```

An array containing as much of those structures as the amount of allowed pipes must be created. Then the character device file handlers can be implemented like described in following lines:

pipe_open(struct inode *inode, struct file *filp) :

inode: contains useful information about the opened device file. The only information which needs to be accessed through this pointer is the minor number, which is returned by this function call:

`MINOR(inode->i_rdev)`

filp: points to a structure which is unique for every device file and contains some attributes. But the most important data within this structure is the "private_data" void pointer. This pointer can be set to point to the corresponding element of the pipestruct array. This may sound complicated, but the effect is that the data of every device file simply can be accessed by "filp->private_data" within the three other handler functions.

The function first checks the "in_use" variable of the private data and returns the busy error value to the opening process if it is in use. If it is not in use it initializes the pipe's private data. "private_data->number" is set to the minor number, "buffer" is cleared and "ptr" points to the beginning of the buffer. At the end "in_use" is incremented to lock the device file.

pipe_write(struct file *filp, const char *buff, size_t len, loff_t *off) :

filp: points to the data of the device file.

buff: buffer containing the data written to the device file in user-space.

len: the length of the string in the buffer.

off: current offset of file. This is only needed for seek operations and not used for the implementation within this thesis.

At the beginning it checks whether the length of the data that should be copied from user-space into kernel-space is not longer than the buffer in kernel-space. Then the data is copied from the user-space buffer "buff" into the kernel-space buffer "filp->private_data->buffer" using the command

```
copy_from_user(pipe->buffer,buff,len);
```

A global mutex is decremented to request exclusive execution of "orf_rtf_server". This prevents from executing two commands over two pipes at the same time. And finally "orf_rtf_server" is called to process the command stored within the buffer. When the function call is returned, the buffer contains the result of the command procession and the critical section is left by incrementing the mutex.

pipe_read(struct file *filp, char *buffer, size_t length, loff_t *off) :

filp: points to the data of the device file.

buffer: points to string buffer in user-space.

length: the amount of requested bytes by the user-space program.

off: current offset, only needed for seek operations.

The only task of this function is to copy the data of the kernel space buffer, which contains the result of the last command, into the user-space buffer. This works by copying one byte after the other using the following command in a loop until the end of the buffer is reached.

```
put_user(*((pipe->ptr)++), buffer++);
```

pipe_release(struct inode *inode, struct file *filp) :

inode: contains information about the device file, has not to be used here.

filp: points to the data of the device file.

This function resets just the "private_data->in_use" value to zero, what means that this device is free.

3.5.3. I/O-API

Since the benchmarks should be done from a very practical point of view it makes sense to implement them to use input and output ports of the hardware. ORF does not have an abstract API function to access I/O of any target yet nor any description in the specification. So the implementation could be done without taking care about conflicting any existing ORF's API function.

Design and implementation of I/O-API for ORF

ORF must be extended by four API-functions, while design is kept very simple so that basically the I/O's can be accessed by RProgs in an easy way. The advantage of this simple design is, that it will work for most I/O hardware. But for usage in a real application the I/O-API may need some enhancements to serve the whole functionality the respective target hardware offers.

orf_initio() Used for initialization of the hardware. Returns 0 on success. When called the first time, it invokes the commands which set up the I/O ports of the particular hardware. For an Intel x86 system for example the call "ioperm(0x378,1,0);" sets the permissions to access the parallel port.

orf_inb() Returns the current value of the I/O port.

orf_outb(uint) Used for setting the output pins to the value given as parameter.

orf_resetio() Resets the I/O port.

Of course the code for accessing the I/O ports of every supported hardware must be added to these functions. Precompiler defines should be used to define the architecture of the several code blocks. This way only code is compiled which is really needed by the target hardware. Figure 3.6 shows how the I/O-API functions are called in a senseful matter.

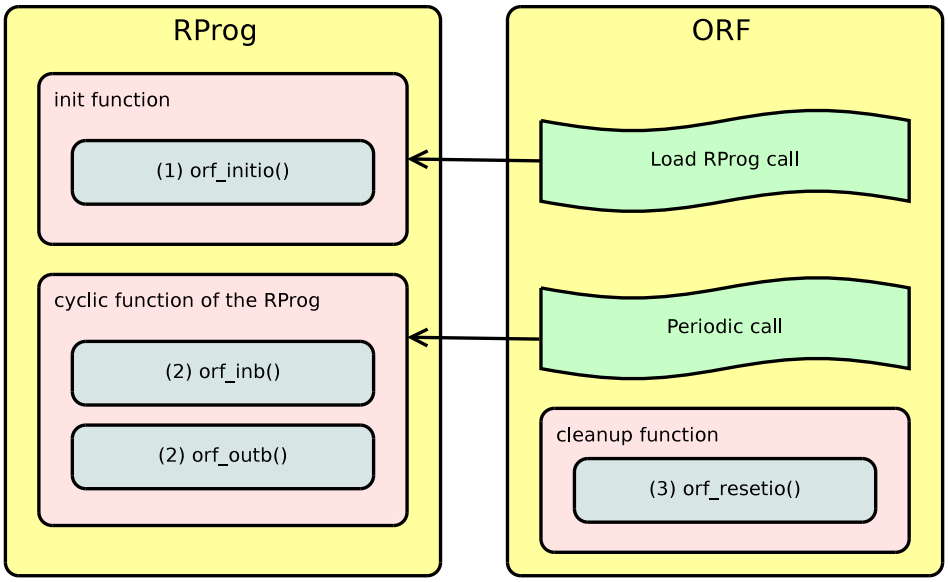

Figure 3.6.: Calls of I/O-API functions

1. The "orf_initio()" function is called when the RProg's init routine is executed by ORF.
2. "orf_inb()" and "orf_outb()" are called within the RProgs periodic function.

3. "orf_resetio()" is called automatically when ORF is stopped and the I/O port has been initialized.

3.5.4. Interrupt handling

In order to catch and handle interrupts with a RProg ORF needs an additional way of calling RProgs than the normal periodic RProg call. The principle idea behind it is, that ORF should offer the functionality to define a RProg which is called when a specific interrupt occurs, e.g. the parallel-port interrupt on an Intel x86 system. As already mentioned for the changes made in the sections before must the modifications made on ORF not conflict with the specification. So again a good way to insert the interrupt handling into ORF must be found.

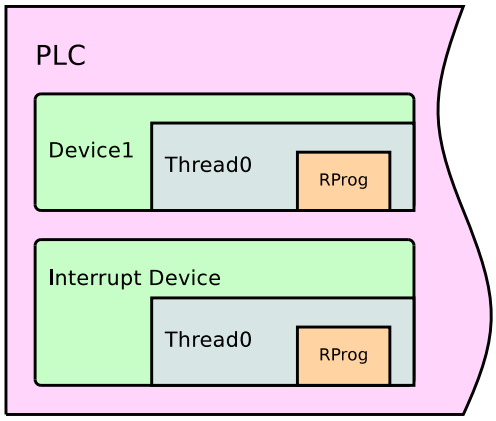

Figure 3.7.: Interrupt device approach

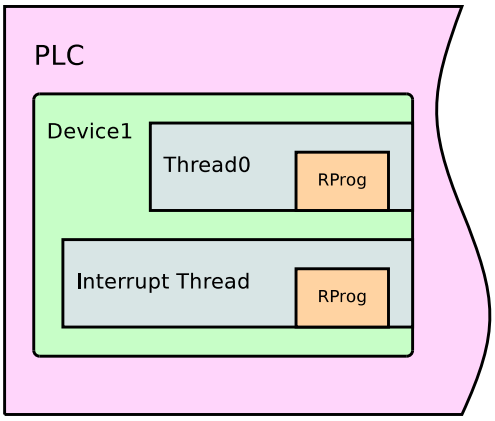

Figure 3.8.: Interrupt thread approach

There are a lot of approaches to adopt the interrupt handling, but the two best are shown on figure 3.7 and 3.8.

The left figure shows the architecture using the interrupt device approach in which an extra device is created for handling interrupts. Implementing this approach would require a modification of the ORF-API functions "orf_init_page" which creates a device, "orf_create_thread0" and "orf_destroy_thread0". "orf_init_page" must be enhanced by the functionality of adding a special device which catches interrupts and handles them. The problem of this modification is, that an additional parameter, specifying the interrupt number is needed. "orf_create_thread0" must be modified to start the RProgs as soon as the device is launched by the interrupt and "orf_destroy_thread0" needs changes to delete the special thread0. Omitting thread0 within the interrupt device is not possible because the priority of a RProg is specified by the thread

it belongs to and besides that the specification defines pernickety that a RProg is called by a thread of a device (see chapter 2.5.1).

Figure 3.8 however shows the interrupt thread approach which defines that besides normal threads a special interrupt thread can be created within a device. This interrupt thread gets a priority and starts the assigned RProg when the interrupt appears. The adoption of this approach needs modifications of the functions "orf_create_thread0", "orf_destroy_thread0", "orf_create_rprog" and "orf_delete_rprog". The advantage would be, that no additional parameter must be added to any orf function, because the parameter which normally specifies the periodicity of a thread can be used to carry the interrupt number. But "orf_create_thread0" must be able to distinguish between the creation of a normal thread and an interrupt thread. This can be done by masking the thread id, e.g. when a defined bit of the id is set, an interrupt thread is meant to be created.

Implementation of interrupt handling

The interrupt thread approach was chosen to be implemented, because it fits much better the specification of ORF. Interrupts can in general only be caught and handled within the kernel. That's why the interrupts cannot be handled by ORF when it is running in user-space with the rt-preempt patched kernel. There are some workarounds to enable catching interrupts within user-space. E.g. by writing a small kernel-module which sends a character to an character device file when an interrupt appears while the user-space process is reading with "blocked" option from this character device. There is another solution[40] consisting of a kernel module which wakes up a process when an interrupt appears, but it is barely documented and needs probably to be modified a lot. So it does not make sense to care about this issue within this diploma thesis.
But it works for ORF when running in kernel-space. Both, Rtai and Xenomai bring their own function set for setting up interrupts, while the principle is the same. Setting up an interrupt starts with registering an interrupt service routine and ends with enabling the interrupt:

- Xenomai:

  ```
  rt_intr_create(&intr0, "ORF_IRQ", int\_nr, \
     (void*)isr_wrapper0, NULL, 0);
  rt_intr_enable(intr);
  ```

  intr0: the handle of the interrupt.

  "ORF_IRQ": the name under which the interrupt will be registered.

  int_nr: the interrupt number which should be caught.

isr_wrapper0: pointer to the interrupt service routine.

- Rtai:

```
rt_request_global_irq(int\_nr,(void *)isr_wrapper0);
rt_enable_irq(int\_nr);
```

int_nr: the interrupt number which should be handled.

isr_wrapper0: pointer to the interrupt service routine.

The modification of "orf_create_thred0" is shown in the left flowchart of figure 3.9. An if-statement checks whether an interrupt thread is meant to be created and if so, the interrupt will be registered, the handler function will be set to the "isr_wrapper" function and the former code of this function will be omitted.

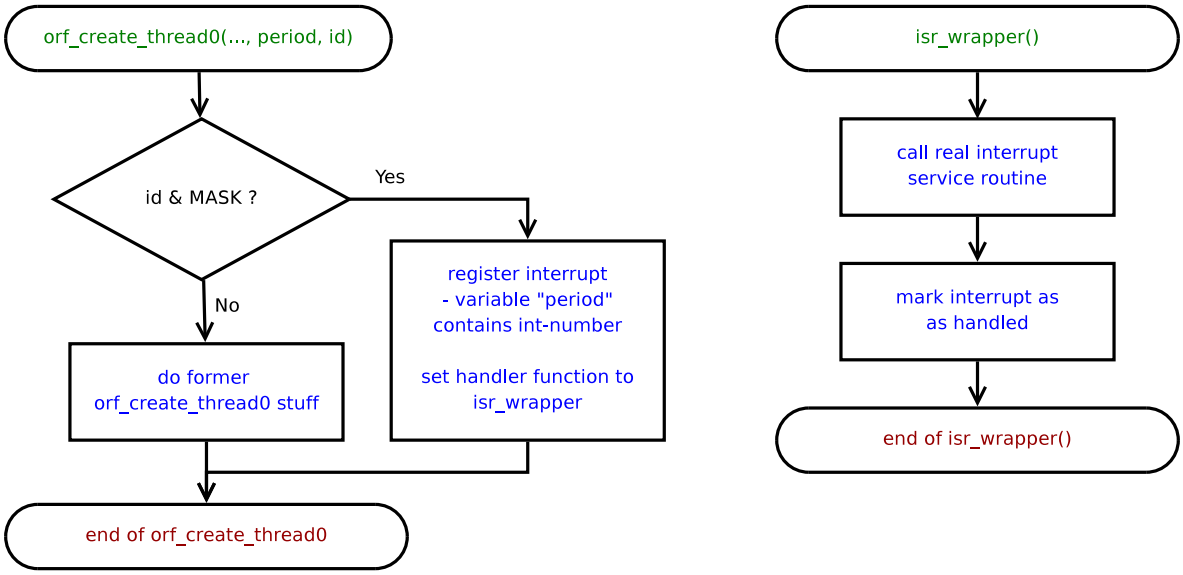

Figure 3.9.: Interrupt handling - modifications to thread

The wrapper function does not do anything but calling the real interrupt handler function and marking the interrupt as handled afterwards, so that the kernel does not care about the interrupt anymore. This wrapper function (shown on the right side of figure 3.9) is needed because Rtai and Xenomai have a different way of marking an interrupt as handled. In fact, for Rtai a special function must be called and for Xenomai a predefined value must be returned by the interrupt service routine. Another issue is that once the interrupt is registered the service routine cannot be changed anymore and using this wrapper, a service routine can be changed by changing the function which the wrapper calls. Changing the routine is needed because when the interrupt thread is created the RProg which handles the interrupt may not yet exist.

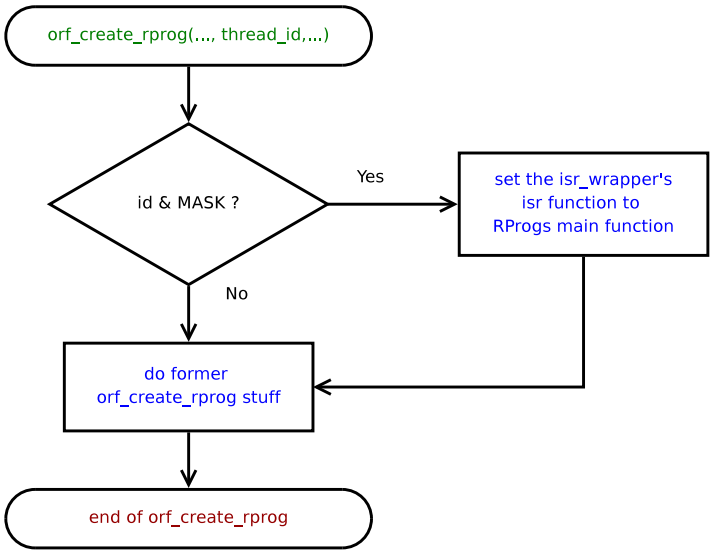

Figure 3.10.: Interrupt handling - modifications to RProg

The only modification on "orf_destroy_thread0" is that it additionally searches the list of interrupt when destroying an thread. If the thread has been found within this list the interrupt will be unregistered and disabled.

Figure 3.10 shows the changes on creation of a RProg. The changes consist of additionally executed code when the RProg is added to an interrupt thread. This additional code just sets the pointer of the function which is called by the wrapper to the corresponding function of the RProg. When deleting a RProg with the "orf_delete_rprog" call the additional code must only reset the pointer to the default empty interrupt service routine of ORF.

Chapter 4.

Benchmarks

As the Intel x86 and the PowerPC target are running with at least Xenomai and Rtai, for x86 also the rt-preempt patch and ORF is modified to offer the functionality for benchmarking the benchmarks can be built. A very important thing is, that all benchmarks must run in a periodic manner, that's generally needed to get meaningful results when measuring with an oscilloscope.
The shown and explained algorithms within this chapter are working fine in theory but in a practical environment there are some disturbances like bouncing output pins of the I/O port and many more. So the actual code includes some additional error handling to care about those disturbances. This additional error handling is not mentioned here, because it would make the functionality more complex and thus more complicated to understand.

4.1. Interrupt latency

The idea behind this benchmark is to measure the time, the target needs to react to an interrupt. To measure the time at least one I/O pin is needed. This pin must be set up to create an interrupt as soon as the state of the pin goes from low to high. Setting up this interrupt may require further steps within the "orf_create_thread0" API function of ORF as well as some hardware configuration. But this differs from hardware to hardware that's why this has to be added to ORF separately for every platform. The setup for the Intel x86 target looks like that:

- Enabling the parallel-port interrupt which is irq 7 by default. This is done by writing the hex value "0x10" to the parallel-port's status register. The status-register is usually addressed at parallel-port's base-address + 2. To automatically perform that, following line must be added to the "orf_create_thread0" function:

  ```
  outb\_p(0x10, 0x378 + 2)
  ```

- Connecting the first I/O pin with pin 10 of parallel-port by a wire. Pin 10 is specially there for creating interrupts, so as soon as the state of first pin changes from low to high an interrupt is thrown.
- Creating a little application which runs in user-space and sets the first pin periodically to high.

Listing 4.1: periodic_int.c

```
ioperm(base,1,1);
while(1){
        outb(1,base);
        usleep(10000);
        outb(0,base);
    }
```

4.1.1. ORF integration

The implementation of the RProg is very simple because it just has to set the first I/O pin to low as soon as it is called.

Listing 4.2: rt_irq.c

```
int irq_para_idle(int device,int id,long para)
{
   /* just output 0 on the I/O port (is called by interrupt service
    * routine when the pin1 is going from low to high */
   orf_outb(0);
   return 0;
}
```

To initialize ORF the modules "orf_method_real", "krn_orf" and "rt_irq" must be loaded and the .ini file which is loaded with "orf_startup" must look like this:

Listing 4.3: rt_irq.ini

```
[...]
ORF_DO_INITFUNCT;*

# Create Thread 0 for Page 0
ORF_CREATE_THREAD0;0;5;4a60;1
# Create interrupt thread catching interrupt 7
ORF_CREATE_THREAD0;0;0;7;40000001

# irqpara_idle function handles interrupt of irq thread
ORF_CREATE_RPROG;0;2;40000001;IRQ_PARA_IDLE; ; ;
[...]
```

Now the "periodic_int" program can be started on the target system. The program sets the first pin to high what causes an interrupt which is handled by the RProg and sets the first pin back to low.

4.1.2. Scope implementation

One channel of the scope is connected to the first pin of the target's parallel-port. Figure 4.1 shows the graph of the scope triggered to rising edge. This signal is periodically repeated. The time between the rising edge and the falling edge of the signal corresponds directly to

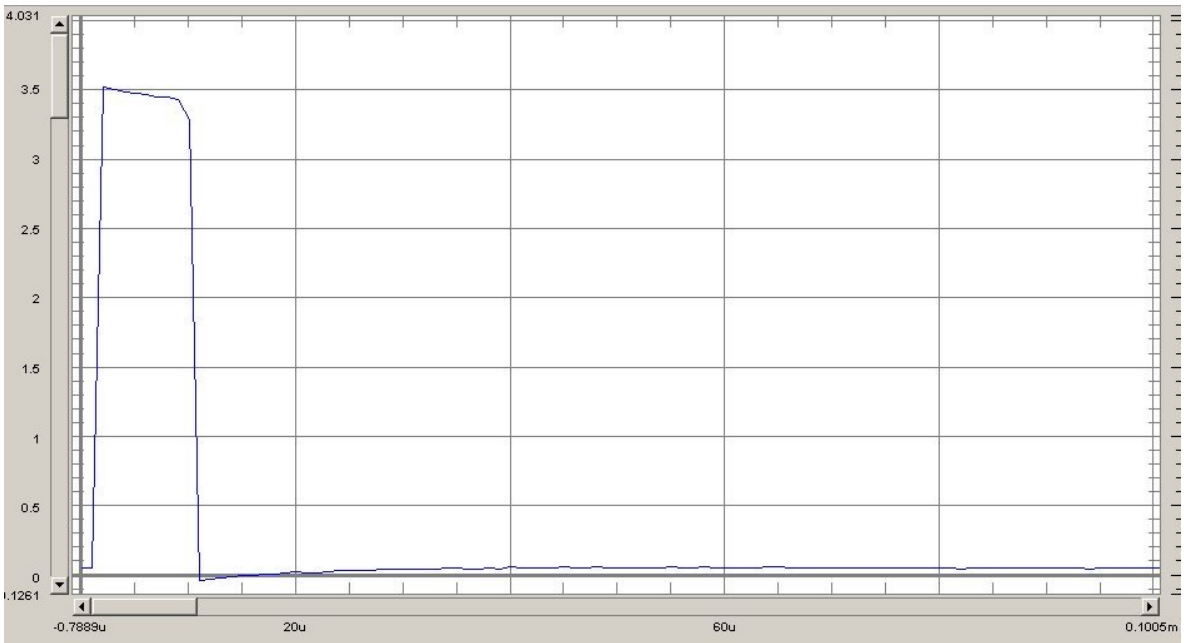

Figure 4.1.: Interrupt latency scope graph

the interrupt latency. With the human eye it is possible to read the interrupt latency of this graph, about 8 to 10μs. But this should be done automatically by a C-program so that the benchmark can be ran a while to get meaningful results, also from a practical point of view. When the test ran a while a minimal and maximal as well as an average interrupt latency can be calculated of the measured data. Figure 4.2 shows the principle control flow to automatically measure the interrupt latency of many cycles and calculate the maximal, minimal and average latency. The single steps of the program are described below.

1. The initialization of the scope consists of setting all of the scope's attributes. This includes for this benchmark these parameters:

   Timebase: Specifies the resolution of the scope. In this case it is set to 1μs, the scope's maximum resolution.

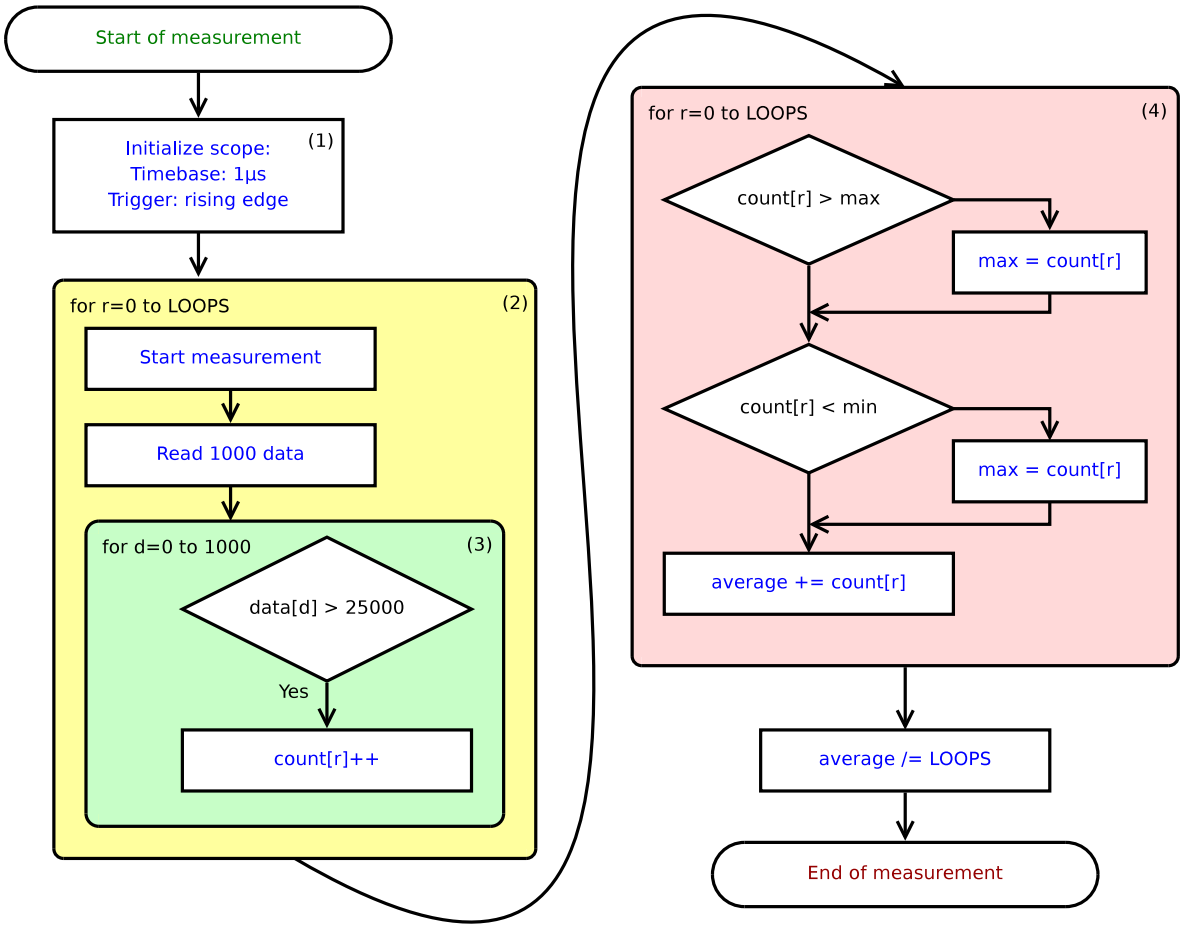

Figure 4.2.: Flowchart, scope implementation of interrupt latency measurement

Amplitude / Offset: These parameters can be modified to fit the signals amplitude and offset, in this case the amplitude is set to 5V because the parallel port has 3.5V. Offset is set to 10V, to have the zero-point at the bottom of the graph. The offset is always calculated based to the standard amplitude of 20V.

Memory depth: It defines the amount of data that should be collected. Exactly this amount has to be read by the program. If a different amount is read, the result will be totally screwed up. It is set to 1000 for this measurement, because it is not expected to have an interrupt latency greater than 1ms.

Trigger type: This indicates how the scope should be triggered. There are several trigger types. Most useful are 'E' - rising edge and 'e' - falling edge. 'T' and 't' for triggering when the voltage is getting higher or lower than a specified threshold may be also usefully. For this measurement the type is set to 'T' and the threshold to 2V. But there is no real reason to do so, same result could be achieved by using 'E'.

2. This is a for-loop which repeats as many times as defined by the "LOOPS" precompiler alias and every loop consists of one single measurement of 1ms.
3. After the data is read from the scope and stored into an array this loop counts how long the signal has high level, what directly corresponds to the interrupt latency. The length is stored into another array called count[].
4. When all measurements of this run are finished this loop searches for the maximal and minimal interrupt latency. It does also save the result of every run to a log-file and add all latencies to an floating point value.

At the end the floating point value calculated in loop (4) is divided by the amount of measurements runs were done ("LOOPS") to get the average value.

4.2. Jitter

The jitter is one of the most expressive attributes of a real-time system. The target generates a square wave signal on one of its I/O ports and the scope has to measure the duration of every half cycle (e.g. how long is the pin on high level) and calculates the maximum difference between those durations.

4.2.1. ORF integration

The implementation of the RProg consists just of a line that increments the data on the I/O port every time the RProgs main function is called:

Listing 4.4: rt_jitter.c

```
int jitter_idle(int device,int id,long para)
{
   /* increment the data in the I/O register
    * creates square-wave on the I/O
    */
   orf_outb(orf_inb()++);
   return 0;
}
```

The initialization of ORF after loading the modules "orf_method_real", "krn_orf" and "rt_jitter" is done by launching "orf_startup" with an ".ini" file containing those statements:

Listing 4.5: rt_irq.ini

```
[...]
ORF_DO_INITFUNCT;*
```

```
#Create Thread 0 for Page 0
ORF_CREATE_THREAD0;0;5;9c4;1

#Create Real-time Progs
ORF_CREATE_RPROG;0;1;0;JITTER_IDLE;  ;  ;
[...]
```

Right after the "orf_startup" application did its work the target begins to generate a square wave signal on the I/O port.

4.2.2. Scope implementation

Again only one channel of the scope needs to be connected to pin1 of the targets I/O port. Figure 4.3 shows the graph of the generated square wave and the durations which are measured and analyzed. In fact the duration of one period is 2ms in this case. There are two

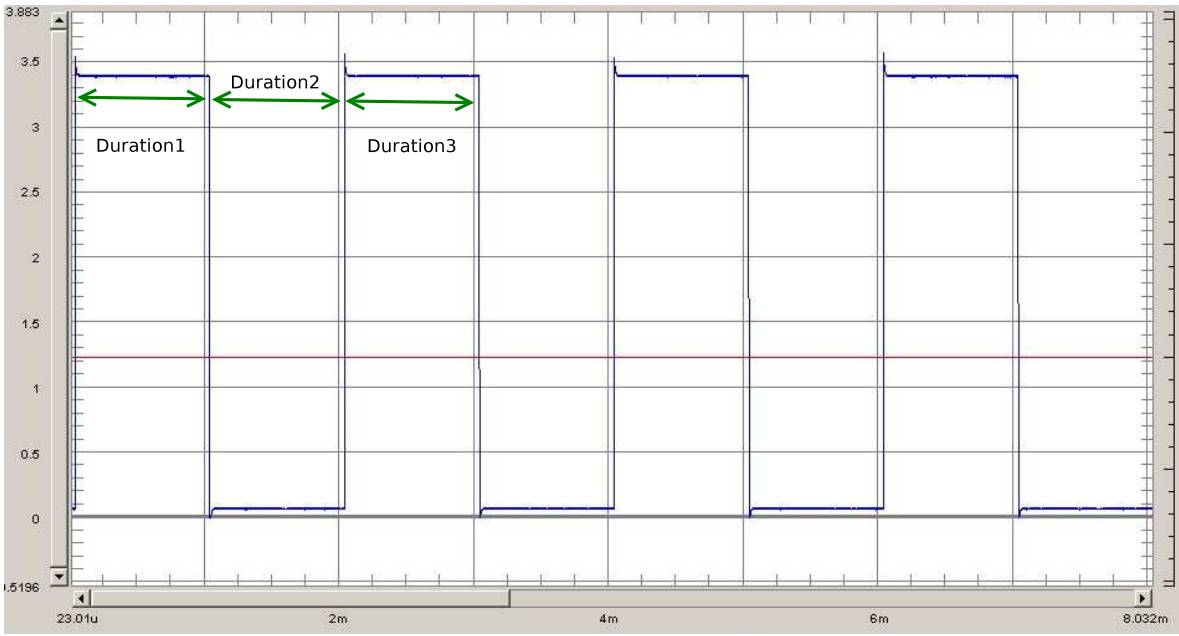

Figure 4.3.: Jitter scope graph

ways to write a program to measure the length of the half period. On the one hand, every edge, doesn't care if rising or falling, can be used as trigger point and the amount of values between two trigger points is counted. On the other hand the program can be triggered on rising edges and the amount of data where the signal is high or low is counted separately. Both ways should do the job as long as there is no interfering signal. But as the hardware may differ in the time it needs to set or reset a port it makes sense to use the second approach.

The principle program flow looks like shown on figure 4.4 and the single steps are described below.

1. The scope is initialized to capture 50000 data at a timebase of 1μs and to trigger at rising edge. Amplitude and offset are set to the values described for the interrupt latency test.

2. The precompiler alias "LOOPS" defines how many measurements should run within one program launch. It is set to 1000 loops what makes the test running about two minutes. At the beginning of this loop the "RUN" command is sent to the scope and the data is read into a local array and then proceed by the steps 3 and 4.

3. This step is for counting how long the signal is high and low for each period. It is implemented as a loop, counting from 0 to 50000. It looks for rising edges, what's actually done by checking if the current value is high level and the previous one is low. If such a rising edge is found, the period counter is incremented. Furthermore it is checked whether the current value iss high or low and the respective local variable "tmpLO[period]" or "tmpHI[period]" is incremented.

4. This step is just for getting rid of the first and the last period of each measurement. Because those two may be invalid, caused by not fully caught cycles. It simply copies the data of every period of the local variables into the global variables "LO[]" and "HI[]" by omitting the first and the last one.
   Thus as result after the last run of this step, "LO[]" and "HI[]" contain the duration of the particular phase of every period of the 1000 loops of step 2.

5. The last job of the application is to analyze the data. It searches the complete "LO[]" and "HI[]" array for their maximal and minimal duration. Additionally it adds the durations into a floating point variable which is divided by the amounts of periods to get average durations. For allowing other applications like Excel to work with the measured data, all durations are saved to a log-file.

4.3. Maximal frequency

The idea behind this benchmark is to figure out the target's and real-time solution's highest frequency of periodic appearing real-time task calls. Therefore a very small real-time task is created which does in principle just toggle a pin of the I/O port. Then a bash script is launched on the target, which raises the frequency step by step while the scope application measures the maximal frequency and saves it.
One of the problems that come with this benchmark is that as soon as the frequency is getting higher than the time needed for a task switch the target will completely freeze. So there must

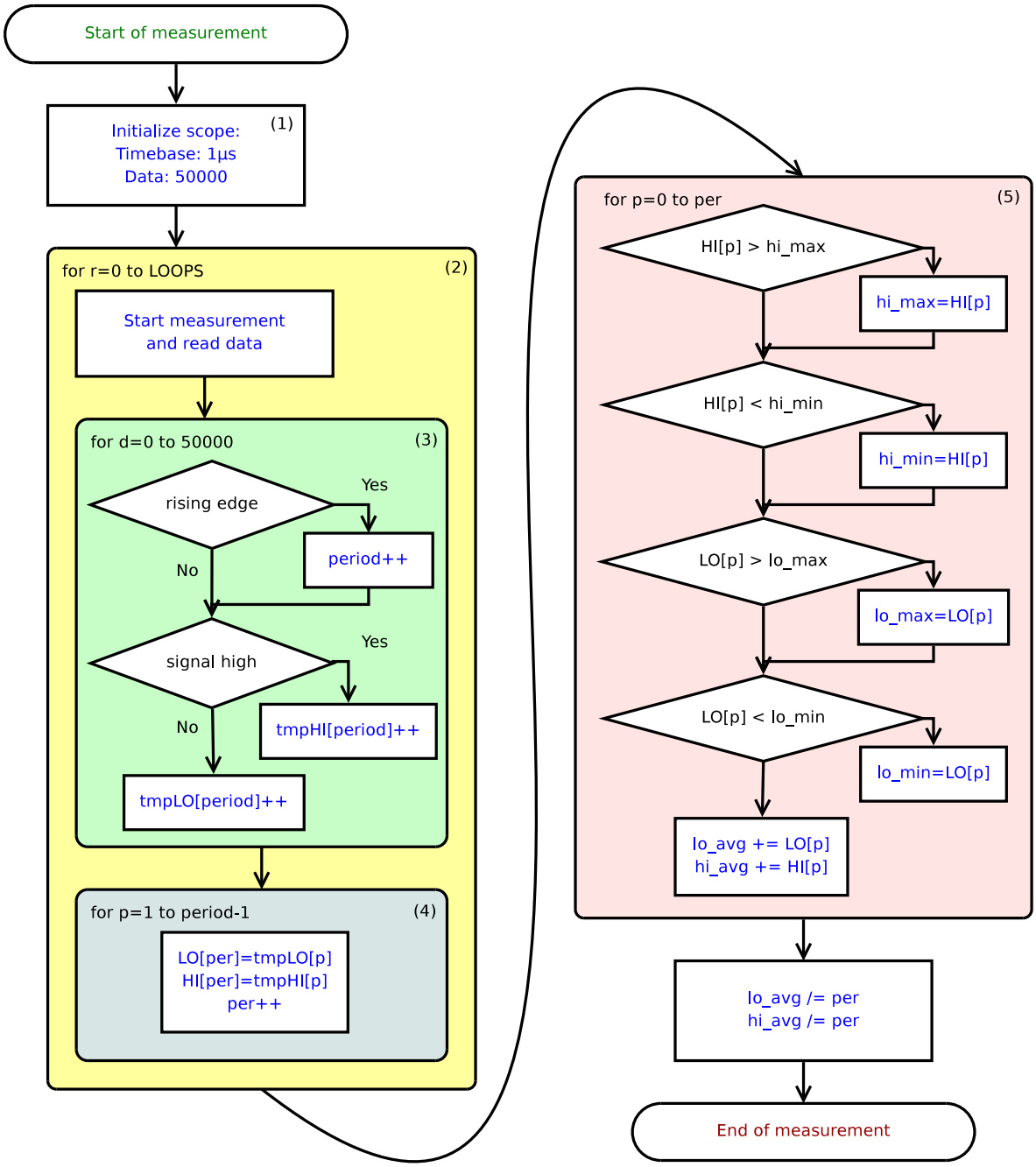

Figure 4.4.: Control flow of jitter implementation

be an alive signal which is also checked by the scope. The normal frequency cannot be taken as a keep-alive signal, because when changing the frequency this signal disappears for a while. Another issue is to tell the scope that the frequency has changed. That's done by sending a high signal for some time when the frequency is going to change.

4.3.1. ORF integration

The implementation of the RProg consists of three states. The first state is true when the module is called the first time. At this state the output pin is set to high level and the starting time is stored. The RProg lasts in second state until 3 seconds have been passed since the module was called the first time and actually does nothing than holding the output pin on high level.
The last state is responsible to toggle the output pin. Its code is executed after the 3 seconds since start up have gone by.

Listing 4.6: rt_freq.c

```
int freq_is_init=0;
long long starttime;
int freq_idle(int device,int id,long para)
{
   unsigned char data;
   /* state 3: toggle output pin */
   if(freq_is_init==2)
   {
      orf_outb(orf_inb()^0x1);
      return 0;
   }
   /* state 1: get start time and set output pin */
   if(freq_is_init==0)
   {
      orf_outb(1);
      starttime=orf_get_cpu_time();
      freq_is_init=1;
   }
   /* state 2: do nothing, just wait till 3
    * seconds have gone by */
   else if(freq_is_init==1 &&
      ((orf_get_cpu_time()-starttime) > 3000000000))
   {
      freq_is_init=2;
   }
   return 0;
}
```

The source of the alive RProg consists just of the lines needed to toggle one pin of the output every time the RProg is called:

Listing 4.7: rt_alive.c

```
int alive_idle(int device,int id,long para)
{
   unsigned char data=0;
   data = orf_inb();
   data ^= 0x2;
   orf_outb(data);
   return 0;
}
```

The initialization of this benchmark is in first place the same like for the benchmarks before. As usual the modules "orf_method_real", "krn_orf", "rt_alive" and "rt_freq" have to been loaded. To start the first frequency a manual execution of "orf_startup" with a ".ini" file as parameter is needed. The ".ini" file must initialize the shared memory, two devices, a thread0 for every device and the two RProgs. The priority of the thread containing the frequency RProg is of course higher than the task with the alive signal to ensure the alive RProg does not disturb the measurement.
After the scope application is started, the bash script called "start_measurement.sh" which takes care about changing the frequency can be started. While a frequency change only the first device is shut down, the second one serving the alive signal is still running.

Listing 4.8: start_measurement.sh

```
# frequencies to test in hex
for i in "2710" "1388" "9c4" "753" "4e2" "3a8" \
   "271" "1d4" "138" "EA" "9c" "75" "4e" "39" \
   "27" "1b" "13" "9"
do
   echo "frequency: $i"
   # modify startdev0.ini to start rt_freq with new freq
   sed "s/__MS__/$i/" startdevice0.ini > startdev0.ini
   # stop device0, delete RProg and destroy Thread0
   ./orf_startup stopdevice0.ini
   # to ensure the frequency module is correctly
   # initialized remove and insert again
   rmmod rt_freq
   insmod rt_freq.ko
   # create RProg, Thread0 with new freq and start device0
   ./orf_startup startdev0.ini
   sleep 20
done
```

4.3.2. Scope implementation

This time both channels of the scope have to be connected to the target's I/O port. The first channel to pin0 of the I/O and the second channel to pin1. Figure 4.5 displays the graph of the output of ORF running the RProgs of this benchmark. The blue line shows

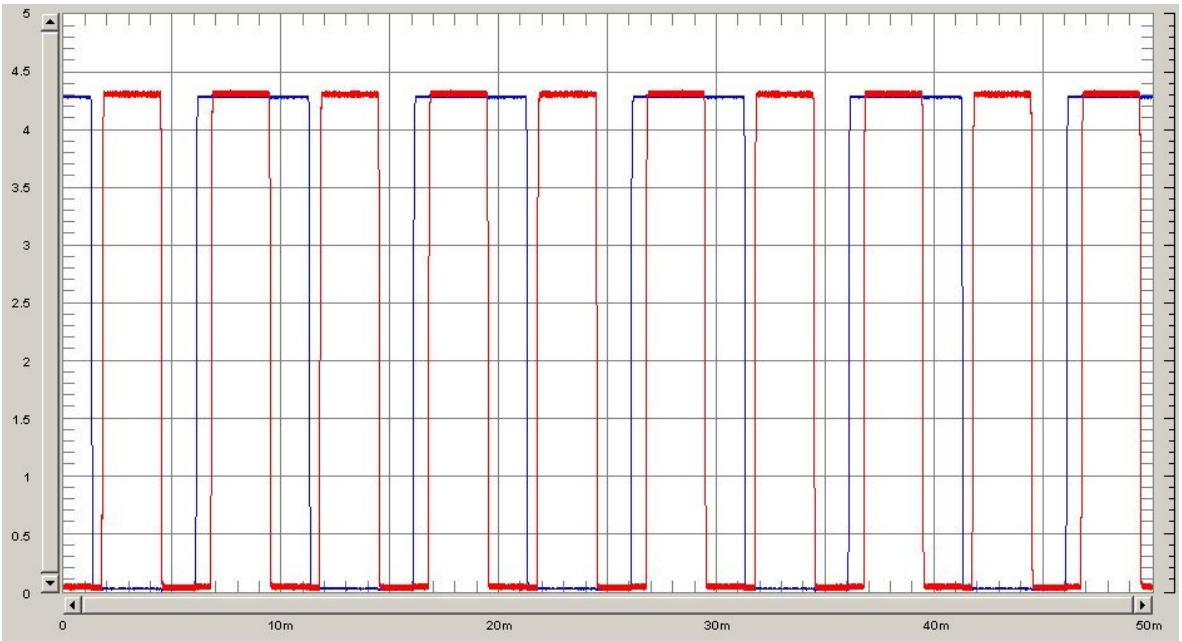

Figure 4.5.: Scope graph of frequency benchmark

the alive signal and the red one the signal which frequency is raised continuously. The blue graph illustrates a square-wave signal with a cycle duration of about 10ms what corresponds to a frequency of 1/0.01s = 100Hz. But as such a cycle consists of two RProg calls, one for the rising edge and one for the falling, the frequency is actually 200Hz. Thus the scope's application has to trigger on rising as well as on falling edges. In general the measurement application for this benchmark should work the way shown on figure 4.6.

1. At the beginning the scope is initialized to capture 50000 values per channel within one measurement. The timebase is set to 1μs and the trigger has to be disabled. The trigger must be disabled because otherwise reading data from the scope will block the measurement program. If so, the application would hang when the target dies and actually the target is going to die at the end of the benchmark. The disadvantage of disabling the trigger is, that the first and the last cycle cannot be used for measurement, because they are most likely incomplete cycles.

2. That's the program's main loop is an endless loop, so once started it has to be explicitly halted. First step is to send the "RUN" command to the scope and read the acquired data. Next step is counting the edges (see 3.) and checking if the alive signal is still sent. If the alive signal is missing, the loop is halted.
   After this it is checked whether the duration of first signal's high level is longer than 40,000 what means that a frequency change is happening. If a frequency change happens, the last stable frequency is copied into a variable and the loop is restarted.

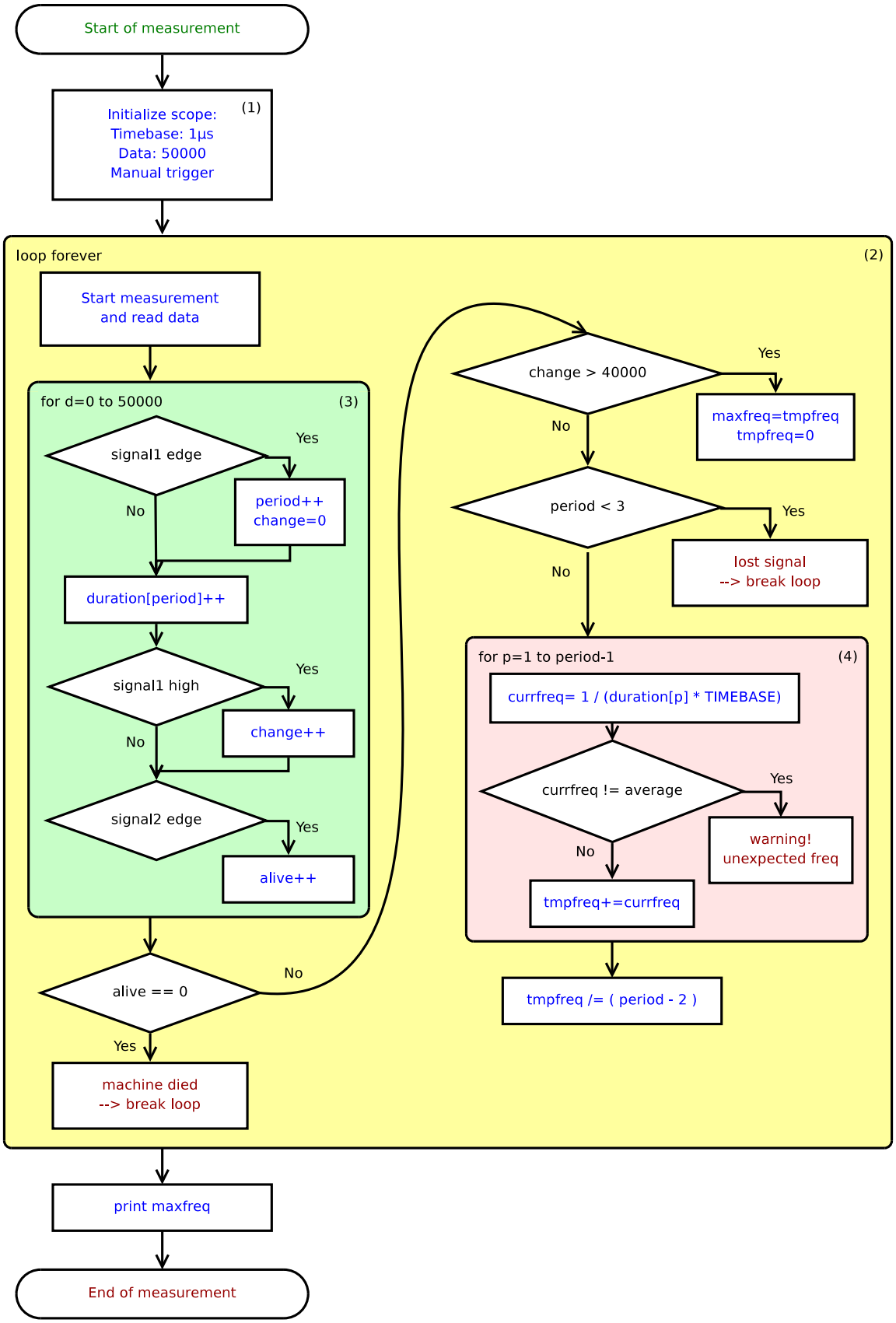

Figure 4.6.: Control flow of the frequency benchmark

Next thing what's checked is the amount of periods within this run. If there are less than three periods the measurement cannot go on. And by the fact, that even the slowest frequency still has more than 3 periods there must be something wrong and the measurement can be stopped.
The last step is to calculate and verify the current frequency (see 4.).

3. Within this for-loop signal 1 is checked for edges. If an edge has been found two things happen. First the period counter is incremented and second the "change" variable which states a frequency change is set to zero. After that, the duration of the current period is increased by one. The remaining functionality of this block consists of incrementing the "change" variable if signal 1 is high and increasing the "alive" variable on an edge of signal 2.

4. This for-loop is counting from the second period to the second last. As already mentioned before the first and the last period could disturbs the results, because they are mostly incomplete. First thing is to calculate the frequency of the current period and compare it with the average. The average is calculated every time, when the comparison is called by the formula "tempfreq/(p-1)". Of course the comparison is done very tolerantly, because this benchmark is not taking care about the jitter of the signal. But if a period is completely missing the program will throw a warning. The last thing is to add the current frequency to the "tmpfreq" value.

4.4. Inter-process communication

The purpose of this test is to measure the amount of data which can be transmitted through the device files or the pipes. But unlike the previous benchmark this one is not measured by the scope. In principle an user-space application like "orf_startup" must write data to the device or pipe file of ORF, which writes the same data back. Thus the functionality of the device or pipe file can be tested. And by additionally sending a specific amount of data and counting the packages it is possible to measure the speed of the communication. But therefore a function within ORF is needed which just echos received data back through the device or pipe file. And additionally a way must be found to count the transmitted packages within a specific time.

4.4.1. ORF integration

First of all the echo function must be added to the ORF framework. Therefore a new message for the pipe communication must be defined, so that the pipe or character device handler knows the command. This is done by adding a precompiler define into the "orf_methods.h"

header file looking like "#define MSG_ORF_ECHO_TEST 0x036". After this the handler's function "orf_handle_message_rtf" has to be extended by the functionality of calling the function "orf_echo_test" and answering the client by sending the received data when the echo test command was put to the pipe.
The "orf_echo_test" itself is only needed to count the received packages. It is defined that one package consists of 256 bytes of data and the "orf_echo_test" function counts 64 of those packages and increments then an integer within the shared memory of ORF. Counting to 64 before incrementing the variable within the shared memory results of a compromise made. This compromise must be accepted to have on the one hand a long benchmark run before the variable of the shared memory overruns and on the other hand still precise results. 64 seemed to be a good compromise, so the resolution of the result is 32KiB.

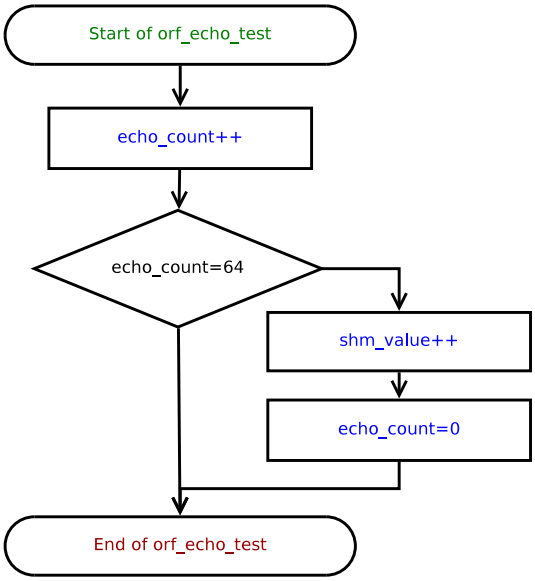

Figure 4.7.: Flow diagram of the echo test function

A RProg called "rt_ipc" is launched every second and reads the integer from the shared memory, calculates and prints out the current transmission speed. The calculation is done by this formula:

$$speed = (shm_value - previous_shm_value) * 64 * 512 Bytes \tag{4.1}$$

While the 64 in this formula corresponds to the 64 counts done in "orf_echo_test" and the 512 Bytes results of the fact, that for every call of "orf_echo_test" the 256 Bytes were transmitted from user-space to ORF and the other way around.
The test can be started by loading the modules "orf_method_real", "krn_orf" and "rt_ipc" and starting "orf_startup" with "ipc.ini". Within the ".ini" file the "rt_ipc" RProg is started on a thread which is initialized to be started every hex f4240μs what's exactly 1 second. The

RProg directly begins to print the transmission speed into the kernel log every second.
To start the transmission a program that writes the echo-test command and data to the device files is needed. Therefore "orf_startup" is modified. It needs to be enhanced by the functionality of sending and receiving continuously the echo-test command when "ORF_ECHO_TEST" is called within the opened ".ini" file. Finally the transmission can be started by running the modified "orf_startup" tool with "echo.ini", which contains the command and 256 Bytes of data:

```
ORF_ECHO_TEST;0000000000000000111111111111111[...]ffffffffffffffff
```

4.5. Overload behavior

This benchmark is intended for checking the behavior of the real-time approaches on their behavior under temporary overload. It is tested by using two threads having the same priority and the RProgs which are launched by the threads set one pin of the output and reset after acquiring the full CPU for a while. There are two different types of overload tests.

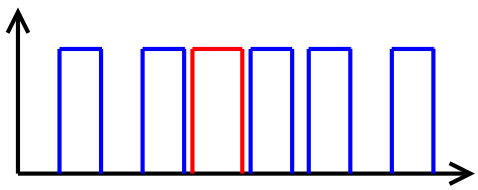

Figure 4.8.: Principle graph of overload test 1

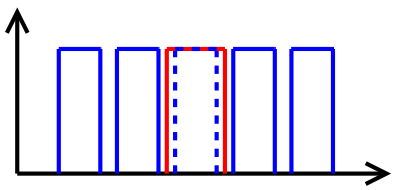

Figure 4.9.: Principle graph of overload test 2

For test1 the CPU usage duration of the second thread is a little longer than the inactive time of the first thread. This way the next launch of the first thread's RProg is delayed. Figure 4.8 shows the graph which should be produced by the test, while blue is thread 1 and red thread 2.
And for test2 the frequency of the first thread is raised, so that the combination of a thread 1 and a thread 2 shot result in the state, where one period of the first thread must be omitted. On figure 4.9 the omitted period of thread 1 is shown with dotted lines.
Those tests don't have any values as outcome, there is just a boolean result whether the signal was proceed even after this overload or not.

4.5.1. ORF integration

Two RProgs are needed to adopt the functionality of this test. The first one simulates the normal usage of the real-time system and the second one simulates short-time overloads. As

already described above they just set a pin of the I/O port, e.g. "rt_oload1" sets pin0 and "rt_oload2" pin1, then they use the full CPU in a for loop for e.g. 200μs and reset their particular pin.
As both threads are initialized with the same priority none of both is allowed to preempt the other. The initialization is done like usual. After loading the modules an ".ini" file is loaded with "orf_startup" which creates two devices within ORF, both with a thread0 and adds the RProgs. The only difference between test1 and test2 is the frequency of the first thread, while the duration of the second thread is both times 2ms. The cycle duration of the first thread is for test1 368μs and for test2 250μs.

4.5.2. Scope implementation

Both analogous channels must be connected to the target's I/O-port. As for the benchmarks before the first channel to pin0 and the second channel to pin1. The graph on figure 4.10

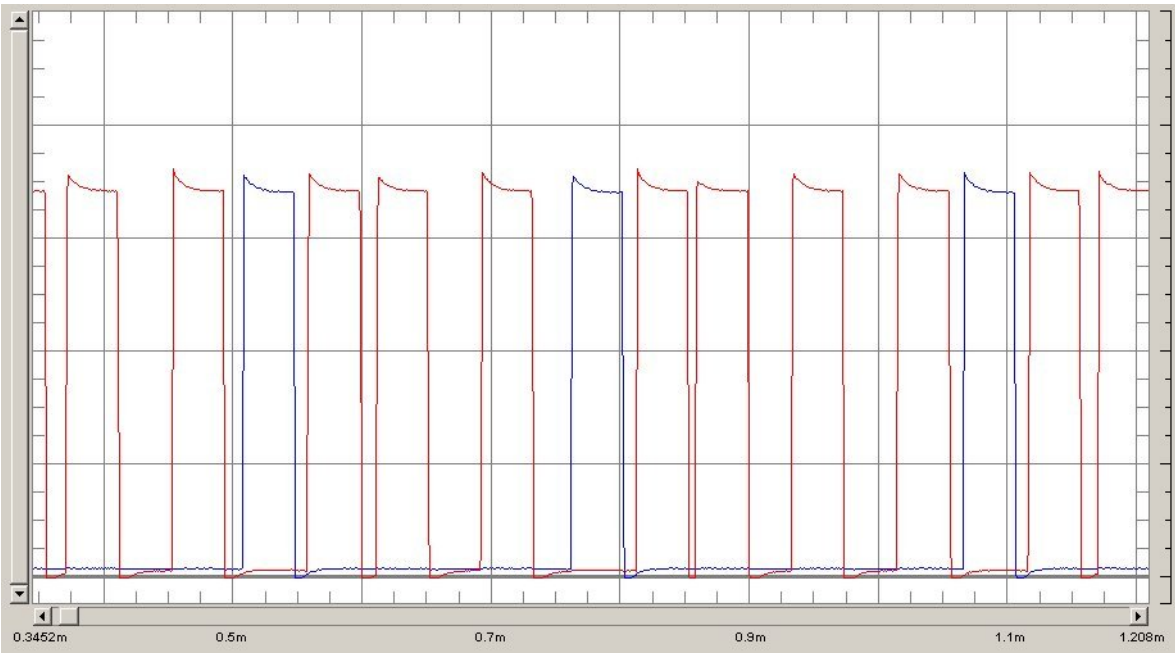

Figure 4.10.: Scope graph of the overload test

demonstrates how the output of the first overload test looks like. The red signal is the signal simulating normal target's workload and the blue one simulates the short-time overloads.
The implementation of the scope's application is not that easy as it may seem in the first place. The problem is that there is no easy way to find out if there is one cycle missing or not, so as a compromise the application should just tell, whether the signal is there, or not. For a real industrial environment there shouldn't be such an overload anyways.

Figure 4.11 shows the flow diagram of the application to measure and analyze the data. The steps are described in detail in the following lines.

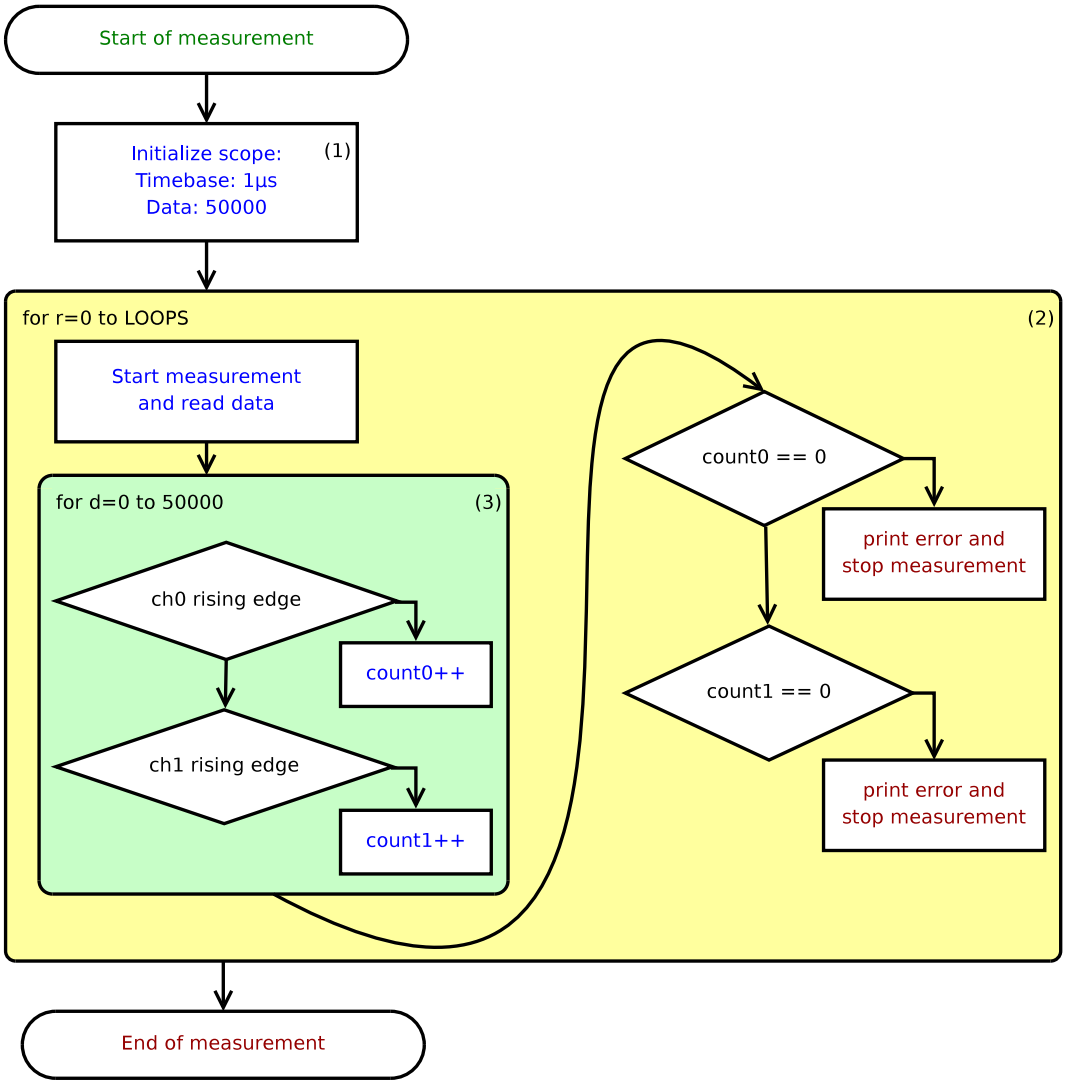

Figure 4.11.: Flow diagram of overload's scope application

1. The scope must be initialized to capture 50,000 values on a timebase of 1μs.

2. This for-loop repeats as often as defined in the precompiler alias "LOOPS". For every run it starts the measurement, reads the data from the scope, counts the rising edges of both signals (see 3.) and checks in the last step whether both signals exist.

3. This loop checks every value of the measured data and increments the particular variable when a rising edge is found.

4.6. Priority functionality

The priority functionality test is meant to test if the priorities are handled correctly by the real-time solution. That's done by launching some RProgs with different priority, different periodicity and different processing time and then it is checked whether a thread gets preempted by a thread with lower priority. If this happens there is a problem with the real-time solution.

4.6.1. ORF integration

Since the current ORF implementation supports only up to four devices it makes sense to create four RProgs. In principle the RProgs set one pin of the I/O-port, acquire the CPU for a while and then set the pin back to low. But to allow precise measurement, it is needed to tell the scope whether a RProg is running or not. This can be either done by toggling the output pin as long as the RProg owns the CPU or by indicating it on a second output line. As there are at least 8 output lines on each target and four RProgs the approach with using a second pin fits and is much easier to implement. The reason why such an indication is needed is that when the signal of a RProg goes to low the scope does not know, is the RProg preempted by another RProg or is it finished.
So the definition is pins 0 to 3 are used to indicate when a RProg is active and the pins 4 to 7 are used to show which RProg is really running. Active in this matter means that the RProg is running in principle but may be preempted by another thread and running means that the RProg does really own the CPU and is processed. To adopt these behavior the code of the RProg's main function looks like that:

Listing 4.9: rt_prio1.c

```
int prio1_idle()
{
   /* store start time */
   unsigned long long starttime=orf_get_cpu_time();
   /* set the 0. and the 4. pin */
   orf_outb(orf_inb()|0x11);
   /* 100% cpu usage for 40us */
   while((orf_get_cpu_time()-starttime) < 40000 )
   {
      /* leave pin 0-3 as they are
       * reset 5-7 and set bit 4 */
      orf_outb((orf_inb()&0x0f)|0x10);
   }
   /* reset pin 0 and pin 4-7, 1-3 are unchanged */
   orf_outb(orf_inb()&0x0e);
   return 0;
}
```

The other 3 RProgs look very similar, the output pins and the time how long the RProg owns the CPU have to be changed. The table shows their particular values:

	rt_prio1	rt_prio2	rt_prio3	rt_prio4
Output pin: active	0	1	2	3
Output pin: running	4	5	6	7
Priority	5	4	3	2
Periodicity (ms)	1	11	130	262
CPU usage duration	40μs	1100μs	13ms	140ms

The values of the periodicity and the CPU usage duration are chosen that the RProgs preempt very often each other. That shortens the time the benchmark must be run to produce meaningful results. After loading all needed modules ORF can be initialized to use 4 devices, each with one thread0 using the particular priority and the periodicity of the table above.

4.6.2. Scope implementation

For this benchmark the scope's two oscilloscope inputs are not enough to measure the 8 signals. That's why the digital data logger option is used. It offers at least 16 input pins, while the resolution is not as high as for the oscilloscope mode. However for this benchmark

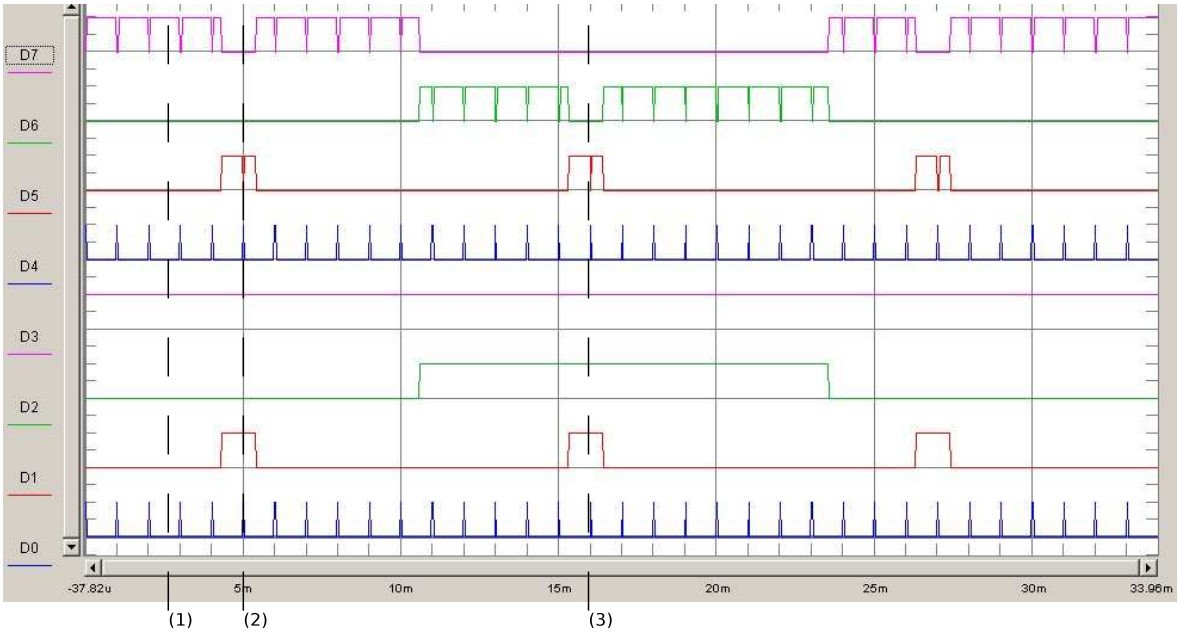

Figure 4.12.: Graph of the priority test

the resolution of the data-logger mode is quite enough. After connecting the I/O pins of the

target with the input pins of the data-logger a graph like shown on figure 4.12 is displayed on the scope's user interface.
The three example states shown in figure 4.12 are described here:

1. rt_prio4 is active and running.
2. rt_prio1, rt_prio2 and rt_prio4 are active, but only rt_prio1 is running as it has highest priority.
3. All four RProgs are active but rt_prio1 is preempting all of them.

The scope application has to check every thread for which other thread is preempting it. At the end a table containing how often a thread was preempted by every other thread can be printed. Figure 4.13 shows the control flow of the main application.

1. The scope is initialized to run in data-logger mode at a timebase of 10μs and to store 50,000 values in its memory.
2. This loop is repeated as often as defined in the "LOOPS" precompiler alias. It first starts the measurement, reads the 50,000 values and then stops the measurement. That is unique for the data-logger mode. It does not stop measuring after the defined amount of data is acquired. The advantage of this behavior is that more data can be measured than actually fit into scope's memory. But as the application has to calculate, it is not able to handle the amount of data as fast as the scope provides it. So the measurement has to be stopped after the 50,000 values have been read. To do a clear stop the data measured by the scope between reading the data and stopping the measurement must be rejected. After these steps are done the loop which analyzes the data (see point 3) is started.
3. This loop counts from the first value to the last value of the array containing the measured data. It executes the code shown in figure 4.14 for every RProg and for every data record to check whether it is preempted and by which RProg it is preempted.
4. To check if thread x is preempted by another task, the first step is to look whether the thread is active at the moment by checking the active pin. If it is active the running pin of the thread is inspected. If the thread is not running it is preempted by another one and the loop described at point 5 is invoked.
5. This loop proves which RProg is preempting thread x by checking each RProg's running signal. If one is found the corresponding variable within the table of preemptions is incremented.

At the end the table containing the preemptions is checked whether a RProg is preempted by another RProg with lower priority and whether every thread has been preempted by every RProg with higher priority. Of course rt_prio1 should not be preempted by any other

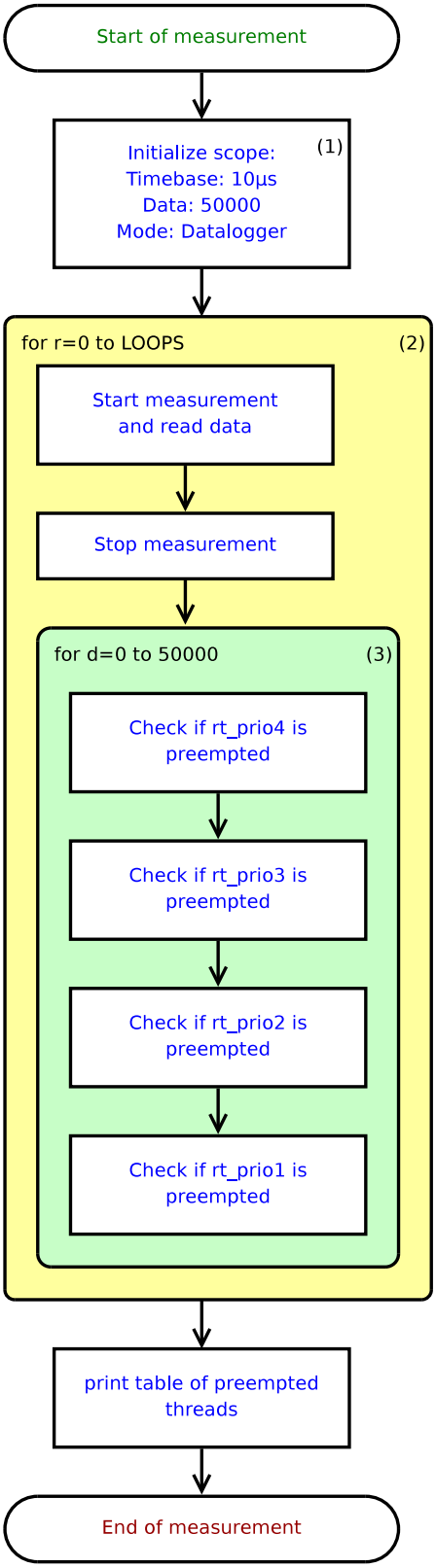

Figure 4.13.: Flow of the priority benchmark

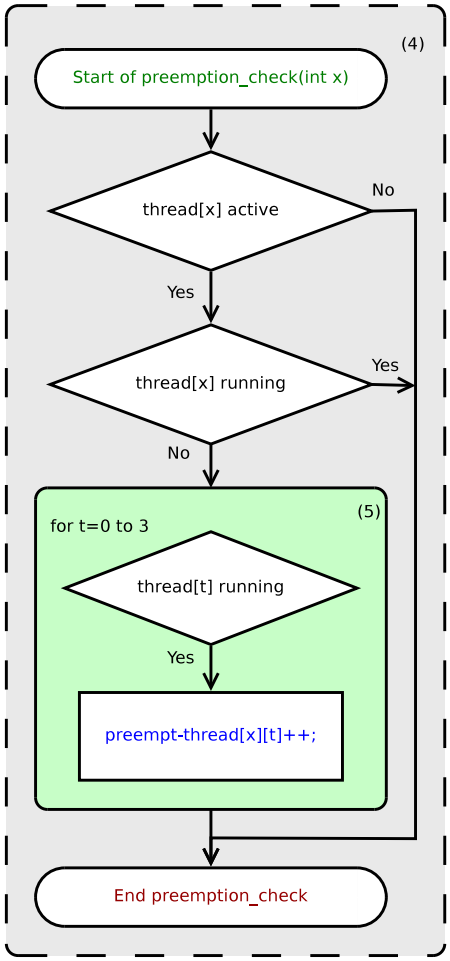

Figure 4.14.: Flow diagram of the preemption check

thread, as it has highest priority. The complete result is also saved into a log file for further processing.

Chapter 5.

Results

As the Intel x86 and PowerPC targets are working and the benchmarks are implemented it is finally time to run the benchmarks and compare the results.
To get an outcome that is meaningful also from a practical point of view it makes sense to run each benchmark twice. Once without workload of the target system and once with heavy workload. The heavy workload is produced by copying data of the "/dev/zero" device to the "/dev/null" device by this command:

```
cat /dev/zero > /dev/null & cat /dev/zero > /dev/null & \
cat /dev/zero > /dev/null &
```

The line starts three times the program "cat" each copying an infinite number of data from one device file to another. All three processes acquire full CPU usage, what rises the *load* of the target up to 3. To create another workload an additional computer is setup to send a ping flood onto the targets network device, what causes an interrupt every time a package appears. That's done by executing "`ping -f target_ip`" on the additional computer.
Following table gives an overview of which benchmarks have been performed on which target:

	ppc xn26	ppc xn24	ppc rtai	x86 xn26	x86 rtai26	x86 rtai24	x86 rt-preempt
Frequency	X		X	X	X	X	
Int latency				X	X	X	
IPC	X		X	X	X	X	X
Jitter	X	X	X	X	X	X	X
Overload	X			X	X	X	
Priority	X			X			

Unfortunately there is a problem with two threads accessing the parallel port when ORF is running in user-space. ORF crashes as soon as a second thread accesses the parallel port, even when using a mutex semaphore. That's why all benchmarks which depend on two or more threads with I/O access can not be performed until the issue has been solved.

5.1. Frequency

The diagram on figure 5.1 shows the outcome of the benchmark which checks the system for

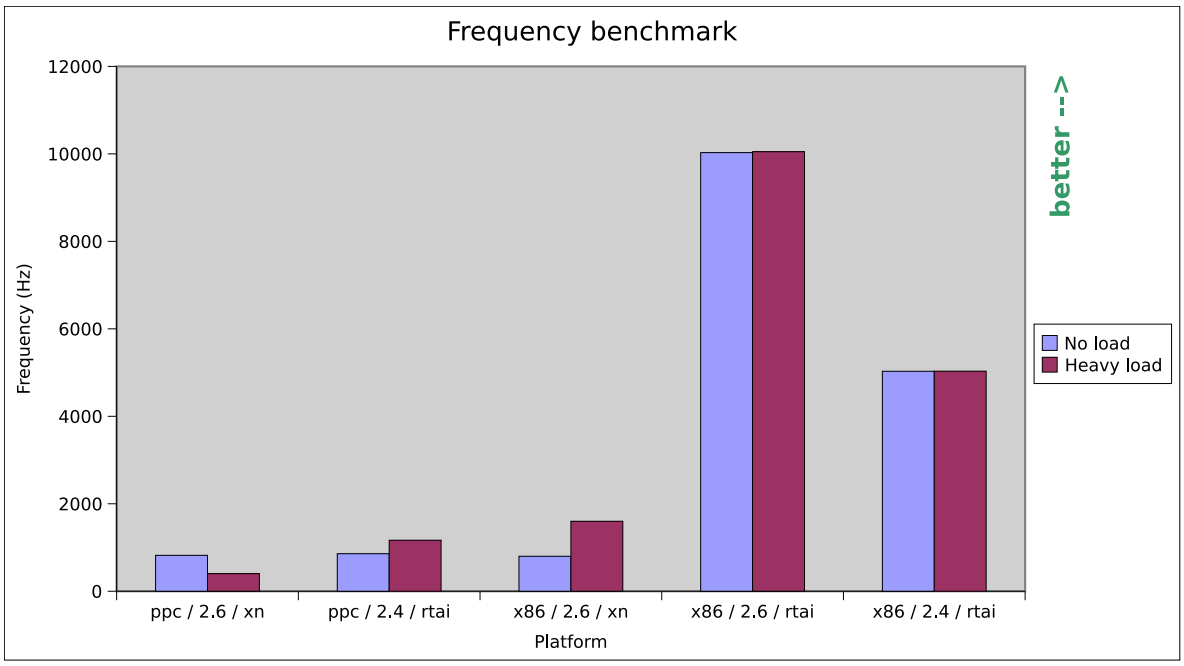

Figure 5.1.: Results of the frequency benchmark

highest possible frequency of the targets, without producing errors. The convention of the platform naming is "architecture / Linux kernel version / real-time extension", while "xn" stands for Xenomai.
The graph shows clearly that the Intel x86 architecture in combination with Rtai reaches the highest stable frequency of 10kHz with Linux kernel 2.6 and 5kHz with a 2.4 Linux kernel. Xenomai on Intel x86 target accomplished just stable 1.6kHz under heavy load and 800Hz without workload. After checking the log file of those runs it showed up, that the platform is able to perform the frequency of 10kHz before the target dies, but different to Rtai, some periods become very imprecise for higher frequencies.
The low frequency of about maximal 1.2kHz of the PowerPC target depends mostly on the slow I/O port of the hardware.

5.2. Interrupt latency

Unfortunately the time was too short to write an initialization sequence for the PowerPC target which enables the external interrupt so that the interrupt latency benchmark was only working for the x86 target. The graphs on figures 5.2 5.3 and 5.4 show the interrupt latency

of the particular real-time extension. The blue line refers on all three figures to the test-run without any workload and the red one illustrates the latencies under heavy load. As clearly

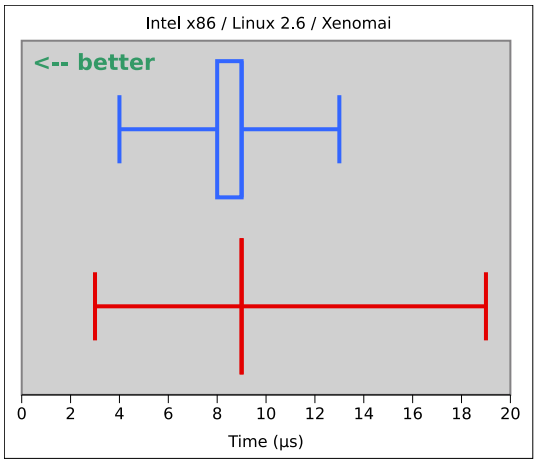

Figure 5.2.: Interrupt latency of Intel x86 architecture with Linux 2.6 and Xenomai

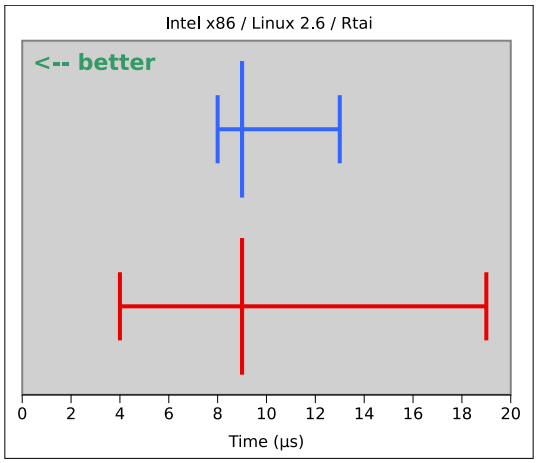

Figure 5.3.: Interrupt latency of x86 architecture with Linux 2.6 and Rtai

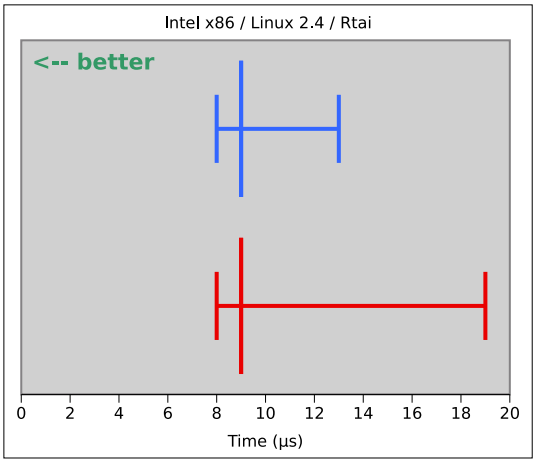

Figure 5.4.: Interrupt latency of Rtai on a x86 target with Linux 2.4

demonstrated by those graphs do all three real-time approaches react on an interrupt in worst case after 19μs. In case of an idling system the interrupts are even handled in 13μs. The average latency of all three platforms is around 8μs and 9μs, even on heavy workload of the system.

5.3. Inter process communication

The transmission speed of the ORF pipe- and device-files which are mainly used to communicate with other processes are displayed on the diagram of figure 5.5 Since this benchmark

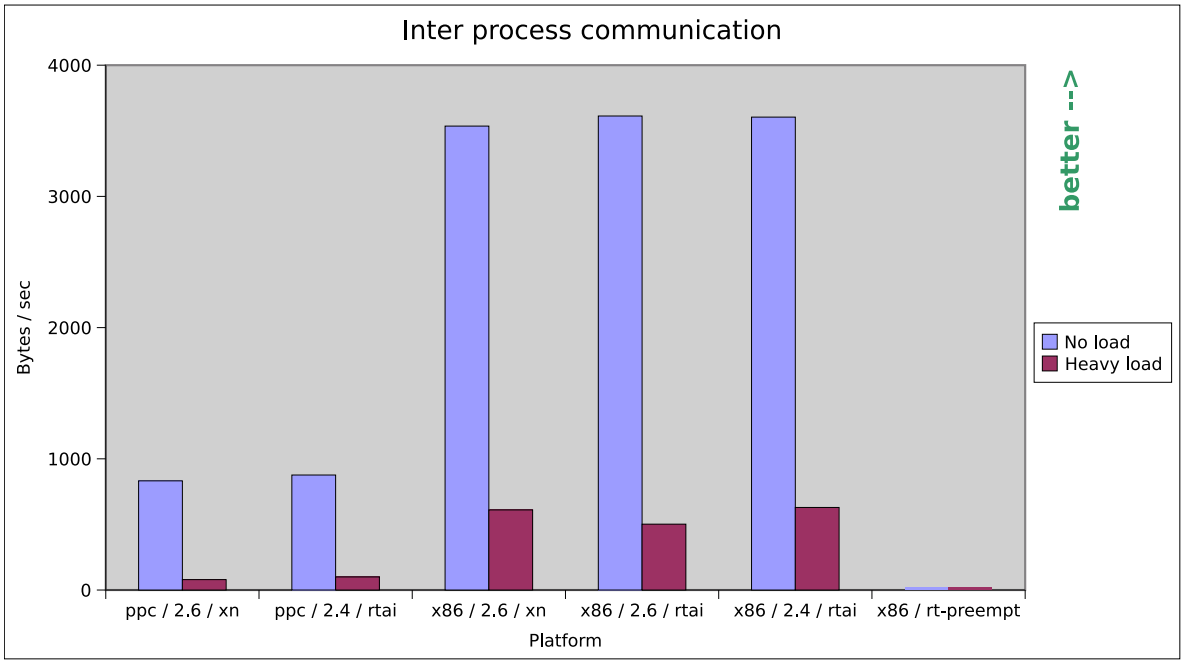

Figure 5.5.: Results of inter-process communication benchmark

mainly depends on how fast the hardware is able to process data the Intel x86 target with its 600MHz and faster bus frequencies achieves a much higher transmission rate than the PowerPC target. In principle doesn't matter whether Xenomai or Rtai or which Linux kernel version is used, because the handling of the character devices is always the same. It just differs, when ORF is running in user-space and pipe files are used, because the pipe files are not handled as soon as any process writes to it. In fact they are polled by ORF as described in chapter 3.5.2. Exactly this applies when using the rt-preempt patch with Linux 2.6 and that's why it has such a bad transmission rate of only 32KiB in 3 seconds. When the polling rate of the pipe files is raised the transmission rate will also rise. But that would be more a workaround than a real solution for providing fast communication between ORF and another process. Much better would be to implement a thread for every pipe-file which just waits for a process writing to it and then handles the request.
The transmission rate of the PowerPC target is about 870KiB/s without workload and 96KiB/s with heavy load. The Intel platform is with 3472KiB/s at system idle and 640KiB/s under heavy load about 4 to 6 times faster than the PowerPC target, when using character files.

5.4. Jitter

Real-time systems offer the ability to create a very periodic shot for waking up the periodic programs. But as these shots cannot be a 100 percent exact they differ to each other. One time the shot comes some micro seconds earlier the next time it may be little bit later. The values in the graphs show the worst difference of those shots on the tested system. The results of the jitter benchmark are divided up into two graphs so that they can be better displayed. Figure (5.6) shows the comparison between PowerPC and normal x86 targets and figure 5.7 compares the results of the geode gx1 target with the results of the normal Intel x86. For

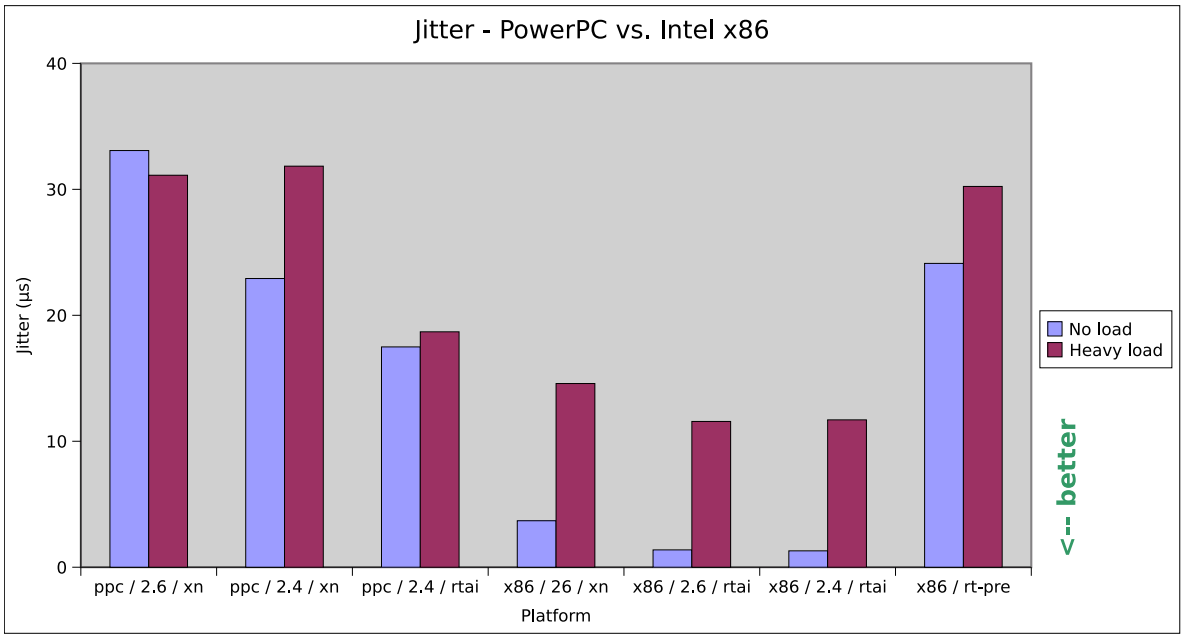

Figure 5.6.: Jitter benchmark - PowerPC vs. Intel x86

the PowerPC target Rtai is very stable with a jitter of about 19μs even when the system is under heavy workload. Xenomai is not that stable, the jitter on the PowerPC target is about 22μs to 33μs. The high jitter of the combination PowerPC, Linux 2.6 and Xenomai without workload seems like a measurement error or that there appeared a disturbing signal like a bouncing output.

On the Intel x86 architecture Rtai seems also to be a little better than Xenomai. The jitter of Rtai is about 2μs when the system is idling and about 12μs when it is under heavy load, while Xenomai's jitter in system's idle state is 4μs and with heavy load about 15μs.

The Linux 2.6 kernel with rt-preempt patch shows with 24μs and 30μs under heavy system load worst jitter on the x86 target, but still better than the Xenomai solution on PowerPC target.

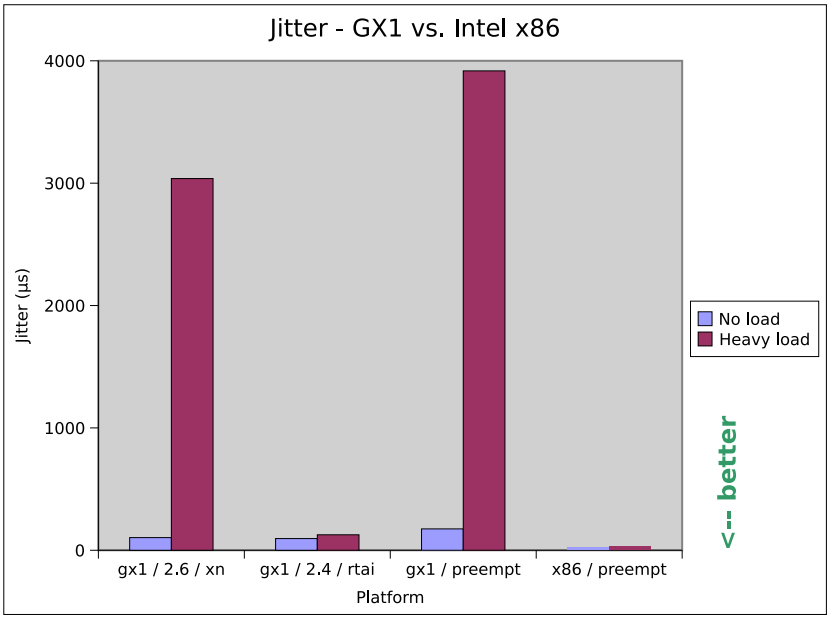

Figure 5.7.: Jitter benchmark - Geode gx1 system

The results of the Geode gx1 target don't look that good at all. The jitter of the Linux kernel 2.6 with Xenomai and the rt-preempt patch could even be reached by the standard Linux kernel without any real-time extension. Both real-time approaches have a jitter of over 3ms when the system is under heavy load. After searching the internet and asking on the mailing list[41] of the rt-preempt patch it turned out that the Geode gx1 system has some software routines within its bios to emulate some x86 features. And those bios routines cause this unreliable behavior. The strange thing about the gx1 target is that it seems to be deterministic with the combination of Rtai and the Linux kernel 2.4. Although the jitter of 96μs when system is idling and 127μs while heavy load are not that good compared to the other targets, it is deterministic and could be used for real-time applications.

5.5. Overload

The overload behavior is a very interesting attribute of a real-time operating system. Especially when the real-time applications needs a lot CPU time, then just a little unexpected signal could produce a short overload of the system. This benchmark helps making a prognosis of the effect of such an overload. As already mentioned in chapter 4.5 are there 2 tests within this benchmark, one simulating light overload and the second one heavy overload. Following table shows the result of the tests on the target systems:

	ppc / 2.6 / xn		x86 / 2.6 / xn		x86 / 2.6 / rtai		x86 / 2.4 / rtai	
	test1	test2	test1	test2	test1	test2	test1	test2
No load	X	-	X	-	X	-	X	-
Heavy load	X	-	X	-	o	-	o	-

X: test passed -: test aborted o: machine died

For the tested platforms with Xenomai the functionality of the RProgs are still given when the light overloads occurs like specified in test 1. But for Rtai this applied only for test 1 when ran under no workload, under heavy workload the machine completely freezes.
The second test, where the overload forces the real-time approach to omit one cycle, the functionality was not given anymore, neither on the x86 platform nor on the ppc hardware. This is related to an overload protection implemented into ORF, which deactivates the RProg when it is running longer than its period duration. Exactly this overload protection applies in the case of the heavy overload. So to make future runs of this benchmark more meaningful there must be an option introduced into ORF which disables this protection while running the benchmark.

5.6. Priority

Priority management is another very important feature a real-time operating system must provide. This benchmark checks for this feature by running 4 threads with different priorities and counting how often any thread is preempted and by which thread. The following table shows the results of the Intel x86 target with Linux kernel 2.6 and Xenomai:

	No load				Heavy load			
	T0	T1	T2	T3	T0	T1	T2	T3
T0	0	0	0	0	0	0	0	0
T1	932	0	0	0	903	0	0	0
T2	922	104	0	0	925	98	0	0
T3	4480	486	40	0	4531	503	48	0

The threads listed in columns preempt the threads listed in the rows. For example thread T2 preempts thread T3 40 times when the system idles and 48 times when the system is under heavy load. Thread T0 has highest priority and T3 has lowest priority.
In summary this table looks like it should look, the threads T1 to T3 are preempted by every other thread having higher priority. T0 has highest priority, so it cannot be preempted.

The same test has been performed also for the PowerPC architecture using Linux 2.6 with Xenomai. But when running the test it turned out that the I/O - hardware of the system is not as fast as the parallel-port of the x86 computer. So the RProg running in T0 was not able to set the output pins correctly. Thus T0 has been disabled for testing the PowerPC target. Here is the full table containing the results:

	No load				Heavy load			
	T0	T1	T2	T3	T0	T1	T2	T3
T0	-	-	-	-	-	-	-	-
T1	-	0	0	0	-	0	0	0
T2	-	18	0	5	-	24	0	1
T3	-	74	54	0	-	84	56	0

There is a problem with this target. Thread T2 gets preempted by T3 which has lower priority. That should not happen. Either there is a bug within Xenomai for this target or the most probably reason is that the I/O-hardware produces again a problem.
To find the reason of this problem the log file of the benchmark must be manually searched for the state in which T2 is preempted by T3. This state applies, when T2 and T3 are active (bit 2 and 3 set), and T3 is running (bit 7 set). Thus the state which must be searched is "0x8C" or 140 in decimal. And in fact, this state can be found in the log file. But just up to 8 of those states in a row and additionally between two valid states. Normally a valid state is represented within the log file as at least 50 values in a row. Thus this problem is related to the I/O-hardware which is not able to switch the pins that fast and produces an invalid state between two valid states.
The benchmark test must to be improved to filter such invalid states caused by too slow hardware to get better results in the future.

Chapter 6.

Conclusion

This project shows, how toolchains for different target systems like Intel x86, PowerPC and ARM are created. It further demonstrates how the real-time approaches Rtai, Xenomai and the rt-preempt patch are compiled within these toolchains and how they are installed on the particular targets.
It basically explains the functionality of the Open Realtime Framework and illustrates what changes were made to allow benchmarking on top of it. The changes of ORF include loading and unloading dynamical linked libraries while runtime, Linux character devices for communication when ran in kernel space, an I/O-API and interrupt handling. Besides those changes ORF has been ported while this project to the PowerPC architecture, to run with Linux kernel 2.6 and to work together with Xenomai.
Furthermore 6 benchmark tests were developed and implemented. While each benchmark consists of ORF implementations and of an application measuring with an usb scope and analyzing the data. The benchmark evaluate important attributes of real-time systems like the jitter, interrupt latency, inter-process communication and more.

Although the time was too short to run all benchmarks on all platforms, a qualitive comparison from a very practical point of view can already be made. The good old Rtai and Linux kernel 2.4 seem to be still the best solution when highest reliability, determinism and good portability is required. But the final application must be tested very intensively, because as the overload benchmark shows does the Rtai system freeze as soon as little overload occur.
Since the rt-preempt patch is still in the very beginning, it does already demonstrate hard real-time capabilities and arouses curiosity about future use-cases of Linux. Xenomai however is a state of the art solution which brings latest technology, like real-time applications in user-space and skin support. It lacks a little bit in performance and precision compared to Rtai but is more stable in case of overloads.

The current state of the project can be used to create qualitative and quantitative evaluations of target systems. Testing one platform with all the benchmarks needs about one

working day and shows exactly the practical capability of the target system as well as its limits. Thus this is an excelent facility to demonstrate the client who orders an embedded real-time system at the very beginning of a project if the proposed hardware will fit the needs and which real-time solution fulfills the requirements best.

Most important characteristics of a real-time system are already covered by the implemented benchmarks, but in the future some experience in those benchmarks can be collected to get even more precise results with less disturbances. To further expand the operational area of this project, the adoption of following enhancements are imaginable:

- ARM support could be added.
- Interrupt handling of PowerPC target so that a comparison of the interrupt latency benchmark can be made architecture independent.
- A solution to use interrupts within a user-space program running with the rt-preempt patch could be established.
- Further benchmarks, like testing semaphores of the real-time approaches, could be introduced.

In summary the goal of this diploma thesis has been successfully accomplished and the listed enhancements show that this project leads to a benchmark environment which is able to test every attribute of any real-time system from a practical point of view.

Appendix A.

Bibliography

[1] Rtai - Homepage of the Rtai project.
URL https://www.rtai.org

[2] Paolo Mantegazza - History of Rtai
URL http://www.aero.polimi.it/ rtai/documentation/articles/history/

[3] Xenomai - Homepage of the Xenomai project.
URL http://www.xenomai.org

[4] Xenomai - History of Xenomai.
URL http://www.xenomai.org/index.php/Xenomai:Roadmap

[5] Prof. Dr. Schlegel - notes of real-time lecute.

[6] Rt-preempt - A wiki containing information about the RT-preempt patch.
URL http://rt.wiki.kernel.org/index.php/Main_Page

[7] Klaas van Gend - Presentation about the RT-preempt patch hold on the FOSDEM 2006.
URL www.opentux.nl/artikelen/FOSDEM2006-RT_patches.pdf

[8] Cirrus Logic - Information about the EP9315 SOC processor.
URL http://www.cirrus.com/en/products/pro/detail/P1052.html

[9] Red Hat - RedBoot User's Guide.
URL http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html

[10] Meilhaus - Mephisto scope document and tools.
URL http://www.meilhaus.de/service/download/inhalte/mephisto-scope-um202-um203

[11] Innotek - Virtualbox projectpage.
URL http://virtualbox.org/

[12] ORF - Homepages of the ORF project.
URL http://www.o-r-f.org/

[13] Hermann Betz, Yellowstone Soft - Specification of ORF.
URL http://www.o-r-f.org/pdf/orf_spezi_0231.pdf

[14] Yellowstone soft - SofCoS info page.
URL http://yellowstone-soft.de/engl/yssofcos.htm

[15] Hermann Betz, Yellowstone Soft - Coryo info page.
URL http://yellowstone-soft.de/engl/yscoryo.htm

[16] Patrick Reinwald, Frenco - Praxissemesterbericht SS 2006.

[17] wxDev-C++ - Homepage of the open-source IDE for Windows.
URL http://wxdsgn.sourceforge.net

[18] DevC++ - Homepage of the DevC++ project.
URL http://bloodshed.net/dev/devcpp.html

[19] wxWidgets - Homepage of the wxWidgets project.
URL http://wxwidgets.org/

[20] MinGW - Homepage of the Minimalist GNU for Windows project.
URL http://www.mingw.org/

[21] Robert Schwebel, Pengutronix - Info page about the ptxdist build system.
URL http://www.pengutronix.de/software/ptxdist/index_en.html

[22] Thomas Petazzoni - Buildroot documentation.
URL http://buildroot.uclibc.org/buildroot.html

[23] ArchLinux - ArchLinux homepage.
URL http://www.archlinux.org

[24] Cirrus Logic - Repository containing ARM toolchain.
URL http://arm.cirrus.com/

[25] DENX Software Engineering - ELDK manual.
URL http://www.denx.de/wiki/DULG/ELDK

[26] Rtai - Rtai installation manual.
URL https://www.rtai.org/RTAILAB/RTAI-TARGET-HOWTO.txt

[27] Rtai - Readme files contained in the Rtai package.

[28] Adeos - Adeos project page.
URL http://www.adeos.org

[29] Grub - Grub manual.
URL http://orgs.man.ac.uk/documentation/grub/

[30] Romain Lenglet - Installation of Xenomai real time Linux.
URL http://www.csg.is.titech.ac.jp/ lenglet/howtos/realtimelinuxhowto/

[31] Xenomai - Readme files within the Xenomai source package.

[32] Ingo Molnar, Redhat - Rt-preempt patch repository.
URL http://people.redhat.com/mingo/realtime-preempt/

[33] Unknown - Unofficial Linux 2.6.x support for Cirrus EP93xx processors.
URL http://web.archive.org/web/20070417081212/http://members.inode.at/m.burian/ep93xx/

[34] Conitec Datensysteme - ARM & EVA Linux development system.
URL http://www.conitec.net/english/linuxboard.htm

[35] Andrew Victor - AT91RM9200 Linux 2.6 Patches.
URL http://maxim.org.za/at91_26.html

[36] Xenomai - Xenomai FAQ page.
URL http://www.xenomai.org/index.php/FAQs

[37] Rtai - Rtai mailing list, support for AT91RM9200 CPU.
URL https://mail.rtai.org/pipermail/rtai/2005-September/012847.html

[38] DENX Software Engineering - Git server.
URL http://www.denx.de/en/Software/GIT

[39] DENX Software Engineering - u-Boot manual.
URL http://www.denx.de/wiki/view/DULG/UBootInstallUsingUBoot

[40] Thomas Gleixner + Greg KH - User-space interface for PCI drivers.
URL http://lkml.org/lkml/2006/8/30/22

[41] RT mailing list - Geode GX1 + RT-Preempt.
URL http://www.mail-archive.com/linux-rt-users@vger.kernel.org/msg00333.html

Appendix B.

Glossary

Big kernel lock The so called BKL was introduced in order to lock critical sections of the kernel when using a multiprocessor system. In general it is a recursive spinning lock which will not end up in a deadlock when two consecutive requests occur.

Bootloader A bootloader typically initializes the hardware either directly after power on (very common for embedded systems) or after the bios is executed (usual for Intel x86 computers). It specifies which operating system should be booted. Additional features of bootloader are e.g. starting the operating system with custom parameters or downloading the kernel from a network server before starting.

Embedded system An embedded system is a computer built for one special purpose. That brings the advantage that the system is optimized exactly for the application and unused hardware can be removed to reduce the cost of the product.

FBD Function Block Diagram is a graphical programming language for PLC's defined in the IEC 61131-3 standard.

GDB The GNU DeBugger is the standard debugger for GNU systems. It supports various languages, architectures and many features like remote debugging, where the debugger runs on the target system but is controlled by a remote development system. The debugger itself is console based, but as it offers a communication interface there are many graphical front-ends like ddd.

Git Git is a revision management tool for text files. It was developed by Linus Torvalds since every other revision control program like SVN or BitKeeper did not meet the needs for managing the Linux kernel sources or is licensed under an unsuitable license.

GPL The GNU General Public License is a very popular free software license. It was introduced and is still maintained by Richard Stallman and the free software foundation. Although it was meant to be used for the GNU system in the first place, most open source projects are licensed under the GPL today. The GPL guarantees that derived works of a project, which is published under GPL has to be available under the same copyleft.

IDE An Integrated Development Environment is a collection of tools needed to develop software or hardware. The IDE installation contains all the tools in one package and connects the tools in one intuitive user interface.

IEC 61131-3 IEC 61131-3 is the third part of the International Electrotechnical Commission standard 61131. It defines 5 programming languages for PLC's and their components.

IL Instruction list is a programming language for PLC's very similar to the assembly language. This programming language is specified within the IEC 61131-3 standard.

JTAG JTAG is a standardized port which allows debugging of the target system directly on the hardware. All registers and memory addresses of the target system can be read or written without interrupting the program running on the hardware.

Minix Minix is an operating system which has been developed by Andrew Tanenbaum since 1987. It was released under a closed source license and could be used by paying a small fee. He used the system also to teach students the principles of an operating system. But since the GNU/Linux community has become so big he published minix under an open license to get more publicity.

MMU A Memory Management Unit is a piece of hardware of a computer what divides the physical memory into virtual memory pages and translates the addresses.

NFS The Network File System is a very common protocol to share files or even complete filesystems over a network connection. A nfs client is completely implemented into the Linux kernel, what allows to boot up from a remote root filesystem.

Nice The nice value is the priority of a Unix process within the classic scheduler. Nice values are defined from -20 (highest priority) and +19 (lowest priority). The nice value of any process can be modified using the renice program.

PI mutex A priority inheritance mutex allows a task with lower priority which blocks a resource to inherit the priority of a high priority task, when it requests this blocked resource. This way the low priority task cannot be preempted by a medium task and the high priority task gets the resource as soon as possible.

PLC A Programmable Logic Controller is a computer mainly used for automation of industrial processes. It mostly contains many input and output ports and a bus system to which further devices can be easily attached. PLC's work usually in a cyclic execution of the three steps, reading inputs, calculating outputs and setting the output signals.

Read-write lock Since it is save for multiple threads to read from data, but just allowed to write data when only one thread is accessing it, this locks were introduced. They are used as normal spinlocks but allowing several readers to read from critical data. While writing excludes every thread but the writing one.

SOC SOC stands for System On Chip and means the inclusion of a complete computer with every needed piece of hardware into one single chip. The advantages are the very compact size of the computer and short communication timings of the various devices, as the signals don't have to leave the chip.

Spinlock A spinlock is a lock where a thread waits until the lock becomes available. It is the most common way to protect critical sections within the Linux kernel.

ST Structured text is a textual programming language for PLC's defined within the IEC 61131-3 standard. It can be compared with e.g. the high level programming language C.

Tftp The Trivial File Transfer Protocol is a network protocol especially designed for transmitting files. A tftp server application serves a directory on a network computer while a tftp client can download or upload files.

Appendix C.

Listings

Listing C.1: rt_para.c

```
#ifdef __KERNEL__
#include <linux/module.h>
#include <linux/kernel.h>
#else
#include <stdio.h>
#include <stdlib.h>
#endif

#include <include/orf.h>
#include <include/orf_methods.h>

#ifdef __KERNEL__
#define PRINT printk
#else
#define PRINT printf
#endif

/*-----------------------------------------------------------------*/

/*
 * Idle function of the program
 */
int para_idle(int device,int id,long para)
{
   unsigned char data=0;
   /* increment the data in the I/O register - creates squarewave on the I/O */
   data = orf_inb();
   orf_outb(++data);
   return 0;
}
/* end rprog_idle() */


/*-----------------------------------------------------------------*/

#ifdef __KERNEL__
static int init(void)
#else
int init(void)
#endif
{
  /* register idlefunction as sysfunction in ORF */
```

```
    PRINT("Initialisierung des rt-para-Moduls\n");
    orf_add_sysfunct("PARA_IDLE",(void*)para_idle);
    PRINT("sysfunct added\n");
    /* initialize IO */
#ifdef __KERNEL__
    orf_initio();
    PRINT("io initialized\n");
#endif /* KERNEL */
    return(0);
}

/* end init() */

/*-----------------------------------------------------------------*/

/*
 * initialization of the module
 */
int init_module()
{
    /* register init function within orf */
    orf_add_initfunct((void*)init,"PARA");
    return(0);
}

/*-----------------------------------------------------------------*/

void cleanup_module()
{
    /* delete idle function from ORF */
    orf_delete_sysfunct("PARA_IDLE");
    /* delete init function from ORF */
    orf_delete_initfunct((void*)init,NULL);
    return;
}
```

Listing C.2: Init.ini

```
# INI-File for starting ORF with realtime thread and the rt_para
# program which reads data from I/O increments and ouputs the data
# back on I/O

# for shared memory which is not managed by linux, dont forget to boot
# linux with mem=0x7400000 parameter

# Open Shared Memory
ORF_SHM_OPEN;7400000;1

#Initialize shared memory
ORF_INIT_ZEROPAGE;2

# Initialize page 0, so that device 0 can use it
ORF_INIT_PAGE;0;0

# execute all known RProg init functions
ORF_DO_INITFUNCT;*

# Create Thread 0 for Page 0 with priority 5 and periodic with 0x4a60 us
ORF_CREATE_THREAD0;0;5;4a60;1

# Create Realtime Progs
# starting prog para_idle on thread0
ORF_CREATE_RPROG;0;1;0;PARA_IDLE;  ;  ;

# Start PLC
ORF_START_PLC;0
```

Listing C.3: List of packages

```
bash 3.1.17-1
binutils 2.16.1-2
coreutils 5.2.1-5
cracklib 2.8.3-4
db 4.3.29-2
dcron 3.1-3
e2fsprogs 1.38-3
filesystem 0.7.2-2
findutils 4.2.27-1
gawk 3.1.5-3
glibc 2.3.6-3
grep 2.5.1a-2
initscripts 0.7.2-1
logrotate 3.7.1-1
mailx 8.1.1-3
mktemp 1.5-1
ncurses 5.5-1
net-tools 1.60-10
pam 0.81-1
pcre 6.6-3
popt 1.7-3
procps 3.2.6-1
readline 5.1.4-1
sed 4.1.5-1
shadow 4.0.14-1
syslog-ng 1.6.10-1
sysvinit 2.86-2
tar 1.15.1-2
tcp_wrappers 7.6-6
udev 091-6
util-linux 2.12-9
zlib 1.2.3-1
```

Appendix D.

License

This work is licensed under a Creative Commons Attribution-Share Alike 3.0 License.

You are free:

- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:

- Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

This is a human-readable summary of the Legal Code of the license:
http://creativecommons.org/licenses/by-sa/3.0/legalcode